Genero Business Development
Language User Guide
Contents

Genero BDL User Guide................................................................................................................ 17

What's new in 3.20.......................................................................................................................... 19

General........................................................................................................................................ 25
  Documentation conventions........................................................................................................ 25
  Syntax diagrams....................................................................................................................... 25
  Warnings, notes and tips........................................................................................................... 26
  Code examples.......................................................................................................................... 27
  General terms used in this documentation............................................................................. 27
  Other source of information...................................................................................................... 28
  Introduction to Genero BDL programming............................................................................. 28
  Overview of Genero BDL......................................................................................................... 28
  Genero BDL concepts................................................................................................................ 30

Installation.................................................................................................................................. 37
  Resources for upgrades.......................................................................................................... 37
  Software requirements.............................................................................................................. 37
  Supported operating systems................................................................................................... 37
  System packages...................................................................................................................... 37
  Database client software......................................................................................................... 38
  C compiler for C extensions..................................................................................................... 38
  Java runtime environment for FGL.......................................................................................... 38
  IPv6 support with Genero........................................................................................................ 39
  Installing Genero BDL.............................................................................................................. 39
  Upgrading Genero BDL............................................................................................................ 40
  Install Genero Mobile for Android®....................................................................................... 40
  Install Genero Mobile for iOS (single version)......................................................................... 42
  Install Genero Mobile for iOS (multiple versions).................................................................. 42
  Platform specific notes............................................................................................................ 43
  HP-UX configuration notes...................................................................................................... 43
  IBM® AIX® configuration notes............................................................................................... 44
  Mac® OS X® configuration notes............................................................................................ 44
  Microsoft® Windows® configuration notes............................................................................ 46
  Web Services platform notes.................................................................................................. 47
  SUSE® Linux Enterprise.......................................................................................................... 47
  Debian / Ubuntu Linux............................................................................................................ 47
  Red Hat® Enterprise Linux® / CentOS..................................................................................... 47

Upgrading.................................................................................................................................... 48
  New features of Genero BDL................................................................................................... 48
  BDL 3.20 new features............................................................................................................ 48
  BDL 3.10 new features............................................................................................................ 54
  BDL 3.00 new features............................................................................................................ 61
  BDL 2.51 new features............................................................................................................ 66
FAQ005: Why doesn't the ESC key validate my input? ................................................................. 252
FAQ006: Why doesn't the Ctrl-C key cancel my input? ................................................................. 253
FAQ007: Why do the gui.* FGLPROFILE entries have no effect? .................................................. 253
FAQ008: Why do I get invalid characters in my form? ................................................................. 253
FAQ009: Why do large static arrays raise a stack overflow? ......................................................... 254
FAQ010: Why do I get error -6366 "Could not load database driver drivername"? ...................... 254

Configuration .......................................................................................................................... 255
  The FGLPROFILE file(s) ........................................................................................................ 255
  Understanding FGLPROFILE ................................................................................................. 255
  Syntax of FGLPROFILE entries ............................................................................................. 255
  FGLPROFILE entries for core language ................................................................................ 256
  FGLPROFILE files selection .................................................................................................. 260
  FGLPROFILE encoding .......................................................................................................... 261
  The default FGLPROFILE ....................................................................................................... 261
  FGLPROFILE for mobile apps ................................................................................................ 261
  Reading FGLPROFILE entries ............................................................................................... 261
  Environment variables ........................................................................................................... 262
  Setting environment variables on UNIX™ ............................................................................. 262
  Setting environment variables on Windows™ ........................................................................ 262
  Setting environment variables in FGLPROFILE (mobile) ....................................................... 263
  Operating system environment variables .............................................................................. 264
  Database client environment variables ................................................................................ 266
  Genero environment variables .............................................................................................. 266
  Front-end connection ............................................................................................................. 285
  Database server connections ................................................................................................. 285

Language basics ....................................................................................................................... 285
  Syntax features ....................................................................................................................... 285
  Lettercase insensitivity .......................................................................................................... 285
  Whitespace separators .......................................................................................................... 286
  Quotation marks .................................................................................................................... 286
  Escape symbol ...................................................................................................................... 287
  Statement terminator ............................................................................................................ 287
  Source comments ................................................................................................................ 287
  Identifiers ............................................................................................................................. 289
  Preprocessor directives ........................................................................................................ 289
  Primitive Data types ............................................................................................................ 289
    BIGINT ............................................................................................................................... 290
    BYTE ................................................................................................................................. 290
    BOOLEAN .......................................................................................................................... 293
    CHAR(size) ....................................................................................................................... 293
    DATE ................................................................................................................................. 295
    DATETIME qual1 TO qual2 ............................................................................................... 296
    DECIMAL(p,s) ................................................................................................................... 298
    FLOAT .............................................................................................................................. 300
    INTEGER .......................................................................................................................... 300
    INTERVAL qual1 TO qual2 .............................................................................................. 301
    MONEY(p,s) ..................................................................................................................... 303
    SMALLFLOAT .................................................................................................................. 304
    SMALLINT ....................................................................................................................... 304
    STRING ............................................................................................................................ 305
    TINYINT ........................................................................................................................... 306
    TEXT ................................................................................................................................. 307
<table>
<thead>
<tr>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>VALIDATE............................................................................................................................................... 403</td>
</tr>
<tr>
<td>Examples............................................................................................................................................... 404</td>
</tr>
<tr>
<td>Constants.............................................................................................................................................. 405</td>
</tr>
<tr>
<td>Understanding constants........................................................................................................... 405</td>
</tr>
<tr>
<td>CONSTANT........................................................................................................................................... 406</td>
</tr>
<tr>
<td>Examples............................................................................................................................................... 408</td>
</tr>
<tr>
<td>Records................................................................................................................................................. 408</td>
</tr>
<tr>
<td>Understanding records................................................................................................................... 409</td>
</tr>
<tr>
<td>RECORD.............................................................................................................................................. 409</td>
</tr>
<tr>
<td>Record initializers..................................................................................................................... 410</td>
</tr>
<tr>
<td>Record definition attributes.............................................................................................................. 411</td>
</tr>
<tr>
<td>Accessing record members............................................................................................................ 412</td>
</tr>
<tr>
<td>Copying records.......................................................................................................................... 412</td>
</tr>
<tr>
<td>Comparing records...................................................................................................................... 414</td>
</tr>
<tr>
<td>Records as function parameters.......................................................................................................... 415</td>
</tr>
<tr>
<td>THRU/THROUGH...................................................................................................................... 417</td>
</tr>
<tr>
<td>Examples............................................................................................................................................... 417</td>
</tr>
<tr>
<td>Arrays................................................................................................................................................... 418</td>
</tr>
<tr>
<td>Understanding arrays........................................................................................................................ 418</td>
</tr>
<tr>
<td>ARRAY......................................................................................................................................... 419</td>
</tr>
<tr>
<td>Static arrays............................................................................................................................... 420</td>
</tr>
<tr>
<td>Dynamic arrays............................................................................................................................ 422</td>
</tr>
<tr>
<td>Array methods............................................................................................................................... 424</td>
</tr>
<tr>
<td>Copying and assigning arrays............................................................................................................... 424</td>
</tr>
<tr>
<td>Examples............................................................................................................................................... 426</td>
</tr>
<tr>
<td>Dictionaries......................................................................................................................................... 426</td>
</tr>
<tr>
<td>Understanding dictionaries.................................................................................................................. 426</td>
</tr>
<tr>
<td>DICTIONARY............................................................................................................................ 427</td>
</tr>
<tr>
<td>DICTIONARY in action..................................................................................................................... 428</td>
</tr>
<tr>
<td>Dictionary methods.......................................................................................................................... 430</td>
</tr>
<tr>
<td>Examples............................................................................................................................................... 430</td>
</tr>
<tr>
<td>Types.................................................................................................................................................... 431</td>
</tr>
<tr>
<td>Understanding type definition.......................................................................................................... 431</td>
</tr>
<tr>
<td>TYPE............................................................................................................................................. 432</td>
</tr>
<tr>
<td>Using types in programs................................................................................................................ 433</td>
</tr>
<tr>
<td>Examples............................................................................................................................................... 434</td>
</tr>
<tr>
<td>Functions............................................................................................................................................. 436</td>
</tr>
<tr>
<td>Understanding functions.................................................................................................................. 436</td>
</tr>
<tr>
<td>FUNCTION definitions...................................................................................................................... 437</td>
</tr>
<tr>
<td>Scope of a function....................................................................................................................... 440</td>
</tr>
<tr>
<td>Local symbol definition.................................................................................................................. 441</td>
</tr>
<tr>
<td>Calling functions............................................................................................................................. 442</td>
</tr>
<tr>
<td>Function parameters........................................................................................................................ 444</td>
</tr>
<tr>
<td>Returning values............................................................................................................................ 446</td>
</tr>
<tr>
<td>Function references........................................................................................................................ 449</td>
</tr>
<tr>
<td>FUNCTION func-spec................................................................................................................ 451</td>
</tr>
<tr>
<td>Methods......................................................................................................................................... 452</td>
</tr>
<tr>
<td>Function attributes....................................................................................................................... 455</td>
</tr>
<tr>
<td>Examples............................................................................................................................................... 456</td>
</tr>
<tr>
<td>Interfaces........................................................................................................................................... 457</td>
</tr>
<tr>
<td>Understanding interfaces................................................................................................................ 457</td>
</tr>
<tr>
<td>INTERFACE............................................................................................................................... 457</td>
</tr>
<tr>
<td>INTERFACE usage...................................................................................................................... 459</td>
</tr>
<tr>
<td>Examples............................................................................................................................................... 460</td>
</tr>
</tbody>
</table>
# Contents

<table>
<thead>
<tr>
<th>Advanced features</th>
<th>462</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program modules.</td>
<td>462</td>
</tr>
<tr>
<td>Structure of a program</td>
<td>462</td>
</tr>
<tr>
<td>Structure of a module</td>
<td>462</td>
</tr>
<tr>
<td>The MAIN block / function</td>
<td>464</td>
</tr>
<tr>
<td>Database schema</td>
<td>467</td>
</tr>
<tr>
<td>Understanding database schemas</td>
<td>467</td>
</tr>
<tr>
<td>SCHEMA</td>
<td>468</td>
</tr>
<tr>
<td>Structure of database schema files</td>
<td>470</td>
</tr>
<tr>
<td>Database schema extractor options</td>
<td>476</td>
</tr>
<tr>
<td>Importing modules</td>
<td>480</td>
</tr>
<tr>
<td>IMPORT C-Extension</td>
<td>481</td>
</tr>
<tr>
<td>IMPORT FGL module</td>
<td>481</td>
</tr>
<tr>
<td>IMPORT JAVA classname</td>
<td>487</td>
</tr>
<tr>
<td>Program execution</td>
<td>488</td>
</tr>
<tr>
<td>Executing programs</td>
<td>488</td>
</tr>
<tr>
<td>RUN</td>
<td>490</td>
</tr>
<tr>
<td>EXIT PROGRAM</td>
<td>492</td>
</tr>
<tr>
<td>BREAKPOINT</td>
<td>493</td>
</tr>
<tr>
<td>Runtime stack</td>
<td>493</td>
</tr>
<tr>
<td>Passing simple typed values as parameter</td>
<td>494</td>
</tr>
<tr>
<td>Passing records as parameter</td>
<td>494</td>
</tr>
<tr>
<td>Passing static arrays as parameter</td>
<td>496</td>
</tr>
<tr>
<td>Passing dynamic arrays as parameter</td>
<td>497</td>
</tr>
<tr>
<td>Passing dictionaries as parameter</td>
<td>497</td>
</tr>
<tr>
<td>Passing objects as parameter</td>
<td>498</td>
</tr>
<tr>
<td>Passing TEXT/BYTE values as parameter</td>
<td>499</td>
</tr>
<tr>
<td>Returning simple typed values from functions</td>
<td>499</td>
</tr>
<tr>
<td>Returning records from functions</td>
<td>499</td>
</tr>
<tr>
<td>Returning dynamic arrays from functions</td>
<td>500</td>
</tr>
<tr>
<td>Returning dictionaries from functions</td>
<td>500</td>
</tr>
<tr>
<td>Returning TEXT/BYTE values from functions</td>
<td>501</td>
</tr>
<tr>
<td>Implicit data type conversion on the stack</td>
<td>501</td>
</tr>
<tr>
<td>Exceptions</td>
<td>502</td>
</tr>
<tr>
<td>Understanding exceptions</td>
<td>502</td>
</tr>
<tr>
<td>Exception classes</td>
<td>502</td>
</tr>
<tr>
<td>Exception actions</td>
<td>502</td>
</tr>
<tr>
<td>WHENEVER instruction</td>
<td>503</td>
</tr>
<tr>
<td>TRY - CATCH block</td>
<td>506</td>
</tr>
<tr>
<td>Tracing exceptions</td>
<td>508</td>
</tr>
<tr>
<td>Default exception handling</td>
<td>508</td>
</tr>
<tr>
<td>Non-trappable errors</td>
<td>509</td>
</tr>
<tr>
<td>Examples</td>
<td>510</td>
</tr>
<tr>
<td>Localization</td>
<td>512</td>
</tr>
<tr>
<td>Application locale</td>
<td>512</td>
</tr>
<tr>
<td>Localized strings</td>
<td>538</td>
</tr>
<tr>
<td>Globals</td>
<td>551</td>
</tr>
<tr>
<td>Understanding global blocks</td>
<td>551</td>
</tr>
<tr>
<td>GLOBALS</td>
<td>552</td>
</tr>
<tr>
<td>Rules for globals usage</td>
<td>552</td>
</tr>
<tr>
<td>Database schema in globals</td>
<td>553</td>
</tr>
<tr>
<td>Content of a globals file</td>
<td>553</td>
</tr>
<tr>
<td>Examples</td>
<td>554</td>
</tr>
<tr>
<td>Predefined constants</td>
<td>555</td>
</tr>
</tbody>
</table>
User interface ...................................................................................................................................... 1105
User interface basics.......................................................................................................................... 1105
The dynamic user interface.............................................................................................................. 1105
The abstract user interface tree......................................................................................................... 1106
Genero user interface modes.............................................................................................................. 1109
GUI front-end connection.................................................................................................................... 1116
Using a text terminal.......................................................................................................................... 1123
Form definitions................................................................................................................................ 1131
Windows and forms............................................................................................................................ 1131
Using images...................................................................................................................................... 1147
Accessibility guidelines....................................................................................................................... 1157
Message files...................................................................................................................................... 1160
Action defaults files............................................................................................................................ 1162
Presentation styles............................................................................................................................... 1165
Form specification files....................................................................................................................... 1237
Form rendering..................................................................................................................................... 1415
Toolbars............................................................................................................................................... 1436
Topmenus............................................................................................................................................ 1444
Dialog instructions.............................................................................................................................. 1452
Static display (DISPLAY/ERROR/MESSAGE/CLEAR).......................................................................... 1452
Prompt for values (PROMPT).............................................................................................................. 1462
Ring menus (MENU). ........................................................................................................................... 1467
Record input (INPUT).......................................................................................................................... 1482
Record list (DISPLAY ARRAY)........................................................................................................... 1501
Programming tools ........................................................................................................ 2065

| Contents |
|----------|---|
| AVG      | 2062 |
| SUM      | 2062 |
| MIN      | 2063 |
| MAX      | 2064 |
| Report engine configuration ................................................................. 2064

Command reference ....................................................................................... 2065
- fglrun ........................................ 2065
- fglform .................................. 2068
- fgl2p .................................... 2070
- fglcomp .................................. 2071
- fgllink ................................... 2076
- fglmkmsg ................................ 2077
- fglmkext ................................ 2077
- fgldb .................................... 2078
- fgl dbsch ................................ 2079
- fglmkstr ................................ 2080
- fglwSDL .................................. 2081
- fglrestful ................................. 2085
- fglpass .................................. 2087
- fglWrt .................................... 2088
- fglgar ................................... 2090
- fglgifformat .............................. 2093
- fglformatdiff .............................. 2094
- fpi ....................................... 2095
- gmabuildtool .............................. 2096
- gmibuildtool .............................. 2102

Compiling source files .................................................................................. 2106
- Compiling form specification files (.per) .................................................. 2106
- Compiling message files (.msg) ................................................................. 2107
- Compiling string resource files (.str) ......................................................... 2107
- Compiling program code files (.4gl) ......................................................... 2108
- Importing modules .................................................................................... 2113
- Linking libraries ........................................................................................ 2113
- Linking programs ...................................................................................... 2115
- Using makefiles .......................................................................................... 2119
- 42m module information ............................................................................ 2120

Source code edition ....................................................................................... 2123
- Choosing the correct locale ........................................................................ 2123
- Avoid Tab characters in screen layouts ..................................................... 2123
- Configure VIM for Genero BDL ................................................................. 2123

Source documentation .................................................................................... 2126
- Understanding source code documentation .............................................. 2126
- Prerequisites to produce documentation .................................................. 2126
- Source documentation structure ................................................................. 2126
- Adding comments to sources .................................................................... 2127
- Run the documentation generator .............................................................. 2132

Source preprocessor ....................................................................................... 2132
- Understanding the preprocessor ............................................................... 2132
- Compilers command line options ............................................................... 2133
- File inclusion .............................................................................................. 2134
- Conditional compilation ............................................................................ 2136
- Simple macro definition ............................................................................. 2136
Extending the language ...................................................................................... 2192
   The Java interface .......................................................................................... 2192
      Prerequisites and installation........................................................................ 2192
      Getting started with the Java interface...................................................... 2196
      Advanced programming.............................................................................. 2197
      Examples........................................................................................................ 2225
   C-Extensions...................................................................................................... 2230
      Understanding C-Extensions........................................................................ 2230
      Header files for ESQL/C typedefs............................................................... 2230
      Creating C-Extensions............................................................................... 2231
      Creating Informix® ESQL/C Extensions...................................................... 2232
      The C interface file...................................................................................... 2233
      Linking programs using C-Extensions....................................................... 2234
Library reference................................................................. 2268
  Built-in functions.......................................................... 2268
  Built-in functions.......................................................... 2270
  List of desupported built-in functions.............................. 2300
  The key code table.......................................................... 2300
  Utility modules............................................................... 2301
  fgldialog: Common dialog functions.................................. 2302
  fgldbutil: Database utility module.................................... 2307
  fglwineexec: Front-end dialogs module............................. 2312
  VCard: VCF file format module........................................ 2316
  fglgallery: Image gallery module...................................... 2322
  fglsvgcanvas: SVG drawing module.................................. 2332
  getopt: Command line options module.............................. 2365
  Built-in packages.......................................................... 2378
  BDL data types package.................................................. 2378
  The base package.......................................................... 2406
  The ui package............................................................... 2470
  The om package............................................................. 2577
  Built-in front calls........................................................ 2635
  Built-in front calls summary.......................................... 2636
  Standard front calls........................................................ 2647
  Web component front calls.............................................. 2666
  Genero Desktop Client front calls.................................... 2668
  Theme front calls........................................................... 2691
  Local storage front calls.................................................. 2695
  Genero Mobile common front calls................................... 2697
  Genero Mobile Android™ front calls................................. 2713
  Genero Mobile iOS front calls........................................... 2718
  Cordova plugin front calls............................................... 2720
  Extension packages........................................................ 2726
  The util package............................................................... 2726
  The os package.............................................................. 2792
  The com package............................................................ 2811
  The xml package............................................................. 2931
  The security package...................................................... 3127
  File extensions............................................................... 3153
  Genero BDL errors.......................................................... 3154

Web services........................................................................ 3270
<table>
<thead>
<tr>
<th>Contents</th>
</tr>
</thead>
</table>

General ................................................................................................................................................ 3270
Introduction to Web services.................................................................................................................. 3270
GWS demos and examples............................................................................................................................ 3275
Debugging .................................................................................................................................................. 3276
Platform-specific notes .............................................................................................................................. 3276
Known issues .............................................................................................................................................. 3277
Legal Notices .............................................................................................................................................. 3277

Concepts.................................................................................................................................................... 3277
Genero Web Services APIs.......................................................................................................................... 3277
SOAP features............................................................................................................................................ 3280
Stateful SOAP Web services....................................................................................................................... 3283
Encryption, BASE64 and password agent with fglpass tool ................................................................. 3290
HTTP compression in SOAP...................................................................................................................... 3295
SOAP multipart style requests in GWS .............................................................................................. 3297
Message Transmission Optimization Mechanism (MTOM) ................................................................ 3297
swaRef (SOAP with attachments using wsi:swaRef) ........................................................................... 3298

Security ..................................................................................................................................................... 3299
Encryption and authentication .................................................................................................................. 3299
Accessing secured services ...................................................................................................................... 3303
HTTPS configuration ............................................................................................................................... 3305
Configure a WS client to access an HTTPS server ............................................................................. 3305
Enable OCSP .......................................................................................................................................... 3310
Troubleshoot certificate issues .............................................................................................................. 3310
Certificates in practice ............................................................................................................................... 3313
Examining certificates .............................................................................................................................. 3316
Troubleshoot common issues .................................................................................................................. 3318
The Diffie-Hellman key agreement algorithm .......................................................................................... 3318

SOAP Web Services.................................................................................................................................. 3321
Writing a Web Services client application ............................................................................................ 3321
Writing a Web Services server application ........................................................................................... 3342
How Do I ... ? ....................................................................................................................................... 3321

REST Web services................................................................................................................................... 3374
Designing REST Web services .............................................................................................................. 3374
RESTful Web services with high-level framework .............................................................................. 3375
RESTful Web services with low-level APIs ............................................................................................ 3418

How Do I ... ? ....................................................................................................................................... 3445
Call Java APIs from Genero in a SOA environment ............................................................................... 3445
Call .NET APIs from Genero in a SOA environment ........................................................................... 3450
Compute a hash value from a BDL string ............................................................................................... 3457

Deploy a Web Service................................................................................................................................ 3459
Web services server program deployment ............................................................................................ 3459
Configuring the Apache Web server for HTTPS .................................................................................... 3460

Reference .................................................................................................................................................. 3463
High-level RESTful Web service attributes ............................................................................................. 3463
Web Services FQLPROFILE configuration .............................................................................................. 3489
XML to BDL serialization options ........................................................................................................... 3499
WSHelper library ..................................................................................................................................... 3499
OAuthAPI library ....................................................................................................................................... 3504
Attributes to customize XML serialization ............................................................................................. 3523
Error handling in GWS calls (STATUS) ..................................................................................................... 3554
Interruption handling in GWS calls (INT_FLAG) ...................................................................................... 3555
Server API functions - version 1.3 only ................................................................................................. 3555
Configuration API functions - version 1.3 only ...................................................................................... 3561
Using fglwsdl to generate code from WSDL or XSD schemas .......................................................... 3564
## Genero Business Development Language User Guide

Manual organization at a glance.

<table>
<thead>
<tr>
<th>General on page 25</th>
<th>Installation on page 37</th>
<th>Upgrading on page 48</th>
<th>Configuration on page 255</th>
</tr>
</thead>
<tbody>
<tr>
<td>• What’s New</td>
<td>• Resources for upgrades on page 37</td>
<td>• New features of Genero BDL on page 48</td>
<td>• The FGLPROFILE file(s) on page 255</td>
</tr>
<tr>
<td>Documentation</td>
<td>• Software requirements on page 37</td>
<td>• Upgrade Guides for Genero BDL on page 109</td>
<td>• Environment variables on page 262</td>
</tr>
<tr>
<td>conventions on page 25</td>
<td>• Installing Genero BDL on page 39</td>
<td>• Planned desupport on page 211</td>
<td>• Front-end connection on page 285</td>
</tr>
<tr>
<td>General terms used in this documentation on page 27</td>
<td>• Upgrading Genero BDL on page 40</td>
<td>• Migrating from IBM Informix 4GL to Genero BDL on page 211</td>
<td>• Database server connections on page 285</td>
</tr>
<tr>
<td>Introduction to Genero BDL programming on page 28</td>
<td>• Platform specific notes on page 43</td>
<td>• Migrating from Four Js BDS to Genero BDL on page 231</td>
<td></td>
</tr>
<tr>
<td>Language basics on page 285</td>
<td></td>
<td>• Frequently Asked Questions on page 250</td>
<td></td>
</tr>
<tr>
<td>• Syntax features on page 285</td>
<td>• Program modules on page 462</td>
<td>• SQL programming on page 590</td>
<td>• IBM Informix</td>
</tr>
<tr>
<td>• Primitive Data types on page 289</td>
<td>• Database schema on page 467</td>
<td>• Database connections on page 656</td>
<td>• IBM DB2 LUW</td>
</tr>
<tr>
<td>• Type conversions on page 311</td>
<td>• Importing modules on page 480</td>
<td>• Database transactions on page 688</td>
<td>• IBM Netezza</td>
</tr>
<tr>
<td>• Formatting data on page 318</td>
<td>• Program execution on page 488</td>
<td>• Static SQL statements on page 694</td>
<td>• Microsoft SQL Server</td>
</tr>
<tr>
<td>• Literals on page 325</td>
<td>• Runtime stack on page 493</td>
<td>• Dynamic SQL management on page 709</td>
<td>• Oracle MySQL</td>
</tr>
<tr>
<td>• Expressions on page 330</td>
<td>• Exceptions on page 502</td>
<td>• Result set processing on page 715</td>
<td>• Oracle Database</td>
</tr>
<tr>
<td>• Operators on page 336</td>
<td>• Localization on page 512</td>
<td>• Positioned updates/deletes on page 726</td>
<td>• PostgreSQL</td>
</tr>
<tr>
<td>• Flow control on page 377</td>
<td>• Globals on page 551</td>
<td>• SQL insert cursors on page 730</td>
<td>• SQLite</td>
</tr>
<tr>
<td>• Variables on page 390</td>
<td>• Predefined constants on page 555</td>
<td>• SQL LOAD and UNLOAD on page 736</td>
<td>• SAP ASE</td>
</tr>
<tr>
<td>• Constants on page 405</td>
<td>• Configuration options on page 557</td>
<td></td>
<td>• SAP HANA on page 1069</td>
</tr>
<tr>
<td>• Records on page 408</td>
<td>• Program registers on page 567</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Arrays on page 418</td>
<td>• OOP support on page 570</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Dictionaries on page 426</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Types on page 431</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>User Interface: User interface basics on page 1105</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• The dynamic user interface on page 1105</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• The abstract user interface tree on page 1106</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Genero user interface modes on page 1109</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• GUI front-end connection on page 1116</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Using a text terminal on page 1123</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>User Interface: Form definitions on page 1131</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Windows and forms on page 1131</td>
</tr>
<tr>
<td>• Using images on page 1147</td>
</tr>
<tr>
<td>• Accessibility guidelines on page 1157</td>
</tr>
<tr>
<td>• Message files on page 1160</td>
</tr>
<tr>
<td>• Action defaults files on page 1162</td>
</tr>
<tr>
<td>• Presentation styles on page 1165</td>
</tr>
<tr>
<td>• Form specification files on page 1237</td>
</tr>
<tr>
<td>• Form rendering on page 1415</td>
</tr>
<tr>
<td>• Toolbars on page 1436</td>
</tr>
<tr>
<td>• Topmenus on page 1444</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>User Interface: Dialog instructions on page 1452</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Static display (DISPLAY/ERROR/MESSAGE/CLEAR) on page 1452</td>
</tr>
<tr>
<td>• Prompt for values (PROMPT) on page 1462</td>
</tr>
<tr>
<td>• Ring menus (MENU) on page 1467</td>
</tr>
<tr>
<td>• Record input (INPUT) on page 1482</td>
</tr>
<tr>
<td>• Record list (DISPLAY ARRAY) on page 1501</td>
</tr>
<tr>
<td>• Editable record list (INPUT ARRAY) on page 1531</td>
</tr>
<tr>
<td>• Query by example (CONSTRUCT) on page 1565</td>
</tr>
<tr>
<td>• Multiple dialogs (DIALOG - inside functions) on page 1585</td>
</tr>
<tr>
<td>• Parallel dialogs (START DIALOG) on page 1700</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>User Interface: User interface programming on page 1706</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Dialog programming basics on page 1706</td>
</tr>
<tr>
<td>• Input fields on page 1714</td>
</tr>
<tr>
<td>• Dialog actions on page 1739</td>
</tr>
<tr>
<td>• List dialogs on page 1831</td>
</tr>
<tr>
<td>• Table views on page 1853</td>
</tr>
<tr>
<td>• Scrollgrid views on page 1880</td>
</tr>
<tr>
<td>• Tree views on page 1889</td>
</tr>
<tr>
<td>• Split views on page 1900</td>
</tr>
<tr>
<td>• Drag &amp; drop on page 1917</td>
</tr>
<tr>
<td>• Front calls on page 583</td>
</tr>
<tr>
<td>• Web components on page 1922</td>
</tr>
<tr>
<td>• Dynamic Dialogs on page 2004</td>
</tr>
<tr>
<td>• Canvases on page 2013</td>
</tr>
<tr>
<td>• Start menus on page 2019</td>
</tr>
<tr>
<td>• Window containers (WCI) on page 2023</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reports on page 2026</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Understanding reports on page 2026</td>
</tr>
<tr>
<td>• XML output for reports on page 2027</td>
</tr>
<tr>
<td>• The report driver on page 2033</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Programming tools on page 2065</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Command reference on page 2065</td>
</tr>
<tr>
<td>• Compiling source files on page 2106</td>
</tr>
<tr>
<td>• Source code edition on page 2123</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Extending the language on page 2192</th>
</tr>
</thead>
<tbody>
<tr>
<td>• The Java interface on page 2192</td>
</tr>
<tr>
<td>• C-Extensions on page 2230</td>
</tr>
<tr>
<td>• User-defined front calls on page 2249</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Library reference on page 2268</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Built-in functions on page 2268</td>
</tr>
<tr>
<td>• Utility modules on page 2301</td>
</tr>
<tr>
<td>• Built-in packages on page 2378</td>
</tr>
</tbody>
</table>
BDL 3.20 new features

Features added in 3.20 releases of the Genero Business Development Language.

Important: This page covers only those new features introduced with the Genero BDL version specified in the page title. Check prior new features pages if you migrate from an earlier version. Make sure to also read the upgrade guide corresponding to this Genero version.

Table 1: Core language

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support for circular dependency of modules with IMPORT FGL.</td>
<td>See Circular dependency with IMPORT FGL on page 126.</td>
</tr>
<tr>
<td>Method declaration, to define functions acting on a user-defined type.</td>
<td>See Methods on page 452, FUNCTION syntax.</td>
</tr>
<tr>
<td>INTERFACE structure, to define a group of methods for a type.</td>
<td>See Interfaces on page 457.</td>
</tr>
<tr>
<td>Passing records by reference (INOUT).</td>
<td>See Passing records by reference with INOUT on page 495.</td>
</tr>
<tr>
<td>Variable initializers: The DEFINE instruction supports a clause to initialize the variable.</td>
<td>See Variable initializers on page 396, DEFINE on page 390.</td>
</tr>
<tr>
<td>Named parameters in function calls: CALL func(pl:value).</td>
<td>See Naming parameters in a function call on page 443.</td>
</tr>
<tr>
<td>Function attributes (for RESTful Web Services). A FUNCTION definition can specify function attributes, parameter attributes and return values attributes.</td>
<td>See Function attributes on page 455, FUNCTION definitions on page 437.</td>
</tr>
<tr>
<td>New STRING and base.StringBuffer methods to remove all kinds of whitespace characters in a string.</td>
<td>See the *WhiteSpace methods in STRING data type methods on page 2382 and base.StringBuffer methods on page 2448.</td>
</tr>
<tr>
<td>FUNCTION definition syntax allows the RETURNS () clause, to specify an empty return list and enforce compilation verifications to avoid RETURN instruction misusage.</td>
<td>See FUNCTION syntax, Returning values on page 446.</td>
</tr>
<tr>
<td>Arrays can be assigned without the .* notation.</td>
<td>See Dynamic array assignment with .* notation on page 125.</td>
</tr>
<tr>
<td>Records can be assigned without the .* notation.</td>
<td>See Record copy with .* notation on page 125.</td>
</tr>
<tr>
<td>FGLPROFILE fglrun.floatToCharScale2 and fglrun.floatToCharScale2.print to control FLOAT/SMALLFLOAT to string conversion.</td>
<td>See FLOAT/SMALLFLOAT to string conversion on page 127.</td>
</tr>
<tr>
<td>GetOpt utility type and methods, to process command line arguments in the getopt style (--option=value).</td>
<td>See getopt: Command line options module on page 2365.</td>
</tr>
<tr>
<td>New os.Path.glob() method to get a list of files from a pattern.</td>
<td>See os.Path.glob on page 2801.</td>
</tr>
<tr>
<td>TEXT and BYTE variables are now automatically located in memory by JSON parsing methods.</td>
<td>See JSON to BDL data conversion.</td>
</tr>
</tbody>
</table>

Table 2: User interface

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>The new gui.rendering FGLPROFILE entry, to enable Universal Rendering.</td>
<td>See Graphical mode with Universal Rendering on page 1112.</td>
</tr>
<tr>
<td>FGLGBCDIR environment variable, to define the directory of the GBC component to be used for Universal Rendering.</td>
<td>See FGLGBCDIR on page 275, Graphical mode with Universal Rendering on page 1112.</td>
</tr>
</tbody>
</table>
Overview

KEYBOARDHINT form attribute allows now the URL value to define an EDIT field for URL input.

GBC supports the "chrome" value for actionPanelPosition, ringMenuPosition and toolBarPosition window style attributes.

ui.Interface class methods to identify the Universal Rendering client and version.

Form elements of type GROUP can be defined as collapsible with new style attributes (collapsible / initiallyCollapsed).

Collapsible groups and folders can be configured to display the collapsor icon at a specific position with the collapsePosition style attribute.

ON FILL BUFFER trigger for display array dynamic dialog.

Built-in classes methods to control form elements are now case-insensitive.

Use the itemsAlignment style attribute to control the alignment of elements inside a scrollgrid.

New enableCalendar style attribute for DateTimeEdit with GDC.

New qtStyle style attribute for ComboBox with GDC.

The INITIALIZER attribute of COMBOBOX accepts module specification as function prefix (the module is then automatically loaded)

Setting the comment of a form element or form field.

New FGLPROFILE entry gui.programStoppedMessage to define a generic message to be displayed to the end user when a program stops because of a runtime error.

Upgrade notes for presentation styles.

Table 3: SQL databases:

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support for Oracle® database 18c and 19c with the dbmora_18 driver.</td>
<td>See Database drivers changes on page 117.</td>
</tr>
<tr>
<td>Using Oracle® 18c private temporary tables to emulate Informix® temporary tables.</td>
<td>See Using the private temporary table emulation on page 949.</td>
</tr>
<tr>
<td>Overview</td>
<td>Reference</td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
</tr>
<tr>
<td>Support for PostgreSQL 11 and 12 with the <code>dbmpgs_9</code> driver.</td>
<td>See <a href="#">Database drivers changes</a> on page 117, PostgreSQL 12 notes on page 124.</td>
</tr>
<tr>
<td>Support for SQL Server 2019 (v15) with the <code>dbmsnc_17</code> driver.</td>
<td>See <a href="#">Database drivers changes</a> on page 117.</td>
</tr>
<tr>
<td>New <code>dbmmys_5_6</code> driver for MySQL 5.6 as replacement of <code>dbmmys_5_5</code> no longer available because of MySQL 5.5 desupport.</td>
<td>See <a href="#">Database drivers changes</a> on page 117.</td>
</tr>
<tr>
<td>Utility function <code>db_get_last_serial()</code> to retrieve the last generated serial, mandatory for BIGINT incremented columns.</td>
<td>See <code>db_get_last_serial()</code> on page 2308.</td>
</tr>
<tr>
<td>FGPLPROFILE entry <code>dbi.database.dbname.pgs.schema</code> to define the schema search path for PostgreSQL.</td>
<td>See <a href="#">PostgreSQL specific FGPLPROFILE parameters</a> on page 676.</td>
</tr>
<tr>
<td>FGLSQLDEBUG output enhancement to identify when SQL INTERRUPT ON/OFF is used.</td>
<td>See note in Performances with SQL interruption on page 124.</td>
</tr>
<tr>
<td>Support for IBM® Informix® trusted connections, with the new <code>TRUSTED</code> keyword for CONNECT TO, and the new <code>SET SESSION AUTHORIZATION</code> statement.</td>
<td>See <a href="#">CONNECT TO</a> on page 683, SET SESSION AUTHORIZATION on page 685.</td>
</tr>
<tr>
<td>Informix SQL SELECT statement row limiting clause <code>SKIP n FIRST m</code> can be converted to native equivalent.</td>
<td>See <a href="#">Row limiting clause (SELECT)</a> on page 650.</td>
</tr>
<tr>
<td>With Microsoft SQL Server, in DDL statements such as CREATE TABLE, character data types can be converted to the equivalent native national character types, with FGPLPROFILE entry <code>dbi.database.dsnname.ifxemul.nationalchars=true</code>.</td>
<td>See <a href="#">character types with SQL Server, ifxemul FGPLPROFILE settings</a>.</td>
</tr>
<tr>
<td>To avoid Microsoft SQL Server 468 collation conflict errors with temporary tables, the ODI drivers for SQL Server add now a COLLATE DATABASE_DEFAULT clause after character types in the native statement generated from a CREATE TEMP TABLE. This is not needed for SELECT INTO TEMP.</td>
<td>See <a href="#">Temporary tables</a> on page 861.</td>
</tr>
<tr>
<td>Oracle® database rowid (in base 64 format) can be found in <code>SQLCA.SQLERRM</code>.</td>
<td>See <a href="#">ORACLE rowid in SQLCA.SQLERRM</a> on page 128.</td>
</tr>
<tr>
<td>Upgrade notes for database drivers.</td>
<td>See <a href="#">Database drivers changes</a> on page 117.</td>
</tr>
<tr>
<td>Upgrade notes for front calls.</td>
<td>See <a href="#">Front calls changes</a> on page 116.</td>
</tr>
<tr>
<td>Upgrade notes for web components.</td>
<td>See <a href="#">Web components changes</a> on page 117.</td>
</tr>
</tbody>
</table>
Table 4: Tools

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source code formatting tool: \texttt{fglcom} --format, including new command line tools \texttt{fglgitformat}, \texttt{fglformatdiff}.</td>
<td>See Source code beautifier on page 2174, \texttt{fglgitformat} on page 2093, \texttt{fglformatdiff} on page 2094.</td>
</tr>
<tr>
<td>Qualifying imported symbols: \texttt{fglcom} --qualify-imports.</td>
<td>See Qualifying imports.</td>
</tr>
<tr>
<td>Use lowercase keywords in VIM with \texttt{fgl_lowercase_keywords=1}.</td>
<td>See Configure VIM for Genero BDL on page 2123.</td>
</tr>
<tr>
<td>\texttt{fglrun/fgllink} supports \texttt{@argfile} to specify the list of files to link.</td>
<td>See Providing the files to link in an arguments file on page 2115, Providing the files to link in an arguments file on page 2114.</td>
</tr>
<tr>
<td>\texttt{fglcom} can now process several .4gl files on the command line.</td>
<td>See Compiling several .4gl sources in a single command on page 2108, \texttt{fglcom} on page 2071.</td>
</tr>
<tr>
<td>Commands accepting a list of files like \texttt{fglcom} or \texttt{fgllink}, can do pathname expansion (\texttt{fglcom} *.4gl), on Unix and Windows platforms.</td>
<td>See Compiling several .4gl sources in a single command on page 2108, \texttt{fglcom} on page 2071.</td>
</tr>
<tr>
<td>To reduce compilation time of large projects, \texttt{fglcom} supports the --make option, to compile the provided .4gl sources only when the .42m is not up-to-date. The --make option can be used with the --simulate option, to simulate the process without compiling.</td>
<td>See Compiling in make mode on page 2109.</td>
</tr>
<tr>
<td>The \texttt{fglcom} --dependencies option can be used to produce makefile-style dependency rules for imported modules.</td>
<td>See Producing make-style dependency rules on page 2111.</td>
</tr>
<tr>
<td>\texttt{fglcom} supports \texttt{@argfile} to specify the list of source files in an input file.</td>
<td>See Providing the source files in an arguments file on page 2111.</td>
</tr>
<tr>
<td>\texttt{fglcom} supports the value &quot;auto&quot; for \texttt{-p} preprocessor type option.</td>
<td>See Preprocessing style option on page 2133.</td>
</tr>
<tr>
<td>\texttt{fglrun} provides the --print-missing-imports option to show only missing IMPORT FGL (\texttt{--print-imports} option prints required imports even if IMPORT FGL is used)</td>
<td>See Identifying modules to be imported on page 483.</td>
</tr>
<tr>
<td>\texttt{fglcom} supports the new warning argument \texttt{-W unused-parameter}, that prints a message when a function parameter is not used. The negative form \texttt{-W no-unused-parameter} can be combined with \texttt{-W unused}, to ignore parameters that are unused.</td>
<td>See \texttt{fglcom} option \texttt{-W} option.</td>
</tr>
</tbody>
</table>
### Table 5: Web Services

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
</table>
| The Genero Web Service engine has been enhanced with a mechanism to provide a high level REST web service. You can implement REST Web services using function attributes (identified with WS* prefix) that you specify in ATTRIBUTES() clauses of functions. | See:  
- Function attributes on page 455  
- High-level RESTful Web service attributes on page 3463  
- RESTful Web services with high-level framework on page 3375 |
| The fglrestful tool takes an OpenAPI specification file (JSON format) and generates the client stub to interact with the REST service. Descriptions using the WSDescription attribute are allowed in input and output parameters. These descriptions can be generated in the REST stub file using the fglrestful tool's --comment option. | See fglrestful on page 2085. |
| com.WebServiceEngine option server_restdefaultformat added to define runtime support for MIME types in REST operations. Method added to publish high-level RESTful Web services in the GWS engine. Method added to provide error management in the high-level RESTful Web service. Empty HTTP POST and PUT requests. The FGLPROFILE entry http.global.request.date can be set to true, in order to force sending HTTP Date header for GET, HEAD and DELETE requests. The com.HTTPRequest.setAuthentication has an enhancement to support the NTLM authentication protocol of Windows® server that requires the connection to be kept open. API added to provide management of OAuth authentication for RESTful Web service secured by OpenID Connect. Upgrade notes for web services. | See WebServiceEngine options on page 2840.  
See Support for empty HTTP POST or PUT requests on page 110.  
See Control HTTP Date header for GET, HEAD and DELETE requests on page 110.  
See com.HTTPRequest.setAuthentication on page 2883 or com.HTTPRequest.setKeepConnection on page 2887.  
See OAuthAPI library on page 3504.  
See Web Services changes on page 110. |

### Table 6: Mobile apps (GMA 1.40 and GMI 1.40)

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
</table>
| gmabuildtool option -bgr / --build-gbc-runtime, to define the GBC to be used to build the app. gmibuildtool option --gbc, to define the GBC to be used to build the app. | See Building Android apps with Genero on page 3591.  
See Building iOS apps with Genero on page 3606. |
GMA supports now the alignment style attribute for BUTTON.

Cordova plugin wrapper libraries now available on the FOURJS Cordova Github.

New Cordova plugin front calls to get plugin information.

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>GMA supports now the alignment style attribute for BUTTON.</td>
<td>See Button.alignment style attribute (GMA) on page 116, Button style attributes on page 1185.</td>
</tr>
<tr>
<td>Cordova plugin wrapper libraries now available on the FOURJS Cordova Github.</td>
<td>See Cordova plugins for GMA, Cordova plugins for GMI.</td>
</tr>
<tr>
<td>New Cordova plugin front calls to get plugin information.</td>
<td>See New Cordova front calls in GMA, New Cordova front calls in GMI.</td>
</tr>
</tbody>
</table>

Table 7: Miscellaneous

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>New parameter for monitor.update front call to control administrator permission elevation prompt.</td>
<td>See Front calls changes on page 116, monitor.update front call.</td>
</tr>
</tbody>
</table>

General

These topics provide an introduction to the Genero Business Development Language

Documentation conventions

Learn about documentation conventions regarding syntax, warnings and code examples.

Syntax diagrams

A syntax diagram describes the context-free, abstract grammar of a product function.

A syntax diagram for example describes a language instruction, the structure of a configuration file, or the options of command-line tool.

The following rules are used in a syntax diagram:

- Invariable syntax elements (keywords) are written in fixed font.
- Language keywords are in uppercase, like INPUT BY NAME.
- Variable syntax elements are written in italics.
- Wildcard characters are underlined and used to indicate syntax elements that can either repeat, be mandatory or optional.

Table 8: Wildcard characters

<table>
<thead>
<tr>
<th>Wildcard characters notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ element ]</td>
<td>Square brackets indicate an optional element in the syntax.</td>
</tr>
<tr>
<td>[ element-1</td>
<td>element-2 ... ]</td>
</tr>
<tr>
<td>{ element-1</td>
<td>element-2 ... }</td>
</tr>
</tbody>
</table>
### Wildcard characters notation

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A sign made of square brackets with three dots indicate that the previous element can appear more than once.</td>
</tr>
<tr>
<td>A sign made of square brackets with a comma followed by three dots indicate that the previous element can appear more than once, and must be separated by a comma.</td>
</tr>
</tbody>
</table>

The following syntax diagram defines the `CALL` instruction of Genero BDL:

```
CALL function-name ( parameter-name [, ...] )
[ RETURNS variable-name [, ...] ]
```

### Warnings, notes and tips

**Documentation notes focus on a technical fact you must be aware of.**

**Important**

An important note describes a fact that needs to be considered, before using the related feature.

**Important:** When a DATE, DATETIME or INTERVAL constant cannot be initialized correctly, it is set to NULL.

Some Genero features are not supported on all back-end or front-end platforms. The following warnings about the limitation:

**Important:** This feature is not supported on mobile platforms.

**Important:** This feature is not supported on mobile platforms (in native rendering mode).

**Important:** This feature is only for mobile platforms.

**Important:** This feature is only for the GMA/Android™ platform.

**Important:** This feature is only for the GMI/iOS platform.

**Important:** This feature is experimental, the syntax/name and semantics/behavior may change in a future version.

**Important:** This feature is provided for development and debug purpose, and must not be used in a production environment.

Some Genero features are deprecated:

**Important:** This feature is deprecated, its use is discouraged although not prohibited.

**Notes**

Simple notes describe a fact you must be aware of, but does not represent any risk.

**Note:** This feature has been introduced in version 2.50.

**Tips**

Tip notes suggest an option or programming pattern to be used.

**Tip:** Use dynamic arrays instead of static arrays to save memory usage.

**Warnings**

Warning notes highlight an action that might cause something to break or otherwise be catastrophic.
Warning: Using a high cost value for the salt is very CPU consuming, and can really slow down the application depending on the system it is running.

Code examples

Code examples contain code that can be copied as-is.

Code examples appear in the documentation as follows:

```
MAIN
  DEFINE a1 ARRAY[100] OF INTEGER,
      a2 ARRAY[10,20] OF RECORD
        id INTEGER,
      ...
```

General terms used in this documentation

This documentation uses general terms that must be clarified for a good understanding.

**Application**

The *application* defines all software components that compose the information system managing a given domain. Usually, the domains covered by programs written in BDL are business oriented.

**Application data**

*Application data* defines the data manipulated by the application. It is typically managed by one or more database systems. The application data has a volatile state when loaded in the runtime system, and it has a static state when stored in the database system.

**Database**

The *database* is a logical entity regrouping the application data. It is managed by the database system.

**Database system**

The *database system* is the software that manages data storage and searching; it is usually installed on the database server machine and is supported by a tier software vendor.

**Developer**

The *developer* is the person in charge of the conception and implementation of the application components.

**Deprecated feature**

A *deprecated feature* is a feature, design, or practice whose use is discouraged although not prohibited. Typically, a deprecated feature has been superseded or is no longer considered safe, but it is not yet removed from the system. Four Js provides support for deprecated features. Bugs will be fixed but enhancements will not be made.

**Desupported feature**

A *desupported feature* is a feature, design, or practice that is no longer supported. A desupported feature may still exist, but bugs will not be fixed. The code supporting the feature may be removed without notice.

**End user**

The *end user* is the person that uses the application; that person works on hardware called the workstation.

**Experimental feature**

An *experimental feature* is a new feature, design, or practice that is provided in a production software package, but that should only be used for testing in the
development environment, as it can be subject of changes in a next version.

Front-end
The front-end is the software that manages the display and input of the user interface on the workstation machine. This component is historically called "the client". It is the software handling the presentation. There are different sort of front-ends available, for desktop workstations (GDC), for web-browsers (GBC/GAS), and on mobile devices (GMA/GMI).

Programs
The programs are the software components that are developed and distributed by the supplier of the application. Programs typically implement business logic. Programs are executed by the runtime system. Program components are typically p-code modules, forms and additional resource files.

Runtime system
The runtime system is the software that manages the execution of the programs, where the business logic is processed. The runtime system is also known as the Dynamic Virtual Machine (DVM - fglrun).

User interface
The user interface defines the parts of the programs that interact with the end user, including interactive elements like windows, screens, input fields, buttons and menus. It is managed by the front-end.

Workstation
The workstation identifies the hardware used by the end user to interact with the front-end. It can be an dumb terminal, a computer, or mobile device, as long as a front-end is available on the hardware.

Other source of information
FourJs Development Tools provides other channels related to Genero BDL

Other source of information
• Licensing, supported platforms, bug database, discussion forum, support request on the 4JS support web site
• Code examples in the $FGLDIR/demo directory.
• Tutorials and samples shipped with Genero Studio.
• Free samples and tools on the FourJs Genero Github

Introduction to Genero BDL programming
Understand the basics about programming, compiling and deploying an application.

Overview of Genero BDL
Genero Business Development Language (BDL) is a program language designed to write an interactive database application.

A Genero BDL application is a set of programs that handle the interaction between a user and a database. Programs communicate with the database server with Structured Query Language (SQL), and execute interactive instruction controlling application forms, to manage user input.
Figure 1: Interactive database applications with Genero

An important feature of the language is the ease with which you can design applications that allow the user to access and modify data in a database. The language syntax includes a set of SQL statements to manipulate the database, powerful interactive instructions that provide simple record input, read-only and read-write record list handling, as well as database query to search the database, by using forms supporting a large variety of graphical widgets.

The program sources are compiled to p-code modules, which can be interpreted on different platforms by the Dynamic Virtual Machine (the Runtime system).

Separation of business logic and user interface
Genero BDL separates business logic and the user interface to provide maximum flexibility.

- Intensive use of XML standards ensures that user interface is well separated from the program logic.
- Forms define the user interface are designed in a simple-to-understand and simple-to-read syntax.
- The business logic is written in .4gl source code modules.
- High-level interactive instructions called dialogs let you write form controllers in a few lines of code.
- Action views (buttons, menu items, toolbar icons) in the form definition can trigger actions defined in the business logic.
- The user interface can be manipulated at runtime, for example to enable/disable fields and action views dynamically.

Related concepts
User interface on page 1105
These topics cover programming the user interface (UI) with the Genero Business Development Language.

Portability - write once, deploy anywhere
Genero application can be deployed for different kinds of display devices, operating systems and database servers, by using the same source code.

Application forms can be displayed with a graphical front-end device based on native desktop frameworks, in web browsers, as well as on simple dumb terminals.

Genero programs can be executed on major Operating Systems such as UNIX™, Linux®, Windows® and Mac OS X®.
SQL can be performed by IBM® Informix®, or any other supported database server such as Oracle® DB, IBM® DB2®, Microsoft® SQL Server, PostgreSQL, Oracle® MySQL, MariaDB, SAP HANA, SQLite (for mobile).
Figure 2: Genero portability

**Genero BDL concepts**

This section describes basic Genero language concepts.

**Genero programs**

Genero Business Development Language (BDL) is a programming language based on simple and readable syntax.

The program logic is written in text files with the `.4g1` file extension, called *program source modules*. Module sources are compiled (fglcomp) into p-code modules with the `.42m` file extension, that can be executed by the *runtime system* (fglrun). Application programs are built with a group of `.42m` modules.

**Related concepts**

[Program modules](#) on page 462
Explains program structure basics.

**Integrated SQL support**
A set of SQL statements are part of the language syntax and can be used directly in the source code, as a normal procedural instruction.

The static SQL statements are parsed and validated at compile time. At runtime, these SQL statements are automatically prepared and executed. Program variables are detected by the compiler and handled as SQL parameters. Common SQL statements such as SELECT, INSERT, UPDATE or DELETE can be directly written in the source code, as part of the language syntax:

```plaintext
MAIN
  DEFINE n INTEGER, s CHAR(20)
  DATABASE stores
  LET s = "Sansino"
  SELECT COUNT(*) INTO n FROM customer WHERE custname = s
  DISPLAY "Rows found: " || n
END MAIN
```

Dynamic SQL management allows you to execute SQL statements that are constructed at runtime. The SQL statement can use SQL parameters:

```plaintext
MAIN
  DEFINE txt CHAR(20)
  DATABASE stores
  LET txt = "SET DATE_FORMAT = YMD"
  PREPARE sh FROM txt
  EXECUTE sh
END MAIN
```

Through the database drivers, the same program can open database connections to any of the supported databases.

**Related concepts**

SQL support on page 590
These topics cover SQL support in the Genero Business Development Language.

**XML support**
The language provides XML support through different classes, depending on your needs.

Genero XML support is provided in two forms:

- For basic XML tasks related to the user interface, use the built-in XML classes.
- For complex XML tasks, and Web Services functions, use the full-featured XML classes provided in the web services extension.

**The user interface**
The Genero user interface technology is based on the sharing of an abstract representation between the runtime system and the front-end.

When a program starts, the runtime system creates the abstract user interface (AUI) tree and passes this tree to the front-end. The front-end renders the abstract element as real graphical objects on the workstation.

When an interaction statement takes control of the application, the tree on the front-end is automatically synchronized with the runtime system tree. Runtime system and front-ends communicate with the front end protocol, through the computer network. The AUI tree and the protocol are using XML standards.
Figure 3: AUI tree synchronization

Resource files describe the appearance (decoration) of some of the graphic objects. Default resource files (default.4ad, default.4st) are provided and can be customized, or replaced with your own versions.

The elements of the AUI tree can be manipulated at runtime with built-in utilities.

Related concepts
User interface on page 1105
These topics cover programming the user interface (UI) with the Genero Business Development Language.

Language library
Several utility packages are provided to ease programming in different domains.

Utility functions and classes are available in different forms, including built-in classes, built-in functions, and loadable modules.

Related concepts
Library reference on page 2268
Reference for classes and functions provided as built-in or extension packages.

Windows and forms
Programs manipulate window and form objects to define display areas controlled by interactive statements.

The forms are defined in text-based form specification files (.per). These form files are transformed by the fglform compiler to produce the runtime form files (.42f) that are deployed in production environments. The resulting (.42f) files are XML documents that describe the form elements, enabling portability across display devices. The XML file can also be written directly, or it can be generated or modified from your program at runtime with XML utilities.

Related concepts
Form definitions on page 1131
This section describes how to define application forms and program resources related to the presentation layer.

Interactive instructions
Control application forms with interactive instructions that perform field input and action handling.

These interactive instructions allow the program to respond to user actions and data input. For example the INPUT BY NAME block controls a set of form fields where the user can enter data:

```genero
DEFINE cust_rec RECORD LIKE customer.*
INPUT BY NAME cust_rec.*
```
Interactive instructions can be implemented as modal or parallel dialogs. Modal dialogs control a given window, and that window closes when the dialog is accepted or canceled. The window displays on top of any existing windows which are not accessible while the modal dialog executes. Parallel dialogs allow access to several windows simultaneously; the user can switch from one window to the other. Parallel dialogs are mainly used to implement split views on mobile platforms.

**Related concepts**

Dialog instructions on page 1452
This section describes the dialog instructions to control application forms and the concepts related to dialog implementation.

**Responding to user actions**

Clicking a form button or pressing a key triggers actions that can invoke the execution of program of code called action handlers. Form elements that can trigger actions are called action views.

Action handlers are defined in interactive statements with the ON ACTION clause. The code defined in action handler blocks is executed when an action is fired. Action objects are created and linked to action views when such ON ACTION handlers are seen by the runtime system. Common action handlers, such as accept (dialog validation) and cancel (dialog cancellation), are created automatically in accordance with the interactive instruction.

By configuring action defaults, you define the default decoration attributes (text, image) and functional attributes (accelerator keys, context menu display) for the action views associated with actions.

**Related concepts**

Dialog actions on page 1739
Describes how to program action handling when the end user triggers an action on the front-end.

**Producing reports**

The language allows you to implement reports easily, producing different sort of output formats.

Page headers and footers, with page numbers, can be defined. Data can be grouped together, with group totals and subtotals shown. The output from a report can be sent to the screen, to a printer, to a file, or (through a pipe) to another program, and report output can even be redirected to an SAX filter in order to write XML data, that can be transformed into HTML, PDF or any other document format that can be generated from an XML source.

**Related concepts**

Reports on page 2026

**Internationalization**

The language supports single-byte and multibyte internationalization.

The language supports single-byte such as ISO-8859-1, as well as multibyte character sets such as BIG5 or UTF-8.

Length semantics to define variables and manipulate character string data can be based on byte or character units.

Labels and messages can be separated from programs and forms, to customize your application for specific subsets of the user population, whether it is for a particular language or a particular business segment.

The source files (4gl, per, 4ad, and so on) can be written in a specific encoding. However, we recommend you to keep sources in ASCII, and store locale-dependent strings in external strings files (str).

**Related concepts**

Localization on page 512
Localization support allows you to implement programs that follow specific language and cultural rules.

Localized strings on page 538
Localized strings provide a means of writing applications in which the text of strings can be customized on site.

**Web services support**
Implement Web services servers and clients with the Genero Web Services (GWS) library.

Web services are a standard way of communicating between applications over an intranet or Internet.

The Genero Web Services Extension (GWS) is an extension to the Genero Business Development Language. It installs within the Genero Business Development Language directory. The fglgws package includes both Genero Business Development Language and Genero Web Services.

The Genero Application Server is required to manage your Web services in a deployment environment. It is not required for Web services development, unless you are interested in testing deployment issues.

**Related concepts**
- [Web services](#) on page 3270
- Create a Web service client or server with Genero BDL.

**Extending the language**
You can extend the language using C or Java.

**Using C**
The language can be completed with *C extensions*. This allows you to implement specific function libraries in C, callable from the program modules. C extension libraries are typically used to interface with specific devices, such as barcode scanners or biometric identification devices.

**Using Java**
You can instantiate Java objects from your programs by using the *Java interface*. This allows you to take benefit of the huge class library of Java.

**Related concepts**
- [C-Extensions](#) on page 2230
- With *C-Extensions*, you can bind your own C libraries in the runtime system, to call C function from the application code.

**The Java interface** on page 2192
The *Java interface* allows you to import Java classes and instantiate Java objects in your programs.

**Programming tools**
Genero BDL includes several programming tools in addition to compilers.

A set of useful programming tools is provided, to help you in the application development process, for debugging, optimization and source documentation production.

**Related concepts**
- [Programming tools](#) on page 2065
- These topics cover programming with the Genero Business Development Language.

**Compiling a program**
You need to compile the source files in order to run the application.

A program can consist of a single source code module, but generally it will be organized in multiple modules, will involve form specification files and perhaps localized string files.

Database schema files are required when you define program data types and variables in terms of an existing database column or table, by using the `DEFINE ... LIKE` statement.

Before running your application with the runtime system, you need to use compilation tools in order to build the various runtime files.
Figure 4: Genero compilation tools

The compiled source code modules can be linked into a .42r program that can be executed by the Runtime System. Compiled modules can also be grouped together into a .42x library that can then be used to build .42r programs.

Figure 5: Linking of compiled modules

It is also possible to declare what modules are needed by the current module with the IMPORT FGL instruction, in order to define the dependency between .4gl modules. When using this language feature, it is no longer required to link modules together to build a program.

```
IMPORT FGL cust_module
MAIN
  DATABASE stores
  CALL cust_module.input_customer()
END MAIN
```

Importing modules is the preferred solution.
**Related concepts**

- **Program modules** on page 462
  Explains program structure basics.

- **Form definitions** on page 1131
  This section describes how to define application forms and program resources related to the presentation layer.

- **Localization** on page 512
  Localization support allows you to implement programs that follow specific language and cultural rules.

**Deploying application program files**

To deploy an application, you must deploy all of the required runtime and resource files. Many (but not all) of these files are compiled from the source files.

**Figure 6: Deployment files**

These program files must be deployed at the user site:

- `.42r` - Programs files
- `.42m` - PCode modules
- `.42f` - Runtime form files
- `.42s` - Compiled localized string files
- `.4ad`, `.4st`, etc - XML resource files

**Runtime environment settings**

The `fglprofile` configuration file and environment variables can be used to change the behavior of programs.

**Related concepts**

- **Program execution** on page 488
  This section describes program execution and language instructions related to program execution.

- **Configuration** on page 255
  These topics cover configuration options of the Genero Business Development Language.

- **Packaging web applications** on page 2183
Describes methods of packaging the runtime files and resources of your web applications and services using the \texttt{fglgar} tool.

## Installation

This chapter contains installation and setup instructions.

### Related concepts

**Upgrading** on page 48

These topics talk about what steps you need to take to upgrade to the next release of Genero Business Development Language, and allows you to identify which features were added for a specific version.

### Resources for upgrades

Version-specific upgrade guides describe potential compatibility issues with new product releases.

Product improvements and upgrade notes can be found in New features of Genero BDL on page 48 and Upgrade Guides for Genero BDL on page 109.

To get the detailed list of corrected defects in the new Genero BDL version, contact your support channel, or visit the Four Js Web site, where you can query the bug tracking database.

## Software requirements

Before installing, ensure that your system meets the minimum system requirements and additional software.

### Supported operating systems

Details of the supported operating systems for the Genero Business Development Language.

Genero Business Development Language is supported on a large range of operating systems, such as Linux\textsuperscript{®}, IBM\textsuperscript{®} AIX\textsuperscript{®}, HP-UX, SUN Solaris, Mac OS X\textsuperscript{®} and Microsoft\textsuperscript{™} Windows\textsuperscript{®}.

You must install the software package corresponding to the operating system that you use. For the detailed list of supported operating systems, refer to the relevant installation guide or contact your support center.

For a detailed list of supported operating systems, refer to the Supported platforms and databases document (available on the Products download page of the Four Js Web site) or contact your support center. This document also informs you which operating systems will no longer be supported as of the next release.

### System packages

Some Genero BDL features need specific operating system packages.

**Curses (or NCurses) library**

On Unix-like platforms, the Genero runtime system (\texttt{fglrun}) needs the Curses library to be installed on the system.

The Curses library is required for the text mode (for \textsc{INFORMIXTERM}=terminfo). However, \texttt{fglrun} is linked directly to the Curses library so it needs to be installed on the system even when the GUI mode is used.

Depending on the operating system, \texttt{fglrun} will require \texttt{libcurses}, \texttt{libnrcurses}, \texttt{libncurses5} or \texttt{libncursesw5} to be installed on the system.

**Note:** For UTF-8 support in text mode, you need the wide-char Curses shared library (\texttt{libncursesw.so.5} for example). Make sure that the Curses software package installed on your system includes this library. See Platform specific notes on page 43.

If the required Curses library is not installed, executing \texttt{fglrun} will report a shared library not found error.
Related concepts
TERMINFO terminal capabilities on page 1123

Database client software
To connect to a database server, the database client software must be installed on the system where you run the Genero BDL programs.

The Genero runtime system uses database drivers to connect to database servers, as a database client program. Database vendor-specific client software needs to be installed on the system where you run the Genero programs.

Genero database drivers are shipped as shared libraries and require the database vendor client software shared library such as Informix® Client SDK (with ESQL/C), Oracle® Client (with OCI), Microsoft® SQL Server ODBC. The database driver to be selected must correspond to the database client type and version.

For a detailed list of supported databases, database driver names, and operating systems on which those database drivers are supported, refer to the System Support matrix (available on the Products download page of the Four Js web site) or contact your support center. This matrix also informs you which database drivers will no longer be supported as of the next release.

Related information
Table 194: Database driver names on page 663

C compiler for C extensions
Ensure you have a C compiler and linker to compile your C-Extensions.

Applications using C extensions, need a C compiler and linker to build the C extension library that will be loaded by the runtime system.

C compiler On UNIX™ platforms
On UNIX® platforms, you need a cc compiler on the system where you create the C extension libraries. Note that some UNIX® systems do not have a C compiler installed by default.

C compiler On Microsoft™ Windows® platforms
On Windows® platforms, it is mandatory to install Microsoft® Visual C++ version corresponding to the installed Genero BDL package. The OS identifier of the Genero BDL package file name identifies the Visual C++ version to be used.

C compiler On Mac OS X™ platforms
On Mac® OS X platforms, it is mandatory to install Xcode® 6.1, on the system where you create the C extension libraries.

Related concepts
C-Extensions on page 2230
With C-Extensions, you can bind your own C libraries in the runtime system, to call C function from the application code.

Java runtime environment for FGL
Software requirements when using the Java Interface

Important: This topic is about JDK requirements for the FGL runtime system, to compile and execute server applications. To build GMA mobile applications, a different Java version may be required by Android™ tools. For more details, see Install Genero Mobile for Android on page 40.

In order to use the Java Interface in your application programs, you need the Java software installed and properly configured.

• Install a Java Development Kit on development sites (if you need to compile your own Java classes)
• Install a Java Runtime Environment on production sites (on the server where your programs are running)

**Note:** Over time, the Java platform has used different versioning systems, to distinguish a "Java developer version", from a "Java product version". For example, the Java developer version (JDK) 1.8 corresponds to the Java product version 8. With recent versions like Java 10, a unique version numbering convention is used.

The Java classes defined by Genero (com.fourjs.fgl.lang.*) are compiled with `javac -source 1.6 -target 1.6` options. Therefore, the minimum theoretical version is Java SE 6. However, depending on the platform, the minimum required Java version may be greater than version 6.

**Note:** As a general rule, always install the current long-term-support (LTS) Java version available on your platform, with JNI support. For example, Java SE 8 (released in March 2014) and Java SE 11 (released in September 2018) are LTS versions.

For a detailed list of supported JVMs, refer to the [Supported platforms and databases](#) document, available on the Products download page of the Four Js Web site.

The version of the installed Java software can be shown with the command:

```
java --version
```

In order to execute Java byte code, the Genero runtime system uses the JNI interface. The JVM is loaded as a shared library and its binary format must match the binary format of the Genero runtime system. For example, a 64-bit Genero package requires a 64-bit JVM.

When implementing Java classes for Genero Mobile for Android™ (GMA), check the JDK version required by the Android™ SDK. For more information, see the [Android™ Studio web site](#).

**Related concepts**
- *The Java interface* on page 2192
- *The Java interface* allows you to import Java classes and instantiate Java objects in your programs.

### IPv6 support with Genero

**IPv6 basics**

IPv6 is the successor for IPv4, to increase the possible number of nodes of a computer network.

**IPv6 support for WS clients**

A Web Services client program can by default access a WS server using IPv6. For more details, see [Configure a WS client to use IPv6](#) on page 3340.

**Note:** Web Services server programs work only in IPv4 to communicate with the GAS (since there is no need for IPv6 on a localhost). It’s up to the Web server to support IPv6 for the internet access of WS clients.

**Related concepts**
- *Web services* on page 3270
- Create a Web service client or server with Genero BDL.

---

### Installing Genero BDL

This section provides Genero BDL installation instructions.

Different forms of installation programs are provided, as individual packages are bundled with other Genero components. Refer to the appropriate installation guide for a detailed description of the installation procedure. Do not hesitate to contact your support center if you need help.

After installing a package, it is recommended that you:
1. Set the FGLDIR on page 275 environment variable to the BDL installation directory.
2. Add $FGLDIR/bin to the PATH environment variable, in order to run compilers and runtime system tools from the command line.
3. Set the database client software environment (for example, INFORMIXDIR, etc)
4. Set access path to database client software DLLs (PATH), or UNIX® shared libraries (LD_LIBRARY_PATH, SHLIB_PATH, LIBPATH)
5. Depending on the database server you want to connect to, set up the correct database driver in FGLPROFILE. The default database driver is selected during the installation procedure.
6. Depending on what rendering mode you want to use (text mode or graphical mode), you will have to set environment variables such as FGLGUI on page 275, FGLSERVER on page 280, TERM, INFORMIXTERM.
7. If your application uses C-Extensions, a C compiler is required and you must recompile your C-Extensions as shared libraries.
8. If your application uses the Java Interface, setup the required JDK environment.

Related concepts
Genero environment variables on page 266
Operating system environment variables on page 264
Describes some well-known system environment variables that are used by Genero software components.

Related tasks
Install Genero Mobile for Android on page 40
To build and package Genero Mobile for Android™ (GMA) applications, you must first install GMA.
Install Genero Mobile for iOS (single version) on page 42
To build and package Genero Mobile for iOS (GMI) applications, you must first install GMI. This topic explains how to install a unique GMI version/package into FGLDIR.
Install Genero Mobile for iOS (multiple versions) on page 42
To build and package Genero Mobile for iOS (GMI) applications, you must first install GMI. This topic explains how to install multiple GMI versions/packages in parallel into different GMIDIR directories.

Upgrading Genero BDL
Genero BDL upgrading requires several steps to consider.

After upgrading to a newer version, follow these next steps:

1. If the new version is a major upgrade (for example, from 2.20 to 2.21), recompile the sources and form files.
   While recompilation is not needed when migrating to maintenance release versions (for example, from 2.21.01 to 2.21.02), there is potentially some p-code optimization benefits to be gained from doing so.
2. If required, you may need to recreate the C-Extensions libraries. C extension libraries must be provided as dynamically loadable modules and a rebuild is generally not needed. However, if the C-Extension API header files have changed, consider recompiling your C sources. Check $FGLDIR/include/f2c for C Extension API header file changes.

Related concepts
Upgrading on page 48
These topics talk about what steps you need to take to upgrade to the next release of Genero Business Development Language, and allows you to identify which features were added for a specific version.

Install Genero Mobile for Android™
To build and package Genero Mobile for Android™ (GMA) applications, you must first install GMA.

Before you begin:
**Important:** The GMA and Genero BDL X.YZ versions are interdependent. For example, GMA 1.30 is required to work with Genero BDL version 3.10, GMA 1.40 is required to work with Genero BDL 3.20.

- Download Genero Mobile for Android™ (GMA) from the Four Js Web site.

1. Install the Java Development Kit (JDK).
   **Important:** JDK version 8 is required to build Android™ apps. For the latest information regarding system requirements and Java support, please refer to the Supported platforms and databases document, available on the "Products" download page of the Four Js Web site.

2. Prepare the Android™ SDK installation directory: First install the Android Command Line Tools and setup the environment. The download of Android SDK packages will be done in a next step with gmabuildtool updatesdk:
   a) Create a directory to hold the Android SDK packages and tools.
   b) Set the ANDROID_SDK_ROOT environment variable with the new created directory.
   c) Create the cmdline-tools directory in $ANDROID_SDK_ROOT.
   d) Download the commandlinetools zip file and unzip the ZIP archive into the $ANDROID_SDK_ROOT/cmdline-tools directory.

   **Note:** On Windows 10, the default zip tool provided by the operating system will corrupt the Android Command Line Tools zip archive. The error displayed is "Error: Could not find or load main class com.android.sdklib.tool.sdkmanager.SdkManagerCli". To avoid this error, use another zip program such as 7zip.
   e) In $ANDROID_SDK_ROOT/cmdline-tools, rename the directory "tools" to "latest".
   f) Add "$ANDROID_SDK_ROOT/cmdline-tools/latest/bin" to your PATH environment variable, to find the sdkmanager command.

3. Install the GMA buildtool and the GMA binary archive.
   They are provided in the GMA distribution archive (fjs-gma-*.zip).
   a) Create a directory (gma-install-dir) for the GMA development tools.

   ```
   $ mkdir /opt/fourjs/gma-1.30
   ```
   b) Extract the content of the Genero Mobile for Android™ package (fjs-gma-*.zip) into this directory.

   ```
   $ unzip -q -o -d /opt/fourjs/gma-1.30 fjs-gma-*.zip
   ```
   c) Add the gma-install-dir directory to your PATH environment variable, in order to find the gmabuildtool command.

4. Execute the gmabuildtool updatesdk command.
   An internet connection is required.
   **Important:** Execute the gmabuildtool updatesdk command every time a new version of the GMA buildtool and GMA binary archive is installed.

5. If you plan to publish your app on Google Play, register to Google Play as a developer and create a Google Play project.

6. If you have installed Cordova plugins, you need to re-install the plugins with the --install-plugin option of gmabuildtool. For more details, see Installing Cordova plugins on page 3645.

**Related concepts**
Building Android apps with Genero on page 3591
Genero provides a command-line tool to create applications for Android™ devices.

### Install Genero Mobile for iOS (single version)

To build and package Genero Mobile for iOS (GMI) applications, you must first install GMI. This topic explains how to install a unique GMI version/package into FGLDIR.

Before you begin:

**Important:** The GMI and Genero BDL X.Y.Z versions are interdependent. For example, GMI 1.30 is required to work with Genero BDL version 3.10, GMI 1.40 is required to work with Genero BDL 3.20, etc.

- Download the Genero Mobile for iOS (GMI) distribution archive from the Four Js Web site. GMI can only be installed on a Mac® OS X computer.

Follow this procedure if you are using only one version of GMI for a Genero BDL installation.

**Important:** When re-installing a new GMI archive, remove all "build" directories created by the gmibuildtool.

1. **Install Xcode.**
   
   The Xcode® version must support the iOS versions of your mobile devices.
   
   **Tip:** As a general rule, update the Xcode® and iOS to the latest versions

2. Define the FGLDIR environment variable with the Genero BDL installation directory.

   ```sh
   $ export FGLDIR=/opt/fourjs/fglgws-3.10
   ```

3. Extract the GMI distribution archive (`fjs-fglgmi-*.zip`) into FGLDIR.

   The GMI distribution archive contains the GMI buildtool and the needed iOS libraries and helper scripts.

   ```sh
   $ unzip -q -o -d $FGLDIR fgl-fglgmi-*.zip
   ```

4. If you have installed Cordova plugins, you need to re-install the plugins with the `--install-plugin` option of gmibuildtool. For more details, see Installing Cordova plugins on page 3645.

**Related concepts**

Building iOS apps with Genero on page 3606

Genero provides a command-line tool to build applications for iOS devices.

### Install Genero Mobile for iOS (multiple versions)

To build and package Genero Mobile for iOS (GMI) applications, you must first install GMI. This topic explains how to install multiple GMI versions/packages in parallel into different GMIDIR directories.

Before you begin:

**Important:** The GMI and Genero BDL X.Y.Z versions are interdependent. For example, GMI 1.30 is required to work with Genero BDL version 3.10, GMI 1.40 is required to work with Genero BDL 3.20, etc.

- Download the Genero Mobile for iOS (GMI) distribution archive from the Four Js Web site. GMI can only be installed on a Mac® OS X computer.

If you need to test several versions of GMI against a single installation of Genero BDL, you will extract each GMI distribution archive to a separate directory. You can then quickly change the version of GMI used by resetting the GMIDIR and PATH environment variables.
**Important:** When re-installing a new GMI archive, remove all "build" directories created by the `gmibuildtool`.

1. Install Xcode®.
   
The Xcode® version must support the iOS versions of your mobile devices.
   
   **Tip:** As a general rule, update the Xcode® and iOS to the latest versions

2. Create a directory for the GMI distribution archive.

   ```bash
   $ mkdir /opt/fourjs/gmi-1.30
   ```

3. Define the GMIDIR environment variable with the GMI installation directory.

   ```bash
   $ export GMIDIR=/opt/fourjs/gmi-1.30
   ```

4. Extract the GMI distribution archive (`fjs-fglgmi-*.zip`) into GMIDIR.
   
The GMI distribution archive contains the GMI buildtool and the needed iOS libraries and helper scripts.

   ```bash
   $ unzip -q -o -d $GMIDIR fgl-fglgmi-*.zip
   ```

5. Update PATH to include `$GMIDIR/bin`.

   ```bash
   $ export PATH=$GMIDIR/bin:$PATH
   ```

6. If you have installed Cordova plugins, you need to re-install the plugins with the `--install-plugin` option of `gmibuildtool`. For more details, see Installing Cordova plugins on page 3645.

**Related concepts**

- Building iOS apps with Genero on page 3606
- Genero provides a command-line tool to build applications for iOS devices.

---

**Platform specific notes**

These notes provide operating system specific information to use Genero BDL on the platform.

**HP-UX configuration notes**

**Thread Local Storage in shared libraries**

On HP-UX, the shared library loader cannot load libraries using Thread Local Storage (TLS), like Oracle® `libclntsh`. In order to use shared libraries with TLS, you must use the `LD_PRELOAD_ONCE` environment variable. For more details, search for "shl_load + Thread Local Storage" on the HP® support site.

**PostgreSQL on HP-UX LP64**

On HP-UX® LP64, it is recommended that the PostgreSQL database driver is linked with the `libxnet.sl` library if you want to use networking. You can force the usage of `libxnet` by setting the `LD_PRELOAD_ONCE` environment variable to `/lib/pa20_64/libxnet.sl`.

**Java Interface**

Check [HP/UX platform notes](#) when using the Java Interface on this operating system.
IBM® AIX® configuration notes

**LIBPATH environment variable**

The LIBPATH environment variable defines the search path for shared libraries. Make sure LIBPATH contains all required library directories, including the system library path /lib and /usr/lib.

**Shared libraries archives**

On AIX®, shared libraries are usually provided in .a archives containing the shared object(s). For example, the DB2® client library libdb2.a contains both the 32-bit (shr.o) and the 64-bit (shr_64.o) versions of the shared library. Not all products follow this rule; for example Oracle® 9.2 provides libclntsh.a with shr.o on 64-bit platforms, and Informix® provides both .a archives with static objects and .so shared libraries as on other platforms.

The runtime system database drivers are created with the library archives or with the .so shared objects, based on the database type and version. No particular manipulation is needed to use any supported database client libraries on this platform.

**The dump command**

On IBM® AIX®, you can check the library dependencies with the dump command:

```bash
$ dump -Hv -X64 libstckp.so
```

**Unloading shared libraries from memory**

In production environments, AIX® loads shared libraries into the system shared library segment in order to improve program load time. Once a shared library is loaded, other programs using the same library are attached to that memory segment.

Once a shared library is loaded by the system, you cannot copy the executable file unless you unload the library from the system memory. This problem will occur when installing a new version of the software, even if it is installed in a different directory. Since shared libraries have the same name, AIX® will not allow multiple versions of the same library to load. Therefore, before installing a new version, make sure all shared libraries are unloaded from memory.

The genkld command prints the list of shared libraries currently loaded into memory. The slibclean command unloads a shared library from the system shared library segment.

**POSIX Threads and shared libraries**

When using a thread-enabled shared library like Oracle's libclntsh, the program using the shared object must be linked with thread support, otherwise you can experience problems (like segmentation fault when the runner program ends). IBM® recommends using the xlc_r compiler to link a program with pthread support.

By default, the runtime system provided for AIX® platforms is linked with pthread support.

**Java Interface**

Check IBM® AIX® platform notes when using the Java Interface on this operating system.

Mac® OS X® configuration notes

**DYLD_LIBRARY_PATH denied in OS X® 10.11**

Starting with Mac® OS X® 10.11 (El Capitan), if the System Integrity Protection (SIP) is enabled, the DYLD_LIBRARY_PATH environment variable is no longer exported in sub processes. This variable defined the shared library search path for software components used by the Genero runtime system. This was required especially for database client libraries installed in directories other than /usr/lib and /usr/local/lib (the default location for shared libraries).
As DYLD_LIBRARY_PATH cannot be used, the proper workaround is to install all required shared libraries in /usr/local/lib. A good practice is to create the installation directory of the software component in /usr/local/product/version, and create symbolic links to the required shared libraries in /usr/local/lib.

To make sure that all required libraries can be found, check the dependencies on a Genero binary (typically, the ODI driver) with the otool -L and otool -l commands.

For example, with Oracle 12 Instant Client:

```
$ mkdir /usr/local/oracle/
$ mkdir /usr/local/oracle/instantclient_12_2
... install Oracle Instant Client in /usr/local/oracle/instantclient_12_2 ...
$ cd /usr/local/lib
$ ln -s /usr/local/oracle/instantclient_12_2/libclntsh.dylib.12.1 libclntsh.dylib.12.1
... set Genero FGL environment ...
$ otool -L $FGLDIR/dbdrivers/dbmora_12.dylib
... check dependencies and paths
$ otool -l $FGLDIR/dbdrivers/dbmora_12.dylib
... check load sequences
```

**Important:** If Mac OSX includes a database client library in system directories such as /usr/lib, the system shared library will be found first, before user shared libraries located in /usr/local/lib. For example, Mac OSX comes with a PostgreSQL client library /usr/lib/libpq.5.dylib -> libpq.5.6.dylib. In this case, if you want to use a more recent PostgreSQL client library, you will have to patch the Genero ODI driver with the install_lib_tool, using the -change option to use the @rpath prefix to libpq.5.dylib shared library entry, and add a search path with -add_rpath to specify target PostgreSQL lib directory:

```
$ cd $FGLDIR/dbdrivers
$ otool -L dbmpgs_9.dylib
... original-path-to-postgresql-dir/libpq.5.dylib (compatibility ...
...
$ install_name_tool \
  -change original-path-to-postgresql-dir/libpq.5.dylib @rpath/libpq.5.dylib \
  -add_rpath /Library/Postgresql/9.5.1/lib \
  dbmpgs_9.dylib
$ otool -L dbmpgs_9.dylib
... @rpath/libpq.5.dylib (compatibility ...
...
$ otool -l dbmpgs_9.dylib
... check load sequences
```

**Java Interface**

Check [Mac OS X platform notes](#) when using the Java Interface on this operating system.
Microsoft™ Windows® configuration notes

**Microsoft™ Visual C++ version**

When using C-Extensions, you need Microsoft™ Visual C++ compiler to compile and link your C sources. Make sure you have installed the software package corresponding to the MSVC version installed on your system. The MSVC version is identified in the software package name.

**Checking binary dependencies**

Microsoft™ Visual C++ provides the dumpbin utility to extract information from a binary file.

Use the /dependents option to check for DLL dependencies:

```
C:\> dumpbin /dependents mylib.dll
Microsoft (R) COFF/PE Dumper Version 7.10.3077
Copyright (C) Microsoft Corporation. All rights reserved.

Dump of file mylib.dll

File Type: EXECUTABLE IMAGE

Image has the following dependencies:

isqlt09a.dll
MSVCR71.dll
KERNEL32.dll

Summary

1000 .data
1000 .rdata
1000 .text
```

**Changing the stack size of fglrun**

On Windows® platforms, the fglrun.exe binary has a predefined C stack size. In some rare cases (for example, when programs do deep recursion), the stack size of fglrun.exe binary needs to be changed to avoid a stack overflow. The stack size of fglrun can be changed permanently by patching the EXE file with the Microsoft™ Visual C++ editbin utility. Check the stack size by running the dumpbin utility on fglrun.exe as follows:

```
C:\> dumpbin /headers %FGLDIR%\bin\fglrun.exe
```

Search for the line containing "stack reserve" words in the OPTIONAL HEADER VALUES section:

```
OPTIONAL HEADER VALUES

... 100000 size of stack reserve
```

The stack size is displayed in hexadecimal value. So for example, a value 100,000 means 1,048,567 bytes = 1MB.

In order to modify the stack size of fglrun.exe, run the editbin utility on fglrun.exe with the /stack option:

```
C:\> editbin /stack:1000000 %FGLDIR%\bin\fglrun.exe
```

See Microsoft™ Visual C++ documentation for more details.

**Java Interface**

Check Microsoft™ Windows® platform notes when using the Java Interface on this operating system.
With C-Extensions, you can bind your own C libraries in the runtime system, to call C function from the application code.

Web Services platform notes

Genero Web Services reference documentation contains a list of platform specific notes to consider. For more details, see Web Services platform specific notes.

SUSE® Linux Enterprise

Important: The fglrun component of Genero BDL package requires version 5 of the wide-char Curses shared library to be installed on the system.

Installing the Curses library on SUSE Linux Enterprise 15 (SLES 15)

On SUSE® Linux Enterprise 15, the version 5 Curses library must be installed as follows:

```
# SUSEConnect -p sle-module-legacy/15/x86_64
# zypper install libncurses5
```

Note: On SUSE, the libncurses5 RPM package includes the libncursesw.so.5 library for UTF-8 support in text mode.

Debian / Ubuntu Linux

Important: The fglrun component of Genero BDL package requires version 5 of the wide-char Curses shared library to be installed on the system.

Installing the Curses library on Debian 9 or Ubuntu 18

On Debian Linux 9 (and derived Linux brands like Ubuntu), the Curses library version 5 must be installed as follows:

```
# apt-get install libncurses5
# apt-get install libncursesw5
```

Note: On Debian 9, the libncursesw.so.5 library for UTF-8 support in text mode is part of the libncursesw5 package: The libncurses5 package does not include the libncursesw.so.5 library.

Installing the Curses library on Debian 10 or Ubuntu 19

To install version 5 of the Curses library on Debian 10 or Ubuntu 19, do the following:

```
# apt-get install libncurses5
# apt-get install libncursesw5
```

Red Hat® Enterprise Linux® / CentOS

Important: The fglrun component of Genero BDL package requires version 5 of the wide-char Curses shared library to be installed on the system.
Installing the Curses library on RedHat Enterprise Linux 7 / CentOS 7

On Red Hat® Enterprise Linux® 7 and Centos 7, the Curses library must be installed as follows. This will install the version 5 of the Curses library:

```
# yum install ncurses-libs
```

**Note:** On Red Hat Enterprise Linux 7 and Centos 7, the ncurses-libs RPM package includes the libncursesw.so.5 library for UTF-8 support in text mode.

Installing the Curses library on RedHat Enterprise Linux 8 / CentOS 8

In order to install the version 5 of the Curses library on Red Hat® Enterprise Linux® 8 and Centos 8, execute the following command:

```
# yum install ncurses-compat-libs
```

**Note:** On Red Hat Enterprise Linux 8 and Centos 8, the ncurses-compat-libs RPM package includes the libncursesw.so.5 library for UTF-8 support in text mode.

Related concepts
- TERMINFO terminal capabilities on page 1123

Upgrading

These topics talk about what steps you need to take to upgrade to the next release of Genero Business Development Language, and allows you to identify which features were added for a specific version.

Related concepts
- Installation on page 37
  - This chapter contains installation and setup instructions.

New features of Genero BDL

These topics provide an look back at the new features introduced with each release of the Genero Business Development Language.

Related concepts
- Upgrade Guides for Genero BDL on page 109
  - Provides information about product changes to be considered when migrating to a new version.

- Planned desupport on page 211
  - Features described in this topic might be desupported in a future release of the product.

BDL 3.20 new features

Features added in 3.20 releases of the Genero Business Development Language.

**Important:** This page covers only those new features introduced with the Genero BDL version specified in the page title. Check prior new features pages if you migrate from an earlier version. Make sure to also read the upgrade guide corresponding to this Genero version.

Table 9: Core language

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support for circular dependency of modules with IMPORT FGL.</td>
<td>See Circular dependency with IMPORT FGL on page 126.</td>
</tr>
<tr>
<td>Method declaration, to define functions acting on a user-defined type.</td>
<td>See Methods on page 452, FUNCTION syntax.</td>
</tr>
<tr>
<td>INTERFACE structure, to define a group of methods for a type.</td>
<td>See Interfaces on page 457.</td>
</tr>
<tr>
<td>Passing records by reference (INOUT).</td>
<td>See Passing records by reference with INOUT on page 495.</td>
</tr>
<tr>
<td>Variable initializers: The DEFINE instruction supports a clause to initialize the variable.</td>
<td>See Variable initializers on page 396, DEFINE on page 390.</td>
</tr>
<tr>
<td>Named parameters in function calls: CALL func(p1:value).</td>
<td>See Naming parameters in a function call on page 443.</td>
</tr>
<tr>
<td>Function attributes (for RESTful Web Services). A FUNCTION definition can specify function attributes, parameter attributes and return values attributes.</td>
<td>See Function attributes on page 455, FUNCTION definitions on page 437.</td>
</tr>
<tr>
<td>New STRING and base.StringBuffer methods to remove all kinds of whitespace characters in a string.</td>
<td>See the *WhiteSpace methods in STRING data type methods on page 2382 and base.StringBuffer methods on page 2448.</td>
</tr>
<tr>
<td>FUNCTION definition syntax allows the RETURNS () clause, to specify an empty return list and enforce compilation verifications to avoid RETURN instruction misusage.</td>
<td>See FUNCTION syntax, Returning values on page 446.</td>
</tr>
<tr>
<td>Arrays can be assigned without the .* notation.</td>
<td>See Dynamic array assignment with .* notation on page 125.</td>
</tr>
<tr>
<td>Records can be assigned without the .* notation.</td>
<td>See Record copy with .* notation on page 125.</td>
</tr>
<tr>
<td>FGLPROFILE fglrun.floatToCharScale2 and fglrun.floatToCharScale2.print to control FLOAT/SMALLFLOAT to string conversion.</td>
<td>See FLOAT/SMALLFLOAT to string conversion on page 127.</td>
</tr>
<tr>
<td>GetOpt utility type and methods, to process command line arguments in the getopt style (--option1=value1).</td>
<td>See getopt: Command line options module on page 2365.</td>
</tr>
<tr>
<td>New os.Path.glob() method to get a list of files from a pattern.</td>
<td>See os.Path.glob on page 2801.</td>
</tr>
<tr>
<td>TEXT and BYTE variables are now automatically located in memory by JSON parsing methods.</td>
<td>See JSON to BDL data conversion.</td>
</tr>
</tbody>
</table>

Table 10: User interface

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>The new gui.rendering FGLPROFILE entry, to enable Universal Rendering.</td>
<td>See Graphical mode with Universal Rendering on page 1112.</td>
</tr>
<tr>
<td>FGLGBCDIR environment variable, to define the directory of the GBC component to be used for Universal Rendering.</td>
<td>See FGLGBCDIR on page 275, Graphical mode with Universal Rendering on page 1112.</td>
</tr>
</tbody>
</table>
### Overview

**KEYBOARDHINT** form attribute allows now the URL value to define an EDIT field for URL input.

GBC supports the "chrome" value for actionPanelPosition, ringMenuPosition and toolBarPosition window style attributes.

`ui.Interface` class methods to identify the Universal Rendering client and version.

Form elements of type **GROUP** can be defined as collapsible with new style attributes (`collapsible`/`initiallyCollapsed`).

Collapsible groups and folders can be configured to display the collapser icon at a specific position with the collapserPosition style attribute.

**ON FILL BUFFER** trigger for display array dynamic dialog.

Built-in classes methods to control form elements are now case-insensitive.

Use the `itemsAlignment` style attribute to control the alignment of elements inside a scrollgrid.

**New enableCalendar** style attribute for DateTimeEdit with GDC.

**New qtStyle** style attribute for ComboBox with GDC.

The **INITIALIZER** attribute of COMBOBOX accepts module specification as function prefix (the module is then automatically loaded)

Setting the comment of a form element or form field.

**New FGLPROFILE** entry `gui.programStoppedMessage` to define a generic message to be displayed to the end user when a program stops because of a runtime error.

Upgrade notes for presentation styles.

### Reference

See **KEYBOARDHINT attribute** on page 1378.

See **Action views rendering in GBC chrome bar** on page 115.


See **Collapsible groups with GBC** on page 115.

See **folder.collapsePosition (GBC)** on page 115, **Collapsible groups with GBC** on page 115.

See **ui.Dialog.addTrigger** on page 2524.

See **Case insensitive names with UI methods** on page 126.

See **Controlling element alignment inside a scrollgrid** on page 1886.

See **New DateTimeEdit.enableCalendar style attribute (GDC)** on page 116.

See **New ComboBox.qtStyle style attribute (GDC)** on page 116.

See **INITIALIZER attribute** on page 1374.

See **ui.Form.setElementComment** on page 2505, `ui.Form.setFieldComment` on page 2508.

See **Default exception handling** on page 508, **Program stop error message box** on page 150.

See **Presentation styles changes** on page 115.

### Table 11: SQL databases:

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support for Oracle® database 18c and 19c with the dbmora_18 driver.</td>
<td>See <strong>Database drivers changes</strong> on page 117.</td>
</tr>
<tr>
<td>Using Oracle® 18c private temporary tables to emulate Informix® temporary tables.</td>
<td>See <strong>Using the private temporary table emulation</strong> on page 949.</td>
</tr>
<tr>
<td>Overview</td>
<td>Reference</td>
</tr>
<tr>
<td>-------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Support for PostgreSQL 11 and 12 with the dbmpgs_9 driver.</td>
<td>See Database drivers changes on page 117, PostgreSQL 12 notes on page 124.</td>
</tr>
<tr>
<td>Support for SQL Server 2019 (v15) with the dbmsnc_17 driver.</td>
<td>See Database drivers changes on page 117.</td>
</tr>
<tr>
<td>New dbmmys_5_6 driver for MySQL 5.6 as replacement of dbmmys_5_5 no</td>
<td>See Database drivers changes on page 117.</td>
</tr>
<tr>
<td>longer available because of MySQL 5.5 desupport.</td>
<td></td>
</tr>
<tr>
<td>New ifxemul.datatype.serial.sqlerrd[2] FGLPROFILE entry to disable</td>
<td>See FGLPROFILE entries for core language on page 256,</td>
</tr>
<tr>
<td>Utility function db_get_last_serial() to retrieve the last generated</td>
<td>See db_get_last_serial() on page 2308.</td>
</tr>
<tr>
<td>serial, mandatory for BIGINT incremented columns.</td>
<td>See PostgreSQL specific FGLPROFILE parameters on page 676.</td>
</tr>
<tr>
<td>FGLPROFILE entry dbi.database.dbname.pgs.schema to define the schema</td>
<td>See note in Performances with SQL interruption on page 124.</td>
</tr>
<tr>
<td>search path for PostgreSQL.</td>
<td></td>
</tr>
<tr>
<td>FGLSQLDEBUG output enhancement to identify when SQL INTERRUPT ON/OFF</td>
<td>See CONNECT TO on page 683, SET SESSION AUTHORIZATION on page 685.</td>
</tr>
<tr>
<td>is used.</td>
<td></td>
</tr>
<tr>
<td>Support for IBM® Informix® trusted connections, with the new TRUSTED</td>
<td>See Row limiting clause (SELECT) on page 650.</td>
</tr>
<tr>
<td>keyword for CONNECT TO, and the new SET SESSION AUTHORIZATION statement.</td>
<td></td>
</tr>
<tr>
<td>Informix SQL SELECT statement row limiting clause SKIP n FIRST m can</td>
<td>See character types with SQL Server, ifxemul FGLPROFILE settings.</td>
</tr>
<tr>
<td>be converted to native equivalent.</td>
<td></td>
</tr>
<tr>
<td>With Microsoft SQL Server, in DDL statements such as CREATE TABLE,</td>
<td></td>
</tr>
<tr>
<td>character data types can be converted to the equivalent native</td>
<td></td>
</tr>
<tr>
<td>national character types, with FGLPROFILE entry</td>
<td></td>
</tr>
<tr>
<td>dbi.database.dsname.ifxemul.nationalchars=true.</td>
<td></td>
</tr>
<tr>
<td>To avoid Microsoft SQL Server 468 collation conflict errors with</td>
<td></td>
</tr>
<tr>
<td>temporary tables, the ODI drivers for SQL Server add now a COLLATE</td>
<td></td>
</tr>
<tr>
<td>DATABASE_DEFAULT clause after character types in the native statement</td>
<td></td>
</tr>
<tr>
<td>generated from a CREATE TEMP TABLE. This is not needed for SELECT</td>
<td></td>
</tr>
<tr>
<td>INTO TEMP.</td>
<td></td>
</tr>
<tr>
<td>Oracle® database rowid (in base 64 format) can be found in</td>
<td>See ORACLE rowid in SQLCA.SQLERRM on page 128.</td>
</tr>
<tr>
<td>SQLCA.SQLERRM.</td>
<td></td>
</tr>
<tr>
<td>Upgrade notes for database drivers.</td>
<td>See Database drivers changes on page 117.</td>
</tr>
<tr>
<td>Upgrade notes for front calls.</td>
<td>See Front calls changes on page 116.</td>
</tr>
<tr>
<td>Upgrade notes for web components.</td>
<td>See Web components changes on page 117.</td>
</tr>
</tbody>
</table>
### Table 12: Tools

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source code formatting tool: <code>fglcomp --format</code>, including new command line tools <code>fglgitformat</code>, <code>fglformatdiff</code>.</td>
<td>See <a href="#">Source code beautifier</a> on page 2174, <a href="#">fglgitformat</a> on page 2093, <a href="#">fglformatdiff</a> on page 2094.</td>
</tr>
<tr>
<td>Qualifying imported symbols: <code>fglcomp --qualify-imports</code>.</td>
<td>See <a href="#">Qualifying imports</a>.</td>
</tr>
<tr>
<td>Use lowercase keywords in VIM with <code>fgl_lowercase_keywords=1</code>.</td>
<td></td>
</tr>
<tr>
<td><code>fglrun</code>/<code>fgllink</code> supports @argfile to specify the list of files to link.</td>
<td>See <a href="#">Providing the files to link in an arguments file</a> on page 2115, Providing the files to link in an arguments file on page 2114.</td>
</tr>
<tr>
<td><code>fglcomp</code> can now process several <code>.4gl</code> files on the command line.</td>
<td></td>
</tr>
<tr>
<td>Commands accepting a list of files like <code>fglcomp</code> or <code>fgllink</code>, can do pathname expansion (<code>fglcomp * .4gl</code>), on Unix and Windows platforms.</td>
<td>See <a href="#">Compiling several <code>.4gl</code> sources in a single command</a> on page 2108, <code>fglcomp</code> on page 2071.</td>
</tr>
<tr>
<td>To reduce compilation time of large projects, <code>fglcomp</code> supports the <code>--make</code> option, to compile the provided <code>.4gl</code> sources only when the <code>.42m</code> is not up-to-date. The <code>--make</code> option can be used with the <code>--simulate</code> option, to simulate the process without compiling.</td>
<td>See <a href="#">Compiling in make mode</a> on page 2109.</td>
</tr>
<tr>
<td>The <code>fglcomp --dependencies</code> option can be used to produce makefile-style dependency rules for imported modules.</td>
<td>See <a href="#">Producing make-style dependency rules</a> on page 2111.</td>
</tr>
<tr>
<td><code>fglcomp</code> supports @argfile to specify the list of source files in an input file.</td>
<td>See <a href="#">Providing the source files in an arguments file</a> on page 2111.</td>
</tr>
<tr>
<td><code>fglcomp</code> supports the value &quot;auto&quot; for <code>-p preprocessor type option</code>.</td>
<td>See <a href="#">Preprocessing style option</a> on page 2133.</td>
</tr>
<tr>
<td><code>fglrun</code> provides the <code>--print-missing-imports</code> option to show only missing IMPORT FGL (<code>--print-imports</code> option prints required imports even if IMPORT FGL is used)</td>
<td>See <a href="#">Identifying modules to be imported</a> on page 483.</td>
</tr>
<tr>
<td><code>fglcomp</code> supports the new warning argument <code>-W unused-parameter</code>, that prints a message when a function parameter is not used. The negative form <code>-W no-unused-parameter</code> can be combined with <code>-W unused</code>, to ignore parameters that are unused.</td>
<td>See <a href="#">fglcomp option -W option</a>.</td>
</tr>
</tbody>
</table>
The Genero Web Service engine has been enhanced with a mechanism to provide a high level REST web service. You can implement REST Web services using function attributes (identified with WS* prefix) that you specify in `ATTRIBUTES()` clauses of functions.

Descriptions using the `WSDescription` attribute are allowed in input and output parameters. These descriptions can be generated in the REST stub file using the `fglrestful` tool's `--comment` option.

The `com.WebServiceEngine` option `server_restdefaultformat` added to define runtime support for MIME types in REST operations.

Method added to publish high-level RESTful Web services in the GWS engine.

Method added to provide error management in the high-level RESTful Web service.

Empty HTTP POST and PUT requests.

The `fglrestful` tool takes an OpenAPI specification file (JSON format) and generates the client stub to interact with the REST service.

The `http.global.request.date` entry can be set to `true`, in order to force sending HTTP Date header for GET, HEAD and DELETE requests.

The `com.HTTPRequest.setAuthentication` has an enhancement to support the NTLM authentication protocol of Windows® server that requires the connection to be kept open.

API added to provide management of OAuth authentication for RESTful Web service secured by OpenID Connect.

Upgrade notes for web services.

---

### Table 14: Mobile apps (GMA 1.40 and GMI 1.40)

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gmabuildtool</code> option <code>-bgr</code> / <code>--build-gbc-runtime</code>, to define the GBC to be used to build the app.</td>
<td>See Building Android apps with Genero on page 3591.</td>
</tr>
<tr>
<td><code>gmibuildtool</code> option <code>--gbc</code>, to define the GBC to be used to build the app.</td>
<td>See Building iOS apps with Genero on page 3606.</td>
</tr>
</tbody>
</table>
Overview | Reference
--- | ---
GMA supports now the alignment style attribute for BUTTON. | See Button.alignment style attribute (GMA) on page 116, Button style attributes on page 1185.
Cordova plugin wrapper libraries now available on the FOURJS Cordova Github. | See Cordova plugins for GMA, Cordova plugins for GMI.
New Cordova plugin front calls to get plugin information. | See New Cordova front calls in GMA, New Cordova front calls in GMI.

Table 15: Miscellaneous

Overview | Reference
--- | ---
New parameter for monitor.update front call to control administrator permission elevation prompt. | See Front calls changes on page 116, monitor.update front call.

**BDL 3.10 new features**
Features added in 3.10 releases of the Genero Business Development Language.

**Important:** This page covers only those new features introduced with the Genero BDL version specified in the page title. Check prior new features pages if you migrate from an earlier version. Make sure to also read the upgrade guide corresponding to this Genero version.

Corresponding upgrade guide: BDL 3.10 upgrade guide on page 129.

Table 16: Core language

Overview | Reference
--- | ---
The **DICTIONARY** type, to define associative arrays / hash-maps. | See Dictionaries on page 426, DICTIONARY as class on page 2400.
New **FUNCTION** definition syntax, to declare exact function signature with parameter types in the parentheses, and return types in the **RETURNS** clause. | See FUNCTION syntax, Function parameters on page 444, Returning values on page 446.
Function references, to call functions only know at runtime. | See Function references on page 449, FUNCTION func-spec on page 451.
Invoking methods with an object reference returned from a method: LET s = s.subString(1,10).toLowerCase() | See Working with objects on page 571.
Java SE 10 and 11 support. | See Java Interface changes on page 135, Java software requirements for FGL on page 2193.
New **DYNAMIC ARRAY** methods **copyTo()**, **search()**, **searchRange()**. | See DYNAMIC ARRAY.copyTo on page 2395, DYNAMIC ARRAY.search on page 2397, DYNAMIC ARRAY.searchRange on page 2398.
Defining the localized string lookup path when program starts. | See base.Application.reloadResources on page 2411.
<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit-wise operation and hexadecimal/binary string conversion on INTEGER values.</td>
<td>See The util.Integer class on page 2735.</td>
</tr>
<tr>
<td>Channel openFile() method can write to stderr.</td>
<td>See base.Channel.openFile on page 2417.</td>
</tr>
<tr>
<td>Variable definition attributes for JSON serialization: json_null, json_name.</td>
<td>See JSON support on page 575.</td>
</tr>
<tr>
<td>Resource files (.42f, etc) can now also be found in the directory where the MAIN module resides.</td>
<td>See Default resource file search path on page 148, FGLRESOURCEPATH on page 279.</td>
</tr>
<tr>
<td>Base64 to/from Hexadecimal string conversion methods.</td>
<td>See util.Strings.base64DecodeToHexString on page 2755, util.Strings.base64EncodeFromHexString on page 2756.</td>
</tr>
<tr>
<td>Source documentation enhancements, to describe module variables, constants and user defined types.</td>
<td>See Commenting a module constant on page 2127, Commenting a module variable on page 2128, Commenting a module type on page 2128.</td>
</tr>
<tr>
<td>fglform checks for unique TABINDEX values.</td>
<td>See Unique TABINDEXes in a form on page 142.</td>
</tr>
<tr>
<td>Upgrade notes for C extensions.</td>
<td>See C Extension changes on page 134.</td>
</tr>
<tr>
<td>Upgrade notes for Java interface.</td>
<td>See Java Interface changes on page 135.</td>
</tr>
</tbody>
</table>

Table 17: User interface

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Click on cell in DISPLAY ARRAY with FOCUSONFIELD attribute.</td>
<td>See Field-level focus in DISPLAY ARRAY on page 1840.</td>
</tr>
<tr>
<td>Get user's preferred language defined on the front-end platform.</td>
<td>See User's preferred language on page 533.</td>
</tr>
<tr>
<td>New &quot;accordion&quot; value for the position style attribute of FOLDER containers, to render the children pages as a set of collapsible group boxes.</td>
<td>See Folder style attributes on page 1195, FOLDER item type on page 1275, Presentation styles changes on page 135.</td>
</tr>
<tr>
<td>A set of ready-to-use web components are now provided in the Genero BDL package.</td>
<td>See Built-in web components on page 1972, Web components changes on page 138.</td>
</tr>
</tbody>
</table>
Overview

- **gICAPI Web Component `onFlushData` and `onStateChange` methods** to better control content change and form field state.
- **New dialog methods** to set text, image and comment of default action views.
- **Defining array cell attributes** in dynamic dialogs with simple and two-dimensional dynamic arrays.
- **Display values** to form fields or screen arrays that are only known at runtime.
- **NOTEDITABLE** can be specified for `TEXTEDIT` form fields.
- **Use the `DateEdit` style attribute `calendarType`** to define the type of calendar.
- **GBC supports the `buttonIcon` style attribute** for `DateEdit` and `DateTimeEdit` classes.
- **The `UserInterface` style attribute `reverse`** can be used for Arabic languages support.
- **`localStorage` front calls** to store key/value pairs on the front-end side.
- **Style attribute `rowActionTrigger (Table class)`**, to define the physical event to fire the row choice action. This allows you to define simple-clicks to select a row in a table controlled by `DISPLAY ARRAY`.
- **`DOUBLECLICK` attribute now supported** for `SCROLLGRID` lists, in `.per file syntax` and as `DISPLAY ARRAY` attribute.
- **`ON CHANGE` is now allowed in `CONSTRUCT`**.
- **`SCREEN` can get a `TAG` attribute**.
- **`bold` attribute in table cell attributes**.
- **The `Window` style attribute `thinScrollbarDisplayTime`** accepts a negative value, to hide the scrollbar.
- **Form item attribute `INITIALPAGESIZE` for `SCROLLGRID` elements**.
- **The `standard.openFiles` front call** can be used to let the user select multiple files on the front-end platform.
- **The `Window` style attributes `actionPanelButtonSize` and `ringMenuButtonSize`** support the `em` unit ("20em").
- **`AGGREGATE` fields** can be used with any item type (was limited to `EDIT` columns).

Reference

- See **`gICAPI.onFlushData()` on page 1938**, **`gICAPI.onStateChanged()` on page 1944**.
- See **Configuring default action views dynamically** on page 1749.
- See **`ui.Dialog.setArrayAttributes` on page 2543**.
- See **`ui.Form.displayTo` on page 2500**.
- See **`NOTEDITABLE attribute` on page 1382**.
- See **`DateEdit style attributes` on page 1190**.
- See **`DateTimeEdit style attributes` on page 1191**.
- See **Right-to-left languages support** on page 534.
- See **`Local storage front calls` on page 2695**, **`New localStorage frontcalls` on page 149**.
- See **Defining the action for a row choice** on page 1867.
- See **`ON CHANGE block` on page 1492**.
- See **`SCREEN section` on page 1300**.
- See **`ui.Dialog.setArrayAttributes` on page 2543**.
- See **Window style attributes: Basics** on page 1220.
- See **`INITIALPAGESIZE attribute` on page 1374**.
- See **The `openFiles` front call**.
- See **Presentation styles changes** on page 135, **Window style attributes: Action Panel** on page 1228, **Window style attributes: Ring Menu** on page 1230.
- See **Summary lines in tables** on page 1866.
Overview | Reference
---|---
GDC auto-update front call (monitor.update) to install a new GDC automatically from a Genero program. | See `monitor.update` frontcall.
Input field hint specification with the `PLACEHOLDER` attribute. | See `PLACEHOLDER` attribute on page 1385.
Front call API allows `RECORD` and `ARRAY` specification in input and output parameters. | See `ui.Interface.frontCall` on page 583, `webcomponent.call`.
The `ui.Dialog.nextField()` method accepts "+NEXT", "+PREV" and "+CURR" field specification. | See `ui.Dialog.nextField` on page 2539.
The `ui.Dialog.cancel()` method can be used to cancel a parent `DIALOG` from the `SUBDIALOG` code. | See `ui.Dialog.cancel` on page 2527.
New theme front calls allow the Genero application to set the theme, retrieve the theme in use, or list all available themes for an application using the Genero Browser Client. | See `Theme front calls` on page 2691.
Better control of row / cell highlighting for `TABLE` and `SCROLLGRID` containers. | See `Row and cell highlighting in TABLE` on page 1208, `Row and cell highlighting in SCROLLGRID` on page 1204, `Row highlighting in TREE` on page 1212.
Set the `Table` style attribute `tableType` to `listView`, to get Mobile list view rendering for tables. | See `Controlling table rendering` on page 1858, `Table style attributes` on page 1204.
Set the `Scrollgrid` style attribute `customWidget` to `pagedScrollGrid`, to render the scrollgrid as a responsive tile list. | See `SCROLLGRID item type` on page 1285, `ScrollGrid style attributes` on page 1202, `Scrollgrid views` on page 1880.
Set the `Checkbox` style attribute `customWidget` to `toggleButton`, to render the checkbox as a toggle switch. | See `CHECKBOX item type` on page 1269, `CheckBox style attributes` on page 1187.
The `Button` style attribute `alignment` can be used to define the content position in the widget. | See `Button style attributes` on page 1185.
The `Window` style attribute `windowState` can be set to `fullscreen` to display the window in fullscreen mode (GDC only). | See `Window style attributes: Basics` on page 1220.
Upgrade notes for presentation styles. | See `Presentation styles changes` on page 135.
Upgrade notes for front calls. | See `Front calls changes` on page 138.
Upgrade notes for web components. | See `Web components changes` on page 138.

Table 18: SQL databases:

Overview | Reference
---|---
Support of Microsoft® ODBC v13 and v17 for SQL Server with dbmsnc_13 and dbmsnc_17 ODI drivers. These drivers are available on Windows® and Linux® 64bits platforms, to connect to SQL Server 2016, 2017 and Azure SQL Database. | See `Microsoft ODBC Driver for SQL Server` on page 141.
### Overview

<table>
<thead>
<tr>
<th>Support</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support for SAP HANA® 2.0 database with the new ODI driver <code>dbmhdb_2</code>.</td>
<td>See SAP HANA on page 1069.</td>
</tr>
<tr>
<td>Support for IBM® DB2® LUW 11.x (with <code>dbmdb2_10</code>).</td>
<td>See Database driver specification (driver) on page 662.</td>
</tr>
<tr>
<td>Support for Oracle® 12c on macOS®.</td>
<td>See Mac OS X configuration notes on page 44 (Special consideration must be taken for <code>libclntsh.so</code>).</td>
</tr>
<tr>
<td>Support for PostgreSQL 9.6 and 10.</td>
<td>See Database driver specification (driver) on page 662.</td>
</tr>
<tr>
<td>Support for Oracle® MySQL 5.7 (<code>dbmmys_5_7</code>) and MySQL 8.0 (<code>dbmmys_8_0</code>).</td>
<td>See Oracle MySQL 5.7 and 8.0 support on page 146, Database driver specification (driver) on page 662.</td>
</tr>
<tr>
<td>Support for MariaDB 10.2 with the new ODI driver <code>dbmmdb_10_2</code>.</td>
<td>See Database driver specification (driver) on page 662, Prepare the runtime environment - connecting to the database on page 877.</td>
</tr>
<tr>
<td>Support for FreeTDS 1.00 (to connect to SQL Server up to version 2016)</td>
<td>See Prepare the runtime environment - connecting to the database on page 828.</td>
</tr>
<tr>
<td>The SNC driver selects automatically the char mode (`snc.widechar FGLPROFILE parameter) depending on the application locale.</td>
<td>See Wide Char mode of SNC driver on page 149.</td>
</tr>
<tr>
<td>Special hints in C-style comments to control SQL statement execution.</td>
<td>See fglhint_* in SQL comments on page 147.</td>
</tr>
<tr>
<td>Defining ODBC connection string parameters for SQL Server with the &quot;datasource?options&quot; notation in the &quot;source&quot; connection parameter.</td>
<td>See Database source specification (source) on page 661, SQL connection identifier on page 678.</td>
</tr>
<tr>
<td>Specifying Oracle® DB schema and tablespace in temporary table emulations.</td>
<td>See Temp table emulation with Oracle DB on page 145.</td>
</tr>
<tr>
<td>Oracle® DB proxy authentication.</td>
<td>See Oracle DB Proxy Authentication on page 145.</td>
</tr>
<tr>
<td>Print SQL debug message only when SQL error occurs, and control SQL debug level from programs with <code>fgl_sqldebug()</code></td>
<td>See Debugging SQL statements on page 603, fgl_sqldebug() on page 2295.</td>
</tr>
<tr>
<td>Converting MATCHES to PostgreSQL SIMILAR TO expressions.</td>
<td>See MATCHES and LIKE on page 995.</td>
</tr>
<tr>
<td>Faster SQL execution with PostgreSQL driver and new pre-fetch row count parameter in FGLPROFILE.</td>
<td>See PostgreSQL specific FGLPROFILE parameters on page 676.</td>
</tr>
<tr>
<td>SET LOCK MODE support with PostgreSQL driver.</td>
<td>See Concurrency management on page 967.</td>
</tr>
<tr>
<td>Upgrade notes for database drivers.</td>
<td>See Database drivers changes on page 140.</td>
</tr>
</tbody>
</table>
### Table 19: Tools

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>GUI log file name with process id (--start-guilog=&quot;myfile-$p.log&quot;).</td>
<td>See Front-end protocol logging on page 1120.</td>
</tr>
<tr>
<td>Source code coverage tool (FGLCOV/fglrun --merge-cov).</td>
<td>See Source code coverage on page 2168.</td>
</tr>
<tr>
<td>Program execution trace (fglrun --trace).</td>
<td>See Execution trace on page 2171.</td>
</tr>
<tr>
<td>VIM plugins can be used directly from $FGLDIR/vimfiles.</td>
<td>See Configure VIM for Genero BDL on page 2123.</td>
</tr>
<tr>
<td>The fpi command line tool can be used to show Genero BDL version information.</td>
<td>See fpi on page 2095.</td>
</tr>
<tr>
<td>The fglgar command line tool is now delivered with the FGLGWS package.</td>
<td>See fglgar on page 2090, Packaging web applications on page 2183.</td>
</tr>
<tr>
<td>fglcomp option -W to-err-file to write warnings to the .err file.</td>
<td>See fglcomp on page 2071.</td>
</tr>
<tr>
<td>fglcomp option --tag=string to write a custom string to the .42m pcode module.</td>
<td>See fglcomp on page 2071, 42m module information on page 2120.</td>
</tr>
</tbody>
</table>

### Table 20: Web Services

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support of Message Transmission Optimization Mechanism (MTOM), a method of efficiently sending binary data to and from Web services.</td>
<td>See Message Transmission Optimization Mechanism (MTOM) on page 3297, XMLOptimizedContent on page 3552, and fglwsdl on page 2081 (-hexb64AsString option)</td>
</tr>
<tr>
<td>The swaRef feature optimizes file transfer when the files are attachments and located on disk.</td>
<td>See swaRef (SOAP with attachments using wsi:swaRef) on page 3298.</td>
</tr>
<tr>
<td>Better support of Web Services on iOS mobile devices.</td>
<td>See Web Services on mobile devices on page 3586.</td>
</tr>
<tr>
<td>New methods clearHeaders() and removeHeader() remove headers from an HTTP multipart object.</td>
<td>See com.HTTPPart methods on page 2903.</td>
</tr>
<tr>
<td>New XML serializer options xml.ignoreunknownelements and xml.ignoreunknownattributes, to ignore unexpected elements or attributes.</td>
<td>See Serialization option flags on page 3046.</td>
</tr>
<tr>
<td>The XMLElementNillable attribute provides an option to set all XML elements explicitly to null and to serialize them with xsi:nil=&quot;true&quot;, when the BDL variable is NULL.</td>
<td>See XMLElementNillable on page 3542, Web Services changes on page 129.</td>
</tr>
<tr>
<td>The XMLNillable attribute provides an option to set an XML node explicitly to null and to serialize it with xsi:nil=&quot;true&quot;, when the BDL variable is NULL but not optional.</td>
<td>See XMLNillable on page 3540, Web Services changes on page 129.</td>
</tr>
<tr>
<td>New built-in server-side methods for the com.HTTPServiceRequest class: getURLHost(), getURLPort(), getURLPath(), getURLQuery(), findRequestCookie(), and setResponseCookie().</td>
<td>See The HTTPServiceRequest class on page 2845.</td>
</tr>
</tbody>
</table>
### Table 21: Mobile apps (GMA 1.30 and GMI 1.30)

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>GMI and GMA include generic Cordova/PhoneGap plugin front calls to make use of the Cordova plugins.</td>
<td>See Cordova plugin front calls on page 2720.</td>
</tr>
<tr>
<td>GMI build tool options --install-plugins and --list-plugins, to manage Cordova plugins.</td>
<td>See gmibuildtool on page 2102 and Cordova plugins on page 3644.</td>
</tr>
<tr>
<td>GMA build tool option --build-cordova, to embed one or more Cordova plugins.</td>
<td>See gmabuildtool on page 2096 and Cordova plugins on page 3644.</td>
</tr>
<tr>
<td>Overview</td>
<td>Reference</td>
</tr>
<tr>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>GMA build tool command scaffold, to manage scaffold archives. This new command comes with the --list-plugins and --install-plugins options to handle plugins in the scaffold archive.</td>
<td>See gmabuildtool on page 2096 and Building Android apps with Genero on page 3591.</td>
</tr>
<tr>
<td>Stretchable SCROLLGRID lists are supported on mobile front-ends.</td>
<td>See SCROLLGRID item type on page 1285.</td>
</tr>
<tr>
<td>Predefined actions enterbackground and s enterforeground to detect app state change.</td>
<td>See Background/foreground modes on page 3573.</td>
</tr>
<tr>
<td>The GMIDIR environment variable allows you to install the GMI archive in a different directory than FGLDIR.</td>
<td>See Building iOS apps with Genero on page 3606.</td>
</tr>
<tr>
<td>GMI build tool option --extensions-libs, to specify the libraries to be used to build your iOS app.</td>
<td>See Building iOS apps with Genero on page 3606.</td>
</tr>
<tr>
<td>GMA build tool options --build-status-icon-* allow you to define the status notification bar icon.</td>
<td>See Building Android apps with Genero on page 3591, gmabuildtool on page 2096.</td>
</tr>
<tr>
<td>GMA build tool option --accept-licenses allows you to silently accept Android™ SDK licenses during updates of Android™ SDK.</td>
<td>See gmabuildtool on page 2096 and Building Android apps with Genero on page 3591.</td>
</tr>
<tr>
<td>GMA supports now the alignment style attribute for IMAGE.</td>
<td>See Presentation styles changes on page 135, Image style attributes on page 1198.</td>
</tr>
<tr>
<td>GMI build tool option --install provides a method for installing without the need to snoop with instruments for the actual device name.</td>
<td>See gmibuildtool on page 2102 and Genero mobile development client for iOS on page 3569.</td>
</tr>
<tr>
<td>The style attribute reduceFilter controls the display of the the listview reduce filter on mobile clients.</td>
<td>See Table style attributes on page 1204.</td>
</tr>
<tr>
<td>GMA build tool option --build-force-scaffold-update, to re-create the app project directory from a fresh GMA scaffold directory.</td>
<td>See gmabuildtool on page 2096 and Genero Mobile for Android (GMA) 1.30 changes on page 130.</td>
</tr>
<tr>
<td>GMA build tool option --build-quietly, to force yes for any answer asked during the build process.</td>
<td>See gmabuildtool on page 2096 and Genero Mobile for Android (GMA) 1.30 changes on page 130.</td>
</tr>
<tr>
<td>The front call mobile.isForeground indicates if the app is in foreground mode.</td>
<td>See Front calls changes on page 138, isForeground on page 2707.</td>
</tr>
</tbody>
</table>

**BDL 3.00 new features**

Features added in 3.00 releases of the Genero Business Development Language.

**Important:** Please read also:

- **BDL 2.51 new features** on page 66, for a list of features that were introduced with the Genero Mobile 1.0 release.

**Important:** This page covers only those new features introduced with the Genero BDL version specified in the page title. Check prior new features pages if you migrate from an earlier version. Make sure to also read the upgrade guide corresponding to this Genero version.

Corresponding upgrade guide: **BDL 3.00 upgrade guide** on page 151.
## Table 22: Core language

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attach the debugger to a running program with <code>fgldb -p process-id</code>.</td>
<td>See <a href="#">Attaching to a running program</a> on page 2143.</td>
</tr>
<tr>
<td>Improved compilation time (<code>fglcomp</code> and <code>fglform</code>)</td>
<td>See <a href="#">Improved compilation time</a> on page 159.</td>
</tr>
<tr>
<td>Date-related utility methods.</td>
<td>See <a href="#">util.Date methods</a> on page 2727.</td>
</tr>
<tr>
<td>Interval-related utility methods.</td>
<td>See <a href="#">util.Interval methods</a> on page 2750.</td>
</tr>
<tr>
<td>Temporary file name creation with <code>os.Path.makeTempName()</code>.</td>
<td>See <a href="#">os.Path.makeTempName</a> on page 2804.</td>
</tr>
<tr>
<td>JSON stringification method to omit NULL elements.</td>
<td>See <a href="#">util.JSON.stringifyOmitNulls</a> on page 2772.</td>
</tr>
<tr>
<td>New <code>fglcomp</code> warning for invalid NULL usage in expressions like <code>var==NULL</code>.</td>
<td>See <a href="#">Compiler warning -6636</a></td>
</tr>
<tr>
<td><code>fglcomp</code> option to avoid source name in the <code>.42m</code> module.</td>
<td>See <a href="#">42m module information</a> on page 2120.</td>
</tr>
<tr>
<td>C Extension runtime stack introspection (parameter type and actual string value size in bytes).</td>
<td>See <a href="#">Runtime stack functions</a> on page 2235.</td>
</tr>
<tr>
<td>The <code>fglmkext</code> command line tool can build your C Extension library.</td>
<td>See <a href="#">fglmkext</a> on page 2077.</td>
</tr>
</tbody>
</table>

## Table 23: User interface

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic dialog creation (equivalent of INPUT, CONSTRUCT and DISPLAY ARRAY blocks).</td>
<td>See <a href="#">Dynamic Dialogs</a> on page 2004, <a href="#">ui.Dialog.createConstructByName</a> on page 2515, <a href="#">ui.Dialog.createInputByName</a> on page 2517, <a href="#">ui.Dialog.createDisplayArrayTo</a> on page 2515.</td>
</tr>
<tr>
<td>Resizable SCROLLGRID containers (WANTFIXEDPAGESIZE=NO).</td>
<td>See <a href="#">WANTFIXEDPAGESIZE attribute</a> on page 1407.</td>
</tr>
<tr>
<td>The <code>ON SORT</code> dialog control block can be used to execute code when the record list is re-ordered by the user.</td>
<td>See <a href="#">List ordering</a> on page 1862, <a href="#">Populating a DISPLAY ARRAY</a> on page 1841, <a href="#">ON SORT block</a> on page 1520, <a href="#">ui.Dialog.getSortKey</a> on page 2534, <a href="#">ui.Dialog.isSortReverse</a> on page 2537.</td>
</tr>
<tr>
<td><code>ON TIMER</code> trigger in dialogs, to execute a block of code at regular intervals.</td>
<td>See <a href="#">Get program control on a regular (timed) basis</a> on page 1713.</td>
</tr>
<tr>
<td>Overview</td>
<td>Reference</td>
</tr>
<tr>
<td>-------------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Autocompletion in text edit fields with the COMPLETER attribute.</td>
<td>See <a href="#">Enabling autocompletion on page 1733</a>.</td>
</tr>
<tr>
<td>Centralization of icon definitions with the FGLIMAGEPATH environment</td>
<td>See <a href="#">Providing the image resource on page 1149</a>, <a href="#">FGLIMAGEPATH on page 276</a>, <a href="#">Built-in front-end icons desupport on page 158</a>.</td>
</tr>
<tr>
<td>variable.</td>
<td></td>
</tr>
<tr>
<td>Binding structured ARRAYs in DISPLAY ARRAY and INPUT ARRAY.</td>
<td>See <a href="#">Structured ARRAYs in list dialogs on page 162</a>.</td>
</tr>
<tr>
<td>Defining an action for IMAGE form items (clickable images).</td>
<td>See <a href="#">Defining action views in forms on page 1759</a>, <a href="#">Image columns firing actions on page 1861</a>, <a href="#">IMAGE item type on page 1279</a>.</td>
</tr>
<tr>
<td>Detect window resizing or device orientation change with the windowresized predefined action.</td>
<td>See <a href="#">Adapting to viewport changes on page 1417</a>.</td>
</tr>
<tr>
<td>Dialog methods to convert the program array row index to the visual index, and the opposite.</td>
<td>See <a href="#">ui.Dialog.arrayToVisualIndex on page 2527</a>, <a href="#">ui.Dialog.visualToArrayIndex on page 2553</a>.</td>
</tr>
<tr>
<td>Providing application image resources to Web Components with</td>
<td>See <a href="#">Using image resources with the gICAPI web component on page 1952</a>, <a href="#">ui.Interface.filenameToURI on page 2475</a>.</td>
</tr>
<tr>
<td>ui.Interface.filenameToURI().</td>
<td></td>
</tr>
<tr>
<td>The standard.openFile front call is now supported with GBC.</td>
<td>See <a href="#">standard front call support matrix</a>.</td>
</tr>
<tr>
<td>The dictionariesDirectory parameter for the standard.feInfo front call can be used to get the directory where spell checker dictionary files can be uploaded.</td>
<td>See <a href="#">standard.feInfo front call</a>.</td>
</tr>
<tr>
<td>The allowWebSelection style attribute can used to enable items selection with a simple mouse drag.</td>
<td>See <a href="#">Table style attributes on page 1204</a>.</td>
</tr>
<tr>
<td>The browserMultiPage style can be used to specify whether the RUN and RUN WITHOUT WAITING instructions will be executed in the same browser tab or in a new browser tab.</td>
<td>See <a href="#">UserInterface style attributes on page 1219</a>.</td>
</tr>
<tr>
<td>Upgrade notes for presentations styles.</td>
<td>See <a href="#">Presentation styles changes on page 158</a>.</td>
</tr>
<tr>
<td>Upgrade notes for front calls.</td>
<td>See <a href="#">Modifications in front calls on page 159</a>.</td>
</tr>
</tbody>
</table>

### Table 24: SQL databases

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support for PostgreSQL 9.4.</td>
<td>See <a href="#">Database driver specification (driver) on page 662</a>.</td>
</tr>
<tr>
<td>Support for SAP® ASE 16.x (Warning: SAP ASE is desupported in BDL version 4.00).</td>
<td>See <a href="#">Database driver specification (driver) on page 662</a>.</td>
</tr>
</tbody>
</table>
### Overview

| Maria DB support (V5.5 and V10): Use the dbmmys driver. | See MariaDB support on page 157. |
| Support for Microsoft® SQL Server 2016 with SNC 11, ESM and FTM drivers. | See Microsoft SQL Server on page 827. |
| Dynamic cursor built-in class base.SqlHandle. | See The SqlHandle class on page 2432. |
| SQL interruption is now supported with Oracle® MySQL. | See Using SQL interruption on page 601. |
| MySQL VARCHAR(N) can be used when N is greater as 255. | See MySQL VARCHAR size limit on page 155. |
| MySQL DATETIME can store fractional seconds. | See MySQL DATETIME fractional seconds on page 156. |
| Native Oracle® NUMBER type (without precision/scale) can be extracted by fglbsch. | See Oracle DB NUMBER type on page 154. |
| Serial emulation based on triggers and sequences with SQL Server 2012 and +. | See SERIAL and BIGSERIAL data types on page 847. |
| PostgreSQL connection string option specification in the source parameter. | See Database source specification (source) on page 661, Prepare the runtime environment - connecting to the database on page 965. |
| Upgrade notes for database drivers. | See Database drivers changes on page 154. |

### Table 25: Web Services

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flushing immediately the response of a web service operation with com.WebServicesEngine.flush.</td>
<td></td>
</tr>
<tr>
<td>Base64 / Hexadecimal / Digest methods using a specific character set for string data.</td>
<td></td>
</tr>
</tbody>
</table>
Specific APIs for Apple® Push Notification Service support.

Methods to perform RESTful requests using files on disk.

FGLPROFILE entries to define XML Signature and XML Encrypted data prefix: xml.signature.prefix and xml.encrypted.prefix.

SOAP fault handling works now when HTTP error 200 is returned by the server.

Client stub multipart supports now optional parts.

Upgrade notes for web services.

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>FGLPROFILE entries to define XML Signature and XML Encrypted data prefix: xml.signature.prefix and xml.encrypted.prefix.</td>
<td>See XML configuration on page 3494.</td>
</tr>
<tr>
<td>SOAP fault handling works now when HTTP error 200 is returned by the server.</td>
<td>See SOAP fault handling in client stub on page 152.</td>
</tr>
<tr>
<td>Client stub multipart supports now optional parts.</td>
<td>See Optional multipart handling in client stub on page 153.</td>
</tr>
<tr>
<td>Upgrade notes for web services.</td>
<td>See Web Services changes on page 151.</td>
</tr>
</tbody>
</table>

Table 26: Mobile apps

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command line tools to build mobile apps.</td>
<td>See Building Android apps with Genero on page 3591, Building iOS apps with Genero on page 3606.</td>
</tr>
<tr>
<td>Starting remote applications from a mobile device with the runOnServer front call.</td>
<td>See Running mobile apps on an application server on page 3616.</td>
</tr>
<tr>
<td>Push notification APIs for Google Cloud Messaging (GMA) and Apple Push Notification Service (GMI), with new predefined actions (notificationpushed).</td>
<td>See Push notifications on page 3619.</td>
</tr>
<tr>
<td>Extended feInfo front call options for mobile devices (deviceModel, deviceId, freeStorageSpace, iccid, imei, ppi, windowSize, and so on).</td>
<td>See feInfo on page 2651.</td>
</tr>
<tr>
<td>New materialFABType and materialFABActionList style attributes for Window class, to control the FAB button on devices following material design guidelines.</td>
<td>See Android floating action button on page 1787.</td>
</tr>
<tr>
<td>Front call to display a box controlling debug settings on GMA.</td>
<td>See showSettings (Android) on page 2715.</td>
</tr>
<tr>
<td>Overview</td>
<td>Reference</td>
</tr>
<tr>
<td>-------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Automatic FGLAPPDIR environment variable (defining the path to the appdir), and automatic FGLDIR environment variable, when executing on mobile devices.</td>
<td>See FGLAPPDIR on page 273, FGLDIR on page 275, Setting environment variables in FGLPROFILE (mobile) on page 263.</td>
</tr>
<tr>
<td>Front calls to take or choose videos on mobile devices.</td>
<td>See chooseVideo on page 2700, takeVideo on page 2712 front calls.</td>
</tr>
<tr>
<td>Front call to ask user for Android™ permissions.</td>
<td>See askForPermission (Android) on page 2714 front call.</td>
</tr>
<tr>
<td>GMA buildtool --clean option to cleanup the scaffold directory in case of interruption or failure in prior build.</td>
<td>See Building Android apps with Genero on page 3591.</td>
</tr>
<tr>
<td>GMA buildtool --no-install-extras option to avoid installation of extras during Android™ SDK update.</td>
<td>See Building Android apps with Genero on page 3591.</td>
</tr>
<tr>
<td>GMI specific style attribute iosTabBarUnselectedColor, to define the color of unselected tab bar elements.</td>
<td>See Navigator pane on page 1904.</td>
</tr>
<tr>
<td>GMA specific style attribute androidKeepForeground, to control the app state and the background state notification.</td>
<td>See Mobile app foreground and background modes on page 489, UserInterface style attributes on page 1219.</td>
</tr>
</tbody>
</table>

Table 27: Experimental features

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stacked form definition in .per files with the new STACK container, for mobile programming.</td>
<td>See Stack-based layout on page 1431, STACK container on page 1309.</td>
</tr>
</tbody>
</table>

**BDL 2.51 new features**

Features added in 2.51 releases of the Genero Business Development Language.

**Important:** This page covers only those new features introduced with the Genero BDL version specified in the page title. Check prior new features pages if you migrate from an earlier version. Make sure to also read the upgrade guide corresponding to this Genero version.

Corresponding upgrade guide: BDL 2.51 upgrade guide on page 162.

**Important:** Most of the new features of BDL 2.51 have been added for Genero Mobile. The features designed for Genero Mobile may not be supported by desktop and web-browser front-ends in the coming releases.

**Genero Mobile V 1.0 (FGL 2.51.06)**

**Table 28: Core language**

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remote debugging through network TCP socket</td>
<td>See Debugging on a mobile device on page 2145.</td>
</tr>
</tbody>
</table>
Table 29: Core language (mobile apps)

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>The <code>sort()</code> method of ARRAY variables.</td>
<td>See <code>DYNAMIC ARRAY.sort</code> on page 2399.</td>
</tr>
<tr>
<td>Datetime-related utility methods.</td>
<td></td>
</tr>
<tr>
<td>String-related utility methods.</td>
<td></td>
</tr>
<tr>
<td>Write to stdout with <code>om.XmlWriter.createFileWriter(NULL)</code>.</td>
<td>See <code>om.XmlWriter.createFileWriter</code> on page 2632.</td>
</tr>
</tbody>
</table>

Table 30: User interface

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>URL-based Web Components</td>
<td>See <code>Using a URL-based web component</code> on page 1928.</td>
</tr>
<tr>
<td>The <code>DATETIMEEDIT</code> form item type</td>
<td>See <code>DATETIMEEDIT item type</code> on page 1273.</td>
</tr>
<tr>
<td>Dialog-level action attribute definitions with <code>ON ACTION name ATTRIBUTES()</code></td>
<td>See <code>Configuring actions</code> on page 1744.</td>
</tr>
<tr>
<td>New <code>ON SELECTION CHANGE</code> control block.</td>
<td>See <code>Multiple row selection</code> on page 1851.</td>
</tr>
</tbody>
</table>

Table 31: User interface (mobile apps)

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>START DIALOG/TERMINATE DIALOG/fgl_eventLoop()</code></td>
<td>See <code>Understanding parallel dialogs</code> on page 1700.</td>
</tr>
<tr>
<td>Window <code>TYPE</code> attribute in <code>OPEN  WINDOW</code> instruction.</td>
<td>See <code>Window TYPE attribute</code> on page 1134.</td>
</tr>
<tr>
<td><code>DISPLAY ARRAY</code> attributes for list views handling: ACCESSORYTYPE, DETAILACTION, DOUBLECLICK.</td>
<td>See <code>Using tables on mobile devices</code> on page 1867.</td>
</tr>
<tr>
<td>The <code>DISCLOSUREINDICATOR</code> action attribute.</td>
<td>See <code>DISCLOSUREINDICATOR action attribute</code> on page 1756.</td>
</tr>
</tbody>
</table>
Overview | Reference
--- | ---
The ROWBOUND action attribute. | See ROWBOUND action attribute on page 1757.
The KEYBOARDHINT form field attribute. | See KEYBOARDHINT attribute on page 1378.
List filter with DISPLAY ARRAY dialog. | See Reduce filter on page 1865.
Method `ui.Interface.getFrontEndName()` can now return GMI or GMA | See `ui.Interface.getFrontEndName` on page 2477.
Front-end functions for Genero Mobile (GMA / GMI) | See Genero Mobile common front calls on page 2697, Genero Mobile Android front calls on page 2713, Genero Mobile iOS front calls on page 2718.
Navigation bar button colors and background colors for iOS device (iosTintColor, iosNavigationBarTintColor, iosToolBarTintColor, iosTabBarTintColor) - provided as Window class style attributes. | See Window style attributes: GMI/iOS on page 1232.

Table 32: SQL databases

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simplified database driver specification</td>
<td>See New database driver name specification on page 163.</td>
</tr>
<tr>
<td>Support for SQL Server 2014</td>
<td>See Database driver specification (driver) on page 662.</td>
</tr>
<tr>
<td>Support for Oracle Database 12c</td>
<td>See Database driver specification (driver) on page 662.</td>
</tr>
<tr>
<td>Support for PostgreSQL 9.3</td>
<td>See Database driver specification (driver) on page 662.</td>
</tr>
<tr>
<td>Better support for DATETIME types with SQLite</td>
<td>See DATETIME types with SQLite on page 164.</td>
</tr>
<tr>
<td>STRING typed variables can be used in SQL statements.</td>
<td>See STRING on page 305.</td>
</tr>
<tr>
<td>Upgrade notes for database drivers</td>
<td>See Database drivers changes on page 162.</td>
</tr>
</tbody>
</table>

Genero Mobile V 1.1 (FGL 2.51.07)

Table 33: Core language (mobile apps)

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementing C-Extensions on iOS / GMI.</td>
<td>See Implementing C-Extensions for GMI on page 2247.</td>
</tr>
<tr>
<td>Using Java interface for Android / GMA.</td>
<td>See Executing Java code with GMA on page 2219.</td>
</tr>
<tr>
<td>Implementing customer front calls for GMA.</td>
<td>See Implement front call modules for GMA on page 2254.</td>
</tr>
</tbody>
</table>
### Table 34: Web Services (mobile apps)

<table>
<thead>
<tr>
<th>Overview</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete support of Web Services on Android mobile devices.</td>
</tr>
<tr>
<td>(Web Services are partly supported on iOS mobile devices)</td>
</tr>
<tr>
<td>Reference</td>
</tr>
<tr>
<td>See Web services on page 3270.</td>
</tr>
</tbody>
</table>

### BDL 2.50 new features

Features added in 2.50 releases of the Genero Business Development Language.

**Important**: This page covers only those new features introduced with the Genero BDL version specified in the page title. Check prior new features pages if you migrate from an earlier version. Make sure to also read the upgrade guide corresponding to this Genero version.

Corresponding upgrade guide: BDL 2.50 upgrade guide on page 167.

### Table 35: User interface (mobile apps)

<table>
<thead>
<tr>
<th>Overview</th>
</tr>
</thead>
<tbody>
<tr>
<td>Presentation styles are now supported by mobile front-ends.</td>
</tr>
<tr>
<td>GMA bundles zxing for Android.</td>
</tr>
<tr>
<td>Reference</td>
</tr>
<tr>
<td>See Style attributes reference on page 1182.</td>
</tr>
</tbody>
</table>

### Table 36: Core language

<table>
<thead>
<tr>
<th>Overview</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support for character length semantics to simplify UTF-8 programming.</td>
</tr>
<tr>
<td>The UTF-8 character set can be used on Microsoft™ Windows® platforms by</td>
</tr>
<tr>
<td>setting the LANG environment variable to .fglutf8.</td>
</tr>
<tr>
<td>JSON (JavaScript Object Notation) utility classes.</td>
</tr>
<tr>
<td>String to DATETIME conversion now accepts ISO 8601 format sub-set.</td>
</tr>
<tr>
<td>The base.Channel method dataAvailable(), to check for channel</td>
</tr>
<tr>
<td>readability.</td>
</tr>
<tr>
<td>With IMPORT FGL, fglcomp now automatically compiles imported modules</td>
</tr>
<tr>
<td>when needed. To avoid implicit compilation, use the --</td>
</tr>
<tr>
<td>implicit=none option of fglcomp.</td>
</tr>
<tr>
<td>The --resolve-calls or -W implicit fglcomp compiler options</td>
</tr>
<tr>
<td>can be used to detected unresolved symbols.</td>
</tr>
<tr>
<td>Reference</td>
</tr>
<tr>
<td>See Length semantics settings on page 521.</td>
</tr>
<tr>
<td>See Language and character set settings on page 519.</td>
</tr>
<tr>
<td>See The util.JSON class on page 2766, The util.JSONObject class on page</td>
</tr>
<tr>
<td>2773, The util.JSONArray class on page 2783.</td>
</tr>
<tr>
<td>See IMPORT FGL module on page 481.</td>
</tr>
<tr>
<td>See IMPORT FGL module on page 481.</td>
</tr>
</tbody>
</table>
The fglrun option --print-imports can be used to find modules dependencies and use IMPORT FGL instead of traditional linking.

Option --doc-private can be used with fglcomp --build-doc in order to document private symbols.

CANCEL DIALOG instruction for DIALOG blocks.

FGLPROFILE fglrun.decToCharScale2 and fglrun.decToCharScale2.print to control DECIMAL(P) to string conversion.

---

**Table 37: User interface**

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dialog modularization. Declarative DIALOG blocks can be defined as module elements and reused with the SUBDIALOG keyword of procedural DIALOG blocks.</td>
<td>See Structure of a procedural DIALOG block on page 1594.</td>
</tr>
<tr>
<td>Form modularization. Use the new FORM layout keyword to include a subform in the current form specification file.</td>
<td>See Form file structure on page 1294.</td>
</tr>
<tr>
<td>CLEAR SCREEN ARRAY instruction clears the values of all the rows of a form list (TABLE, TREE, SCROLLGRID).</td>
<td>See CLEAR SCREEN ARRAY.</td>
</tr>
<tr>
<td>AUTONEXT attribute is allowed in DATEEDIT, SPINEDIT and TIMEEDIT fields.</td>
<td>See DATEEDIT, SPINEDIT, TIMEEDIT.</td>
</tr>
<tr>
<td>BUTTONEDIT item type attribute NOTEDITABLE, to disable the field editor.</td>
<td>See NOTEDITABLE attribute on page 1382.</td>
</tr>
<tr>
<td>ON CHANGE fired when selecting a date in DATEEDIT calendar or when changing the value of a TIMEEDIT widget.</td>
<td>See ON CHANGE block on page 1492.</td>
</tr>
<tr>
<td>Presentation style attributes ringMenuButtonTextHidden and actionPanelButtonTextHidden added to customize the default action view panels.</td>
<td>See actionPanelButtonTextHidden, ringMenuButtonTextHidden.</td>
</tr>
<tr>
<td>Presentation style attribute thinScrollbarDisplayTime to define the display time of the thin scrollbar when scrolling in fixed screen arrays.</td>
<td>See thinScrollbarDisplayTime.</td>
</tr>
<tr>
<td>Presentation style attribute customWidget:</td>
<td></td>
</tr>
<tr>
<td>• Defines the CHECKBOX widget (attribute is removed in V3.00)</td>
<td></td>
</tr>
<tr>
<td>• Defines the RichText TEXTEDIT widget (attribute is removed in V3.10)</td>
<td></td>
</tr>
<tr>
<td>fglrun options --start-guilog and --run-guilog, to generate and replay a GUI protocol exchange.</td>
<td>See Front-end protocol logging on page 1120.</td>
</tr>
<tr>
<td>Upgrade notes for presentation styles.</td>
<td>See Presentation styles changes on page 169.</td>
</tr>
</tbody>
</table>

---

**Table 38: SQL databases**

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>The SQLite driver dbmsqt 3xx is now statically linked with the SQLite library, except on platforms where the SQLite library is usually present such as Linux® and Mac OS-X™.</td>
<td>See the SQLite adaptation guide.</td>
</tr>
</tbody>
</table>
Overview | Reference
--- | ---
Database driver for PostgreSQL 9.2: dbmpgs92x. This driver is similar to the prior PGS 9.x drivers, it is supported for strict binary compatibility with the PostgreSQL 9.2 client library and is compiled with the 9.2 libpq headers. | See Database driver dbmpgs92x.

Database driver for IBM® DB2® LUW version 10: dbmdb2Ax. This driver is similar to the prior DB2® 9.x driver, it is supported for strict binary compatibility with the DB2® 10.x client library and is compiled with the 10.x CLI headers. | See Database driver dbmdb2Ax.

Support for the Oracle® RAW data type, in order to use the SYS_GUID() values generator. | See The RAW data type on page 936.

FGLPROFILE entry for MySQL to specify the my.cnf client configuration file: dbi.database.dbname.mys.config. | See Oracle MySQL specific FGLPROFILE parameters on page 675.

Upgrade notes for database drivers. | See Database drivers changes on page 169.

Table 39: Web Services

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>New security library provides classes and methods to support basic cryptographic features. Although added for Genero Web Services, can be used for any Genero application.</td>
<td>See The security package on page 3127.</td>
</tr>
<tr>
<td>New signature methods in xml.Signature class: signString() and verifyString().</td>
<td>See xml.Signature methods on page 3076.</td>
</tr>
<tr>
<td>Support of Diffie-Hellman key-agreement algorithm. It allows two peers to agree on the same symmetric key, the shared secret, without exchanging confidential data.</td>
<td>See The Diffie-Hellman key agreement algorithm on page 3318, Supported kind of keys on page 3061 and Computing the shared secret with Diffie-Hellman on page 3068.</td>
</tr>
<tr>
<td>HTTP compression support has been added for Genero Web Services.</td>
<td>See HTTP compression in SOAP on page 3295.</td>
</tr>
<tr>
<td>The com.HTTPRequest.setAutoReply() method now works for HTTP HEAD method as well as the GET method.</td>
<td>See com.HTTPRequest methods on page 2871.</td>
</tr>
</tbody>
</table>
### Overview

**DOM features:**
The Genero XML DOM library has been enhanced with new **features** that can be set with the `setFeature()` method or retrieved with the `getFeature()` method.

- **load-save-base64-string** - loads and saves an XML document from/to a base64 string
- **auto-id-attribute** - sets at document loading all unqualified attributes named ID, id, Id or iD of type ID
- **auto-id-qualified-attribute** - sets at document loading all qualified attributes named ID, id, Id or iD of type ID
- **enable-html-compliancy** - allows HTML document parsing and modification using the `xml.DomDocument` API.

**Binary support on HTTP layer:**
The Genero COM library has been enhanced to support transport of binary data via the Genero BYTE data type.

On the client side, it is now possible to send and read binary data to/from a server with the following two methods:

- **doRequest()** - sends binary data from a BYTE to a HTTP server
- **getDataResponse()** - reads binary data from a HTTP server into a BYTE

On the server side, it is possible to read and write binary data to a client with following two methods:

- **readDataRequest()** - reads binary data from a HTTP client into a BYTE
- **sendDataResponse()** - sends binary data from a BYTE to a HTTP client

Access the HTTP headers request and response in high level web services.

The standard API is enhanced with few new methods and a new class called `HTTPPart` to handle the different part in a HTTP request or response at client and server side.

The client side is able to generate stubs to support multiple part with Genero Web Services. Support for the server side is not yet provided.

**Note:** Starting with version 2.50.25, when generating client stubs managing multipart, you will get extra input and/or output variables called `AnyInputParts` and `AnyOutputParts`, defined as a `DYNAMIC ARRAY` of `com.HTTPPart` objects. These arrays may contain additional input and/or output HTTP parts not specified in the WSDL. You will have to adapt your client program, to handle those dynamic arrays in any functions calling such stubs. See [Client stubs managing multipart changes](page 171).

### Reference

See [The DomDocument class](page 2931).

See [The com package](page 2811).

See how to modify your server or use `fglwssdl` generated [global end point](page 171).

See [The HTTPPart class](page 2902), [The HTTPRequest class](page 2871), [The HTTPResponse class](page 2894), [The HTTPServiceRequest class](page 2845).

See [Multipart in the client stub](page 3332), [SOAP multipart style requests in GWS](page 3297).
### Overview

FGLPROFILE HTTPS configuration details no longer needed to perform HTTPS communication. A default SSL/TLS configuration is now generated automatically.

Creating URL base that applies to multiple server applications by using a wildcard in the URL, allowing for a shared server configuration (such as authentication and HTTPS).

Upgrade notes for web services.

<table>
<thead>
<tr>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>See HTTPS configuration on page 3305.</td>
</tr>
<tr>
<td>See FGLPROFILE: Server URL patterns on page 3498.</td>
</tr>
<tr>
<td>See Web Services changes on page 168.</td>
</tr>
</tbody>
</table>

### BDL 2.41 new features

Features added in 2.41 releases of the Genero Business Development Language.

**Important:** This page covers only those new features introduced with the Genero BDL version specified in the page title. Check prior new features pages if you migrate from an earlier version. Make sure to also read the upgrade guide corresponding to this Genero version.

Corresponding upgrade guide: BDL 2.41 upgrade guide on page 172.

#### Table 40: User interface

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>The <code>dataTypeHint</code> style attribute (for Edit item types) and <code>nativeLook</code> style attribute (for CheckBox item types) have been added for use by the Genero web client.</td>
<td>See Edit style attributes on page 1194 and CheckBox style attributes on page 1187.</td>
</tr>
<tr>
<td><strong>Important:</strong> In 2.50, the <code>nativeLook</code> attribute is renamed <code>customWidget</code>.</td>
<td></td>
</tr>
</tbody>
</table>

### BDL 2.40 new features

Features added in 2.40 releases of the Genero Business Development Language.

**Important:** This page covers only those new features introduced with the Genero BDL version specified in the page title. Check prior new features pages if you migrate from an earlier version. Make sure to also read the upgrade guide corresponding to this Genero version.

Corresponding upgrade guide: BDL 2.40 upgrade guide on page 172.

#### Table 41: Core language

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>The <code>NVL()</code> operator allows you to write the equivalent of an <code>IF expr IS NOT NULL THEN RETURN expr ELSE RETURN default END</code> statement in a single scalar expression.</td>
<td>See NVL() on page 347.</td>
</tr>
<tr>
<td>The <code>IIF()</code> allows you to write the equivalent of an <code>IF bool-expr THEN RETURN true-value ELSE RETURN false-value END</code> statement in a single scalar expression.</td>
<td>See IIF() on page 347.</td>
</tr>
<tr>
<td>A new global program option has been added, <code>OPTIONS SHORTCIRCUIT</code>, to instruct the runtime system to evaluate Boolean expressions by using the <code>short-circuit evaluation</code> (also called <code>minimal evaluation</code>) method.</td>
<td>See OPTIONS (Compilation) on page 557 and Controlling semantics of AND / OR operators on page 558.</td>
</tr>
</tbody>
</table>
### Table 42: User interface

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>New ON INSERT, ON APPEND, ON UPDATE and ON DELETE interaction blocks are now allowed in DISPLAY ARRAY dialogs to implement list modification, as an alternative to the traditional INPUT ARRAY dialog. These new triggers simplify the programming of modifiable record lists.</td>
<td>See DISPLAY ARRAY modification triggers on page 1850.</td>
</tr>
<tr>
<td>The new find and findnext actions of DISPLAY ARRAY and INPUT ARRAY can be used by the user to search rows where a field value matches the value entered in the find dialog box.</td>
<td>See Find function on page 1863.</td>
</tr>
<tr>
<td>The DISPLAY ARRAY dialog now supports a built-in seek feature to quickly find rows where a field value starts with the character typed by the user.</td>
<td>See Keyboard seek on page 1864.</td>
</tr>
<tr>
<td>Define a summary line for TABLEs by using AGGREGATE form fields. Values can be automatically computed or can be calculated and displayed by program.</td>
<td>See AGGREGATE item definition on page 1332.</td>
</tr>
<tr>
<td>Using the terminfo database for text terminal mode (FGLGUI=0) by setting INFORMIXTERM=terminfo.</td>
<td>See Using a text terminal on page 1123.</td>
</tr>
</tbody>
</table>

### Table 43: SQL databases

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>New database drivers are provided.</td>
<td>List of new database drivers:</td>
</tr>
<tr>
<td>• dbmntz6x for IBM® Netezza® ODBC client</td>
<td>• dbmsncB0 for SQL Server 2012 Native client</td>
</tr>
<tr>
<td>• dbmesmB0 for SQL Server 2012, with Easysoft ODBC driver</td>
<td>• dbmpgs91x for PostgreSQL 9.1.x client</td>
</tr>
<tr>
<td>The fglcomp compiler now supports SQL ... END SQL blocks for compliance with IBM® Informix® 4GL.</td>
<td>See SQL ... END SQL on page 702.</td>
</tr>
<tr>
<td>The Static SQL syntax has been extended to allow the FIRST, LIMIT, SKIP and MIDDLE SELECT projection clause options.</td>
<td>See Static SQL statements on page 694.</td>
</tr>
<tr>
<td>The CASE operator is now allowed in Static SQL statements.</td>
<td>See Static SQL statements on page 694.</td>
</tr>
<tr>
<td>The syntax of DDL (Data Definition Language) statements in Static SQL now allows the IF NOT EXISTS and IF EXISTS clauses.</td>
<td>See Static SQL statements on page 694.</td>
</tr>
<tr>
<td>The transaction instruction set has been completed with SAVEPOINT and ROLLBACK WORK TO SAVEPOINT.</td>
<td>See SAVEPOINT on page 690, ROLLBACK WORK on page 691.</td>
</tr>
<tr>
<td>Control shadow column extraction with fgldbsch.</td>
<td>See fgldbsch on page 2079.</td>
</tr>
<tr>
<td>A new FGLPROFILE entry parameter has been added to control the ORACLE DATE fetch into CHAR/VARCHAR variables.</td>
<td>See DATE and DATETIME data types on page 927.</td>
</tr>
</tbody>
</table>
Support for the ROWVERSION data type of SQL Server (2008 and +) has been added.

Table 44: Web Services

The Genero Web Service engine has been enhanced to support a part of SOAP 1.2 protocol, restricted to the SOAP POST feature only. It does not support the SOAP 1.2 encoding feature, as it is prohibited by the WS-I Basic Profile 2.0.

To allow the SOAP 1.2 protocol in your Genero Web service application, call the setFeature() method of your web service to enable SOAP 1.2 support.

The same Web service can provide both the SOAP 1.1 and SOAP 1.2 protocol.

You can also specify the SOAP role of your Genero application if you pass the new SoapModuleURI option to the WebServiceEngine setOption() method in order to identify the headers the SOAP engine has to understand.

The Genero Web Service engine has been enhanced to support the WS-Addressing 1.0 specification. To enable WS-Addressing 1.0 specification in your Genero Web service application, call the setFeature() method of your web service with "TRUE" or "REQUIRED" as a parameter.

There are two kinds of stateful services:

- Based on WS-Addressing: independent from the transport protocol used to convey the state between the client and the server.
- Based on HTTP cookies: depends on the transport protocol to convey the state between the client and the server.

To create a stateful web service, call com.WebService createStatefulWebService() with a simple BDL variable or a dedicated W3CEndpointReference record to handle the service state.

You can also take a look at WS-Addressing and at the following links for additional information: JAX-WS, Oracle and Stateful based on cookies.
Overview

The Genero Web Service engine has been enhanced to support SOAP faults in RPC and Document style services.

On the server side, you can define BDL variables that will be thrown as SOAP faults to a web service client using the SOAP 1.1 or SOAP 1.2 protocol.

The fglwsdl tool has also been enhanced to generate client and server stubs according to the SOAP fault described in the WSDL.

- Method `createFault()`
- Method `addFault()`
- Method `SetFaultDetail()`
- Tool `fglwsdl`

The Genero fglwsdl tool generates a new Endpoint record per service in the client stub to configure the client behavior at runtime without the need to modify the generated code.

This feature requires regeneration of the client stub and modification of the server location assignment if used in your application (See also Web Services changes on page 172).

The Genero fglwsdl tool has been enhanced to support WS-Addressing 1.0, the SOAP 1.2 protocol and to handle operation faults in SOAP 1.1 and SOAP 1.2.

The generated client and server stub will handle WS-Addressing 1.0, SOAP 1.2 protocol and manage soap faults as defined in the WSDL.

The following options have been added:

Options related to SOAP:
- `-soap11` : Generate only client and server stubs supporting the SOAP 1.1 protocol.
- `-soap12` : Generate only client and server stubs supporting the SOAP 1.2 protocol.
- `-ignoreFaults` : Do not generate soap faults.

Options related to WS-Addressing:
- `-wsa <yes|no>` : Force support of WS-Addressing 1.0. if yes, disable support of WS-Addressing 1.0, if no, otherwise support WS-Addressing 1.0 according to the definition in the WSDL.

Other options:
- `-alias` : Generate FGLPROFILE Logical names in place of URLs for all client stubs.
- `-extDir` : Add all schema files located in a directory and ending with .xsd as external schemas.
- `-CA` : Validate HTTPS certificate against a certificate authority list.

<table>
<thead>
<tr>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>See Change WS client behavior at runtime on page 3324.</td>
</tr>
<tr>
<td>See fglwsdl on page 2081.</td>
</tr>
</tbody>
</table>
### Overview

The XML-Signature and XML Encryption API of the XML library have been enhanced with new built-in methods to ease compatibility with the WS-Security specification:

- Method `getSignatureMethod()`
- Method `getThumbprintSHA1()`
- Method `getSHA1()`

The XML library has been enhanced to support XML parsing from PIPE and saving to PIPE:

- Method `loadFromPipe()`
- Method `saveToPipe()`
- Method `readFromPipe()`
- Method `writeToPipe()`

The Genero Web Services service library has been enhanced to support global SSL/TLS security configuration in `FGLPROFILE` for HTTPS communication.

You can now define the SSL/TLS certificate and private key to be used for all secured connections with the following entries and still use a dedicated SSL/TLS configuration if needed for a particular server:

- Entry `security.global.certificate`
- Entry `security.global.privatekey`
- Entry `security.global.keysubject` (Windows only)
- Entry `security.global.protocol`

A universal unique identifier function, `CreateUUIDString()`, has been added to the COM library. This function generates a universal unique identifier in BDL.

The Genero Web services library has been enhanced with two new serializers:

- `xml.Serializer.DomToStax()` converts a Dom node to a Stax writer
- `xml.Serializer.StaxToDom()` converts a Stax reader to a Dom node

Upgrade notes for web services.

### Reference

See [XML security classes](#) on page 3047.

See [The xml package](#) on page 2931

- `xml.DomDocument.loadFromPipe` on page 2953,
- `xml.DomDocument.saveToPipe` on page 2955,
- `xml.StaxReader.readFromPipe` on page 3029,
- `xml.StaxWriter.writeToPipe` on page 3011).

See [Web Services FGLPROFILE configuration](#) on page 3489.

This method is desupported since 3.00, use `security.RandomGenerator.CreateUUIDString` on page 3128 as replacement.


See [Web Services changes](#) on page 172.

### BDL 2.32 new features

Features added in 2.32 releases of the Genero Business Development Language.

**Important:** This page covers only those new features introduced with the Genero BDL version specified in the page title. Check prior new features pages if you migrate from an earlier version. Make sure to also read the upgrade guide corresponding to this Genero version.

Corresponding upgrade guide: [BDL 2.32 upgrade guide](#) on page 176.
Table 45: Web Services

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>The COM library enables to intercept high-level web services operation on server side. You can now define three BDL functions via methods of the web service class. They will be executed at different steps of a web service request processing in order to modify the SOAP request, response or the generated WSDL document before or after the SOAP engine has processed it. This helps handle WS-* specifications not supported in the web service API.</td>
<td>See The WebService class on page 2811.</td>
</tr>
<tr>
<td>• Method registerWSDLHandler()</td>
<td></td>
</tr>
<tr>
<td>• Method registerInputRequestHandler()</td>
<td></td>
</tr>
<tr>
<td>• Method registerOutputRequestHandler()</td>
<td></td>
</tr>
<tr>
<td>All three kinds of BDL callback functions must conform to the following prototype:</td>
<td></td>
</tr>
</tbody>
</table>

BDL 2.30 new features

Features added in 2.30 releases of the Genero Business Development Language.

**Important:** This page covers only those new features introduced with the Genero BDL version specified in the page title. Check prior new features pages if you migrate from an earlier version. Make sure to also read the upgrade guide corresponding to this Genero version.

Corresponding upgrade guide: BDL 2.30 upgrade guide on page 178.

Table 46: Core language

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Genero is now available on Mac OS-X™. You need at least Mac OS X version 10.5. The Operating System code for Mac OS X 10.5 64-bit is m64x105. Platform identifier is now displayed when using the -V option with command-line tools. The FGLPROFILE environment variable now accepts multiple file specification with an operating-system-specific path separator. The LOAD, UNLOAD and base.Channel class support the “CSV” delimiter specification to read/write files in Comma Separated Value format. Version 2.30.04 supports now the fglrun.arrayIgnoreRangeError entry which can be set to true to force the runtime system to return the first element of an array when the array index is out of bounds. The version 2.30.04 introduces the new fglrun.mapAnyErrorToError FGLPROFILE entry. This configuration parameter can be set to true to map the default action of the WHENEVER ANY ERROR exceptions to the action defined for the WHENEVER ERROR exception type.</td>
<td>See Supported operating systems on page 37. See fglrun on page 2065. See The FGLPROFILE file(s) on page 255. See LOAD on page 736, UNLOAD on page 740 and The Channel class on page 2412. See Arrays on page 418. See Exceptions on page 502.</td>
</tr>
</tbody>
</table>
### Table 47: User interface

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drag &amp; Drop support in DISPLAY ARRAY for tables or tree views.</td>
<td>See The DragDrop class on page 2570</td>
</tr>
<tr>
<td>A new form item type called WEBCOMPONENT is provided to integrate</td>
<td>See WEBCOMPONENT item type on page 1293.</td>
</tr>
<tr>
<td>external Java-Script-based widgets in your forms.</td>
<td></td>
</tr>
<tr>
<td>New ui.Form class method to make a specific form field visible, showing</td>
<td>See ui.Form.ensureFieldVisible on page 2502 and</td>
</tr>
<tr>
<td>the parent containers automatically.</td>
<td>ui.Form.ensureElementVisible on page 2501.</td>
</tr>
<tr>
<td>This method can also be used to bring a given folder page to the front,</td>
<td></td>
</tr>
<tr>
<td>even if the field is not active (i.e. not driven by a dialog).</td>
<td></td>
</tr>
<tr>
<td>The ERROR and MESSAGE instructions get an additional STYLE attribute,</td>
<td>See MESSAGE on page 1453.</td>
</tr>
<tr>
<td>to reference a presentation style and define the rendering with font,</td>
<td></td>
</tr>
<tr>
<td>color, and position.</td>
<td></td>
</tr>
<tr>
<td>New style for TOOLBAR and TOPMENU elements. See Front-End</td>
<td>See Toolbars on page 1436 and</td>
</tr>
<tr>
<td>documentation for more details about possible decoration attributes.</td>
<td>Topmenus on page 1444.</td>
</tr>
<tr>
<td>As with COMBOBOX, the items of a RADIOGROUP are now filled with</td>
<td>See RADIOGROUP item definition on page 1343.</td>
</tr>
<tr>
<td>the values of the INCLUDE attribute, if specified.</td>
<td></td>
</tr>
<tr>
<td>Identify the last clicked CANVAS item with the drawGetClickedItemId()</td>
<td>See Example 1: Simple canvas on page 2017.</td>
</tr>
<tr>
<td>function of fgldraw.4gl.</td>
<td></td>
</tr>
<tr>
<td>The FIELD_TOUCHED() operator and ui.Dialog.getFieldTouched() method</td>
<td>See FIELD_TOUCHED() on page 2532.</td>
</tr>
<tr>
<td>accept now a simple star as parameter, in order to check all fields used</td>
<td></td>
</tr>
<tr>
<td>by the dialog.</td>
<td></td>
</tr>
<tr>
<td>The JUSTIFY attribute is now supported for all form item types, in</td>
<td>See JUSTIFY attribute on page 1377.</td>
</tr>
<tr>
<td>order to let you specify both the data justification in the field/cell</td>
<td></td>
</tr>
<tr>
<td>and the alignment of the table column header.</td>
<td></td>
</tr>
<tr>
<td>The ui.Dialog.setFieldActive() method takes now a list of fields as</td>
<td>See ui.Dialog.setFieldActive on page 2549.</td>
</tr>
<tr>
<td>parameter, with the &quot;dot-asterisk&quot; notation, like the setFieldTouched()</td>
<td></td>
</tr>
<tr>
<td>method.</td>
<td></td>
</tr>
<tr>
<td>This new feature is part of the fix for bug #18224.</td>
<td>See The Dialog class on page 2512.</td>
</tr>
<tr>
<td>When modifying a tree during the dialog execution (for example, when</td>
<td></td>
</tr>
<tr>
<td>implementing dynamic trees with ON EXPAND / ON COLLAPSE triggers), if</td>
<td></td>
</tr>
<tr>
<td>you use the ui.Dialog.insertRow(), ui.Dialog.deleteRow() or ui.Dialog.</td>
<td></td>
</tr>
<tr>
<td>deleteAllRows() methods to modify the node list, the internal tree</td>
<td></td>
</tr>
<tr>
<td>structure was corrupted. The program array can be safely modified</td>
<td></td>
</tr>
<tr>
<td>directly with array methods, but multi-range selection flags and cell</td>
<td></td>
</tr>
<tr>
<td>attributes are not synchronized when doing this. Starting with 2.30.02,</td>
<td></td>
</tr>
<tr>
<td>you can now use the ui.Dialog.insertNode(), ui.Dialog.appendNode() and</td>
<td></td>
</tr>
<tr>
<td>ui.Dialog.deleteNode() methods to manipulate the node list and get</td>
<td></td>
</tr>
<tr>
<td>additional data like row selection flags and cell attributes</td>
<td></td>
</tr>
<tr>
<td>synchronized.</td>
<td></td>
</tr>
<tr>
<td>Upgrade notes for presentations styles.</td>
<td>See Presentation styles changes on page 180.</td>
</tr>
</tbody>
</table>
### Table 48: SQL databases

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>New database drivers</td>
<td>List of new database drivers:</td>
</tr>
<tr>
<td></td>
<td>• dbmase0Fx for SAP ASE 15.x (2.30.01) (Warning: SAP ASE is desupported in BDL version 4.00)</td>
</tr>
<tr>
<td></td>
<td>• dbmmys55x for a Mysql 5.5.x client (2.30.01)</td>
</tr>
<tr>
<td></td>
<td>• dbmpgs90x for a PostgreSQL 9.0.x client (2.30.02)</td>
</tr>
</tbody>
</table>

Informix® SMALLFLOAT and FLOAT can now be stored in Oracle native BINARY_FLOAT / BINARY_DOUBLE types.

The LOAD, UNLOAD and base.Channel class support the "CSV" delimiter specification to read/write files in Comma Separated Value format.

Use the fgl_db_driver_type() built-in function to identify the target database type.

In order to identify the reason why a database driver cannot be loaded, when setting FGLSQLDEBUG you now get an additional debug message that contains the operating system error message (dlerror())

The fgldbsch tool can now extract database schema from SQLite. However, pay attention to the data types used in SQLite (V 3.6): This database supports some standard type names in the SQL syntax but in reality the types used to store data are very limited. For example, a DATE will be stored as an integer or string (i.e. there is no native DATE type). See SQLite documentation for more details.

The fgldbsch tool will extract the schema based on the original type names used to create the table.

### BDL 2.21 new features

Features added in 2.21 releases of the Genero Business Development Language.

**Important:** This page covers only those new features introduced with the Genero BDL version specified in the page title. Check prior new features pages if you migrate from an earlier version. Make sure to also read the upgrade guide corresponding to this Genero version.

Corresponding upgrade guide: BDL 2.21 upgrade guide on page 180.

### Table 49: Core language

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program module dependency specification with IMPORT FGL instruction.</td>
<td>See The IMPORT FGL instruction.</td>
</tr>
</tbody>
</table>
### Support for C1 Ming Guo date format modifier

Enable the digit-based Ming Guo date format by adding the C1 modifier at the end of the value set for the DBDATE environment variable:

```
$ DBDATE="Y3MD/C1"
$ export DBDATE
```

**Note:**
- When using C1, the possible values for the Yn specifier are Y4, Y3, Y2.
- The MDY() function is sensitive to the C1 modifier usage in DBDATE.
- The USING operator supports the C1 modifier as well.
- The C2 modifier to use Era names is not supported.
- Unlike Informix® 4gl, when using negative years, the minus sign is placed over the left-most zero of the year.
- Front-ends may not support the Ming Guo calendar for widgets like DATEEDIT.

---

### Table 50: User interface

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support for C1 Ming Guo date format modifier: Enable the digit-based Ming Guo date format by adding the C1 modifier at the end of the value set for the DBDATE environment variable:</td>
<td>See DBDATE on page 267.</td>
</tr>
</tbody>
</table>
| $ DBDATE="Y3MD/C1"
$ export DBDATE                                                          |                                  |
| **Note:**                                                               |                                  |
| • When using C1, the possible values for the Yn specifier are Y4, Y3, Y2. |                                  |
| • The MDY() function is sensitive to the C1 modifier usage in DBDATE.   |                                  |
| • The USING operator supports the C1 modifier as well.                  |                                  |
| • The C2 modifier to use Era names is not supported.                     |                                  |
| • Unlike Informix® 4gl, when using negative years, the minus sign is     |                                  |
|    placed over the left-most zero of the year.                           |                                  |
| • Front-ends may not support the Ming Guo calendar for widgets like DATEEDIT. |                                  |

---

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>VALUEMIN/VALUEMAX attributes for the SPINEDIT widget.</td>
<td>See SPINEDIT</td>
</tr>
<tr>
<td>New presentation styles attributes for Image nodes.</td>
<td>See alignment.</td>
</tr>
<tr>
<td>Numeric keypad decimal separator: The decimal separator defined by DBMONEY or DBFORMAT will be used when pressing the dot key of the numeric keypad.</td>
<td>See DBMONEY and DBFORMAT.</td>
</tr>
<tr>
<td>Automatic display of BYTE images: Image data contained in a BYTE variable are now displayed automatically when using a simple DISPLAY BY NAME, DISPLAY TO or when the BYTE variable is used by a dialog instruction. The BYTE data must be located in a file (LOCATE IN FILE &quot;path&quot;) or temp file (LOCATE IN FILE).</td>
<td>See IMAGE item definition on page 1340.</td>
</tr>
<tr>
<td>Paged DISPLAY ARRAY supports undefined initial row count: With this feature, when using a Paged DISPLAY ARRAY, it was mandatory to provide the total number of rows in the result set, which required a SELECT COUNT(*) before executing the dialog instruction. The dialog now supports an undefined number of rows, with value -1 in the COUNT dialog attribute.</td>
<td>See Record list (DISPLAY ARRAY) on page 1501.</td>
</tr>
<tr>
<td>New ui.Interface.setSize() method to to let you define the initial size of the WCI container window.</td>
<td>See The Interface class on page 2470.</td>
</tr>
<tr>
<td>New formScroll presentation style attribute for windows.</td>
<td>See Window style attributes: Miscellaneous on page 1225.</td>
</tr>
<tr>
<td>Overview</td>
<td>Reference</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>New database drivers</td>
<td>List of new database drivers:</td>
</tr>
<tr>
<td></td>
<td>• dbmesmA0 for an Easysoft 1.2.3 client</td>
</tr>
<tr>
<td></td>
<td>• dbmpgs84x for a PostgreSQL 8.4.x client</td>
</tr>
<tr>
<td></td>
<td>• dbmoraB2x for Oracle 11g release 2 (11.2)</td>
</tr>
<tr>
<td>New Easysoft driver to connect from UNIX™ to SQL Server. This driver is</td>
<td>See Microsoft SQL Server on page 827.</td>
</tr>
<tr>
<td>based on the Easysoft SQL Server ODBC client.</td>
<td></td>
</tr>
<tr>
<td>New PostgreSQL 8.4 driver with INTERVAL support: dbmpgs84x. This driver</td>
<td>See PostgreSQL on page 964.</td>
</tr>
<tr>
<td>converts Informix-style INTERVALs to native PostgreSQL INTERVALs.</td>
<td></td>
</tr>
<tr>
<td>Static SQL column definition supports DEFAULT clause: The syntax of the</td>
<td>See Static SQL statements on page 694.</td>
</tr>
<tr>
<td>CREATE TABLE and ALTER TABLE Static SQL statements allows the DEFAULT</td>
<td></td>
</tr>
<tr>
<td>clause in column definitions.</td>
<td></td>
</tr>
</tbody>
</table>

```
CREATE TABLE item (  
um SERIAL,  
name VARCHAR(50)  
DEFAULT '<undefined>'  
NOT NULL )
```

PostgreSQL database driver supports now TEXT/BYTE. See TEXT and BYTE (LOB) types on page 985. See INSERT on page 696.

New Static SQL syntax for the INSERT statement, which removes the record member defined as SERIAL, SERIAL8 or BIGSERIAL in the schema file:

```
SCHEMA mydb
...
DEFINE record RECORD LIKE table.*
...
INSERT INTO table VALUES record.*
```

The LOAD can now raise error -846 when the input file has a corrupted line (missing or invalid field separator, invalid character set, UNIX/DOS line terminators). You can now easily find the invalid line by setting the FGLSQLDEBUG on page 281 environment variable. The runtime system will display such debug messages with the line number:

```
| DBI: LOAD: Corrupted data file,  
check line #12345.
```

ODBC Character type control with SNC driver is now possible by using simple char or wide-char character strings for ODBC, with the following FGLPROFILE entry:

```
dbi.database.<dbname>.snc.widechar = true/false
```
### Table 52: Web Services

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>The fglwsdl tool supports HTTPS request to retrieve WSDL or XSD on the</td>
<td>See fglwsdl on page 2081.</td>
</tr>
<tr>
<td>network. You must specify the X509 certificate and private key using</td>
<td></td>
</tr>
<tr>
<td>these options:</td>
<td></td>
</tr>
<tr>
<td>• (-cert) filename: The filename of the X509 PEM-encoded certificate.</td>
<td></td>
</tr>
<tr>
<td>• (-key) filename: The filename of the X509 PEM-encoded private key</td>
<td></td>
</tr>
<tr>
<td>associated to the above certificate.</td>
<td></td>
</tr>
<tr>
<td>• (-wCert) name: The name of the X509 certificate and its associated</td>
<td></td>
</tr>
<tr>
<td>private key in the Windows® key store (Windows® Only)</td>
<td></td>
</tr>
<tr>
<td>The fglwsdl tool allows http authentication and proxy authentication</td>
<td>See fglwsdl on page 2081.</td>
</tr>
<tr>
<td>when requesting a WSDL or an XSD on the network, and supports basic and</td>
<td></td>
</tr>
<tr>
<td>digest authentication. Two options have been added for authentication.</td>
<td></td>
</tr>
<tr>
<td>• (-pAuth) login password: The login and the password to be used</td>
<td></td>
</tr>
<tr>
<td>for proxy authentication.</td>
<td></td>
</tr>
<tr>
<td>• (-hAuth) login password: The login and the password to be used</td>
<td></td>
</tr>
<tr>
<td>for http or https authentication.</td>
<td></td>
</tr>
<tr>
<td>The fglwsdl tool provides a new option that generates:</td>
<td>See WS client stubs and handlers on page 3328.</td>
</tr>
<tr>
<td>• a client stub entirely based on the DOM API</td>
<td></td>
</tr>
<tr>
<td>• calls to a request, response and fault callback function per service</td>
<td></td>
</tr>
<tr>
<td>This option is especially useful when you have to communicate with</td>
<td></td>
</tr>
<tr>
<td>another web service that requires additional information on the</td>
<td></td>
</tr>
<tr>
<td>XML request, or when it returns additional information that was not</td>
<td></td>
</tr>
<tr>
<td>specified in the WSDL. For instance, this is the case if you have to</td>
<td></td>
</tr>
<tr>
<td>communicate with web services using WS-Security. You can manipulate</td>
<td></td>
</tr>
<tr>
<td>the XML document in the generated client stub using the XML-Signature</td>
<td></td>
</tr>
<tr>
<td>or XML-Encryption API to perform the security part by hand before it is</td>
<td></td>
</tr>
<tr>
<td>sent on the network.</td>
<td></td>
</tr>
<tr>
<td>The following option has been added for that purpose:</td>
<td></td>
</tr>
<tr>
<td>• (-domHandler): Generate function calls to a request, response and</td>
<td></td>
</tr>
<tr>
<td>fault callback handler, and force the use of DOM in the client stub.</td>
<td></td>
</tr>
<tr>
<td>The COM library is enhanced by a new function called HandleRequest</td>
<td>See com.WebServiceEngine.HandleRequest on page</td>
</tr>
<tr>
<td>to allow low-level and high-level web services on the same server.</td>
<td>2833</td>
</tr>
<tr>
<td>The COM library is enhanced to perform automatic reply on HTTP GET</td>
<td>See com.HTTPRequest.setAutoReply on page 2885.</td>
</tr>
<tr>
<td>request when the server requires HTTP authentication, proxy authentication,</td>
<td></td>
</tr>
<tr>
<td>or returns an HTTP redirect.</td>
<td></td>
</tr>
<tr>
<td>The XML library supports a new option, xml_useutctime, to serialize any</td>
<td>See Serialization option flags on page 3046.</td>
</tr>
<tr>
<td>BDL DATE and DATETIME using the UTC format requested in most WS-Security</td>
<td></td>
</tr>
<tr>
<td>exchanges.</td>
<td></td>
</tr>
<tr>
<td>Overview</td>
<td>Reference</td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
</tr>
</tbody>
</table>
| The XML library has been enhanced with two APIs in the CryptoKey class. Due to security issues, the usage of a direct shared symmetric or HMAC key is not recommended; operations are secure using a key derived from a common shared key instead. The XML library has been enhanced with two APIs in the CryptoKey class:  
  • Constructor `CreateDerivedKey()`  
  • Method `deriveKey()` | See Derived keys on page 3065. |
| The COM library has been enhanced with two helper APIs in a new Util class. In most Web Service security exchanges, the application must be able to compute digest passwords and use random binary data to detect reply attacks (for instance). The COM library has therefore been enhanced with two helper APIs in a new Util class:  
  • Static method `CreateDigestString()`  
  • Static method `CreateRandomString()` | These methods are desupported since 3.00, use `security.Digest.CreateDigestString` on page 3146 and `security.RandomGenerator.CreateRandomString` on page 3128. |
| The StAX reader and writer classes have been enhanced with two new methods to set up the XML stream on a TEXT lob. It enables parsing of an XML document in StAX directly from a TEXT with the `readFromText()` method, and creating a new XML document saved directly as TEXT with the `writeToText()` method. | See The StaxWriter class on page 2999 and The StaxReader class on page 3012. |
| The Genero Web Services library has been enhanced to support XML wildcard attributes. Such wildcard attribute can be set in a XML schema or in a WSDL via the `anyAttribute` tag. It allows additional attributes belonging to other XML schemas in a main XML schema. The additional attributes are not necessarily known by the main schema. The fglwsdl tool has been enhanced to recognize the additional attribute and to generate a one-dimensional dynamic array with a new `XMLAnyAttribute` attribute, and the XML Serializer has been enhanced to handle the new `XMLAnyAttribute` during the serialization and deserialization process. | See Attributes to customize XML serialization on page 3523 and The Serializer class on page 3033. |
| A new option called `xs_processcontents` is supported by the XML Serializer to generate the XML schema of such wildcard attributes with a processContents tag that defines the way a validator will handle them. | N/A. |
| The package contains a new demo called SimplePKI that demonstrates the usage of XML-Encryption in Genero. It allows several clients to register to a centralized PKI (Public Key Infrastructure) service that generates a unique RSA key-pair per user. The private key is returned to the user during the registration or login, using a derived symmetric key based on the user's password to make it secure. Then any client is able to retrieve the public key of the registered users, and to encrypt XML data only readable by that user. | N/A. |
| **Note:** This demo can easily be adapted in a real-world application if (for instance) all key-pair are stored in a database for persistence. You can find the demo in the demo/WebServices/simplepki subdirectory or by running the demo application in your installation directory. | N/A. |
BDL 2.20 new features

Features added in 2.20 releases of the Genero Business Development Language.

Important: This page covers only those new features introduced with the Genero BDL version specified in the page title. Check prior new features pages if you migrate from an earlier version. Make sure to also read the upgrade guide corresponding to this Genero version.

Corresponding upgrade guide: BDL 2.20 upgrade guide on page 183.

Table 53: Core language

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Java Interface allows your programs to use the Java library.</td>
<td>See Java Interface.</td>
</tr>
<tr>
<td>New TINYINT, BIGINT and BOOLEAN data types.</td>
<td>See TINYINT on page 306, BIGINT on page 290, BOOLEAN on page 293.</td>
</tr>
<tr>
<td>Private functions: It is now possible to hide a function (or report) to</td>
<td>See Understanding functions on page 436.</td>
</tr>
<tr>
<td>the other modules with the new PRIVATE keyword.</td>
<td></td>
</tr>
<tr>
<td>Automatic source documentation generator.</td>
<td>See Source documentation on page 2126.</td>
</tr>
<tr>
<td>The fglcomp compiler has been extended with a new option (--timestamp)</td>
<td>See fglcomp on page 2071.</td>
</tr>
<tr>
<td>to write the compilation timestamp to the generated 42m p-code module.</td>
<td></td>
</tr>
<tr>
<td>If present, the timestamp will be printed when using fglrun -b. Use</td>
<td></td>
</tr>
<tr>
<td>compilation timestamps only if really needed; every new compiled .42m</td>
<td></td>
</tr>
<tr>
<td>module will be different, even if the source code has not changed.</td>
<td></td>
</tr>
<tr>
<td>The FGLRESOURCEPATH environment variable to define search paths for</td>
<td>See FGLRESOURCEPATH on page 279.</td>
</tr>
<tr>
<td>program resource files like forms.</td>
<td></td>
</tr>
<tr>
<td>New precision math built-in functions for DECIMAL data.</td>
<td>See fgl_decimal_truncate() on page 2276, fgl_decimal_sqrt() on page 2276, fgl_decimal_exp() on page 2276, fgl_decimal_login() on page 2276, fgl_decimal_power() on page 2277.</td>
</tr>
<tr>
<td>Automatic Code Completion with VIM: If you have Vim 7 installed, you</td>
<td>See Source code edition on page 2123.</td>
</tr>
<tr>
<td>can now use .per and .4gl code completion.</td>
<td></td>
</tr>
</tbody>
</table>

Table 54: Reports

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>The START REPORT instruction now allows to specify the XML SAX Document</td>
<td>See TO XML HANDLER syntax.</td>
</tr>
<tr>
<td>Handler to process XML output with the TO XML HANDLER clause.</td>
<td></td>
</tr>
<tr>
<td>Report definition file generation with fglcomp --build-rdd option.</td>
<td>See See fglcomp on page 2071.</td>
</tr>
</tbody>
</table>

Table 55: User interface

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support for typical Tree-View widgets with the new TREE container.</td>
<td>See Tree views on page 1889.</td>
</tr>
<tr>
<td>Overview</td>
<td>Reference</td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
</tr>
<tr>
<td>The traditional user interface mode: To simplify migration from Informix® 4GL or Four Js BDS, you can now run applications in traditional mode to render windows as simple boxes, as in the WTK front-end.</td>
<td>See Graphical mode with Traditional Display on page 1114.</td>
</tr>
<tr>
<td>Phantom form fields can be used to define the screen-record or screen-array, but are not used in the LAYOUT section of the form. Phantom fields are especially useful when implementing a TREE container.</td>
<td>See Phantom fields on page 1246.</td>
</tr>
<tr>
<td>Multi-row selection allows end users to highlight several rows in a list of records.</td>
<td>See Syntax of DISPLAY ARRAY instruction on page 1502.</td>
</tr>
<tr>
<td>Built-in sort works now in INPUT ARRAY.</td>
<td>See List ordering on page 1862.</td>
</tr>
<tr>
<td>New <code>contextMenu</code> action default attribute to allow you to specify whether the menu option is visible in the default context menu. The default value is &quot;yes&quot; - the option is visible whenever the action is visible.</td>
<td>See Action defaults files on page 1162.</td>
</tr>
<tr>
<td>New <code>integratedSearch</code> presentation style attribute for TEXTEDIT fields to enable text search.</td>
<td>See TextEdit style attributes on page 1212.</td>
</tr>
<tr>
<td>FOLDER elements can now use a &quot;position&quot; style attribute to define the position (top, left, right, bottom) of folder tabs.</td>
<td>See Folder style attributes on page 1195.</td>
</tr>
<tr>
<td>BUTTON form items get a new &quot;buttonType&quot; attribute to define the rendering of the button.</td>
<td>See Button style attributes on page 1185.</td>
</tr>
<tr>
<td>MENU object created with the popup option can be placed with the &quot;position&quot; style attribute.</td>
<td>See Menu style attributes on page 1199.</td>
</tr>
<tr>
<td>Window Menu and Action panel decoration can be customized using the new &quot;ringMenuDecoration&quot;, &quot;actionPanelDecoration&quot; style attributes.</td>
<td>See Window style attributes: Ring Menu on page 1230, Window style attributes: Action Panel on page 1228.</td>
</tr>
<tr>
<td>The new &quot;tabbedContainer&quot;, &quot;tabbedContainerCloseMethod&quot; style attributes can be used to turn on and customize tabbed WCI containers.</td>
<td>See Window style attributes: Basics on page 1220.</td>
</tr>
<tr>
<td>TABLE elements can use the new &quot;tableType&quot; attribute to render data in different ways. The new &quot;resizeFillsEmptySpace&quot; attribute can be used to define how the last column is resized when the table is resized.</td>
<td>See Table style attributes on page 1204.</td>
</tr>
<tr>
<td>All items with an IMAGE attribute can use the new &quot;imageCache&quot; attribute to define if the picture can be cached locally on the front-end.</td>
<td>Starting with GDC 3.20, this style attribute has been desupported. See Presentation style changes in BDL 3.20.</td>
</tr>
<tr>
<td>New Front-End Functions &quot;getWindowId&quot;, &quot;feInfo&quot;, &quot;launchURL&quot;.</td>
<td>See Standard front calls on page 2647.</td>
</tr>
<tr>
<td>Front-End protocol compression can now be disabled with a new FGLPROFILE entry. This is especially useful in fast networks to save processor time.</td>
<td>See GUI protocol compression on page 1119.</td>
</tr>
<tr>
<td>New built-in functions are now available to control the part of the text that is selected in the current field.</td>
<td>See <code>fgl_dialog_getselectionend()</code> on page 2281, <code>fgl_dialog_setselection()</code> on page 2282.</td>
</tr>
<tr>
<td>New IMAGE attribute in form LAYOUT element: The LAYOUT section of a form definition can now use the IMAGE attribute to define the icon to be used for the parent Window. This is especially useful in a Container-based application, to distinguish child programs inside the WCI container.</td>
<td>See LAYOUT section on page 1301.</td>
</tr>
</tbody>
</table>
Use the new INFIELD clause in ON ACTION interactive block to automatically enable/disable the action when entering/leaving the specified field.

Getting the current active dialog with ui.Dialog.getCurrent().

**Table 56: SQL databases**

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>New database drivers.</td>
<td>List of new database drivers:</td>
</tr>
<tr>
<td>MySQL Driver supports TEXT/BYTE data types.</td>
<td>• dbmsqt3xx for an SQLite 3 library (2.20.01)</td>
</tr>
<tr>
<td>To work around conflicts with the Informix® database path specification in DBPATH, use the FGLRESOURCEPATH environment variable.</td>
<td>See Oracle MySQL / MariaDB on page 876.</td>
</tr>
<tr>
<td>Database user authentication callback function can be used to specify a database user and password when the DATABASE instruction cannot be replaced by CONNECT TO.</td>
<td>See FGLRESOURCEPATH on page 279.</td>
</tr>
<tr>
<td>FGLSQLDEBUG output is improved to display and SQL command header with SQL command name and source/line information before executing the underlying ODI driver code. If the driver code crashes or stops the process with an assertion, you can easily identify the last SQL instruction that was executed.</td>
<td>See User authentication callback function on page 680.</td>
</tr>
</tbody>
</table>

**Table 57: Web Services**

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Genero Web Services XML Library has been improved to support the XML-Signature and XML-Encryption specifications defined by the W3C (also known as XML-Security).</td>
<td>See XML security classes on page 3047.</td>
</tr>
<tr>
<td>The library enables BDL applications to handle public, private, symmetric or hmac keys and X509 certificates in order to sign XML documents or document fragments, and verify a XML signature against a certificate or key. It also enables the applications to encrypt XML nodes using symmetric keys, and decrypt them back using DOM manipulation. Combined with the COM library, any BDL application can now exchange any XML documents over the Internet in a completely secured manner.</td>
<td></td>
</tr>
<tr>
<td>The library provides classes for:</td>
<td></td>
</tr>
<tr>
<td>• Manipulating cryptography keys</td>
<td></td>
</tr>
<tr>
<td>• Handling X509 certificates for identification</td>
<td></td>
</tr>
<tr>
<td>• Encrypting and decrypting XML documents, document fragments, or symmetric keys</td>
<td></td>
</tr>
<tr>
<td>• Signing XML documents, document fragments, or any kind of data, and validating them againstXML signatures</td>
<td></td>
</tr>
</tbody>
</table>
Overview | Reference
--- | ---
The Genero Web Services XML library provides APIs to encrypt and decrypt strings with symmetric or RSA public/private keys. These APIs can be used to encrypt/decrypt passwords directly in BDL applications. | See The Encryption class on page 3104 and fglpass on page 2087.
The Genero Web Services provides support for the new BOOLEAN, TINYINT and BIGINT data types. | See Attributes to customize XML serialization on page 3523 and fglwsdl on page 2081.
You can use these data types when writing your web service or to customize your BDL RECORDs for XML serialization. The fglwsdl tool has been enhanced to generate these new data types automatically when encountered in WSDL files or XML schemas. | 
Note: For compatibility issues, the fglwsdl tool allows code generation without these new data types by using the option ‘--legacyTypes’.

**BDL 2.11 new features**
Features added in 2.11 releases of the Genero Business Development Language.

**Important:** This page covers only those new features introduced with the Genero BDL version specified in the page title. Check prior new features pages if you migrate from an earlier version. Make sure to also read the upgrade guide corresponding to this Genero version.

Corresponding upgrade guide: BDL 2.11 upgrade guide on page 189.

**Table 58: Core language**

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>New -p noln preprocessor option to remove line number information to get a readable output:</td>
<td>See Source preprocessor on page 2132.</td>
</tr>
<tr>
<td>fglcomp -E -p noln mymodule.4gl</td>
<td></td>
</tr>
<tr>
<td>The -b option of fglrun has been extended to recognize headers of p-code modules compiled with older versions of Genero.</td>
<td>See 42m module information on page 2120.</td>
</tr>
<tr>
<td>The fglform compiler now writes build information in the .42f files, to identify on the production site what version was used to compile forms.</td>
<td>See Compiling form specification files (.per) on page 2106.</td>
</tr>
</tbody>
</table>

**Table 59: User interface**

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>The ui.ComboBox class has been extended with new methods: getTextOf() and getIndexOf().</td>
<td>See The ComboBox class on page 2561.</td>
</tr>
<tr>
<td>A new FGLPROFILE entry has been added to force the current row to be shown automatically after a sort in a table:</td>
<td>See Dialog configuration with FGLPROFILE on page 1708.</td>
</tr>
<tr>
<td>Dialog.currentRowVisibleAfterSort = 1</td>
<td></td>
</tr>
<tr>
<td>By default, the offset does not change and the current row may disappear from the window. When this new parameter is used, the current row will always be visible.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 60: SQL databases

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static SQL syntax now supports derived tables and derived column lists in the FROM clause. For example:</td>
<td>See SELECT on page 700.</td>
</tr>
<tr>
<td><code>SELECT * FROM (SELECT * FROM customer ORDER BY cust_num) AS t(c1,c2,c3,...)</code></td>
<td></td>
</tr>
<tr>
<td>See database server documentation for more details about this SQL feature.</td>
<td></td>
</tr>
<tr>
<td>Informix® 11 does not support the full ANSI SQL 92 specification for derived columns, while other databases like DB2® do. For this reason, fglcomp allows the ANSI standard syntax.</td>
<td>See SET ISOLATION on page 692.</td>
</tr>
<tr>
<td>The SET ISOLATION statement now supports the new Informix® 11 clauses for the COMMITTED READ option:</td>
<td></td>
</tr>
<tr>
<td><code>SET ISOLATION TO COMMITTED READ [LAST COMMITTED] [RETAIN UPDATE LOCKS]</code></td>
<td></td>
</tr>
<tr>
<td>When connecting to a non-Informix database, the LAST COMMITTED and RETAIN UPDATE LOCKS are ignored; other databases do not support these options, and have the same behavior as when these options are used with Informix® 11.</td>
<td></td>
</tr>
<tr>
<td>The CAST operator can now be used in static SQL statements:</td>
<td>See Static SQL statements on page 694.</td>
</tr>
<tr>
<td><code>CAST ( expression AS sql-data-type )</code></td>
<td></td>
</tr>
<tr>
<td>Only Informix® data types are supported after the AS keyword.</td>
<td>See CONNECT TO on page 683.</td>
</tr>
<tr>
<td>In order to execute database administration tasks, you can now connect to Oracle as SYSDBA or SYSOPER with the CONNECT instruction:</td>
<td></td>
</tr>
<tr>
<td><code>CONNECT TO &quot;dbname&quot; USER &quot;scott/SYSDBA&quot; USING &quot;tiger&quot;</code></td>
<td></td>
</tr>
</tbody>
</table>

### Table 61: Web Services - Version 2.11.00

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Genero Web Services com library provides the HTTPServiceRequest class to perform low-level XML and TEXT over HTTP communication on the server side. This allows communication at a very low-level layer, to write your own type of web services.</td>
<td>See The HTTPServiceRequest class on page 2845.</td>
</tr>
</tbody>
</table>
### XML facet constraints attributes:

The Genero Web Services XML library provides 12 new XML attributes to map to simple BDL variables. These attributes restrict the acceptable value-space for each variable in different ways such as:

- a minimum or a maximum number of XML characters or bytes.
- a strict number of XML characters or bytes.
- a minimum inclusive or exclusive value depending on the data type.
- a maximum inclusive or exclusive value depending on the data type.
- a enumeration of authorized values.
- a number of digits and fraction digits.
- how whitespaces have to be handled.
- a regular expression to match. (See Section F of XML Schema Part 2)

The fglwsdl tool has been enhanced with the following three new options:

- **-disk**: to retrieve locally a WSDL or an XSD with all its dependencies from an URL on the disk
- **-noFacets**: to avoid the generation of the new facet constrain attributes (for compatibility)
- **-regex**: to validate a value against a regular expression as described in the XML Schema specification

### The Genero Web Services library

The Genero Web Services library provides two new methods in the `WebOperation` class to create One-Way operations in services.

A One-Way operation means that the server accepts an incoming request, but doesn't return any response back to the client. There is one method called `CreateOneWayRPCStyle` to create an RPC Style operation, and another one called `CreateOneWayDOCStyle` to create a Document Style operation.

For instance, a One-Way operation can be used as a logger service, where a client sends a message to the server, but doesn't care about what the server is doing with it.

The fglwsdl tool has been enhanced with the following new options:

- **-b**: Generate code from a WSDL using the binding section instead of the service section
- **-autoNsPrefix**: Determine the prefix for variables and types based on the XML namespace they belong to
- **-nsPrefix**: Set the prefix for a variable or a type belonging to the given XML namespace

The following options have been changed:

- **-o**: If there are several services in one WSDL, they will be generated in the same file with the given base name instead of returning an error
- **-disk**: Retrieves and displays all dependencies to the current directory but there are no sub directories any longer.
- **-prefix**: Accepts patterns %s, %f and %p

### Table 62: Web Services - Version 2.11.04

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Genero Web Services library provides two new methods in the <code>WebOperation</code> class to create One-Way operations in services.</td>
<td>See The <code>WebOperation</code> class on page 2821.</td>
</tr>
<tr>
<td>A One-Way operation means that the server accepts an incoming request, but doesn't return any response back to the client. There is one method called <code>CreateOneWayRPCStyle</code> to create an RPC Style operation, and another one called <code>CreateOneWayDOCStyle</code> to create a Document Style operation.</td>
<td></td>
</tr>
<tr>
<td>For instance, a One-Way operation can be used as a logger service, where a client sends a message to the server, but doesn't care about what the server is doing with it.</td>
<td></td>
</tr>
<tr>
<td>The fglwsdl tool has been enhanced with the following new options:</td>
<td>See fglwsdl on page 2081.</td>
</tr>
<tr>
<td>- <strong>-b</strong>: Generate code from a WSDL using the binding section instead of the service section</td>
<td></td>
</tr>
<tr>
<td>- <strong>-autoNsPrefix</strong>: Determine the prefix for variables and types based on the XML namespace they belong to</td>
<td></td>
</tr>
<tr>
<td>- <strong>-nsPrefix</strong>: Set the prefix for a variable or a type belonging to the given XML namespace</td>
<td></td>
</tr>
<tr>
<td>The following options have been changed:</td>
<td></td>
</tr>
<tr>
<td>- <strong>-o</strong>: If there are several services in one WSDL, they will be generated in the same file with the given base name instead of returning an error</td>
<td></td>
</tr>
<tr>
<td>- <strong>-disk</strong>: Retrieves and displays all dependencies to the current directory but there are no sub directories any longer.</td>
<td></td>
</tr>
<tr>
<td>- <strong>-prefix</strong>: Accepts patterns %s, %f and %p</td>
<td></td>
</tr>
</tbody>
</table>
The Genero Web Services library has been enhanced to support WSDL with circular references.

The Genero language doesn't provide a way to define variables or types that refer to themselves. However, to provide better interoperability and a way to handle such circular data, the fglwsdl tool now generates variables or types of xml.DomDocument type when circular references are detected during the processing of WSDL files. This gives the user the ability to manipulate the circular data by hand, using the XML DOM API.

See The xml package on page 2931.

### BDL 2.10 new features

Features added in 2.10 releases of the Genero Business Development Language.

**Important**: This page covers only those new features introduced with the Genero BDL version specified in the page title. Check prior new features pages if you migrate from an earlier version. Make sure to also read the upgrade guide corresponding to this Genero version.

Corresponding upgrade guide: BDL 2.10 upgrade guide on page 190.

<table>
<thead>
<tr>
<th>Table 63: Core language</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Overview</strong></td>
</tr>
<tr>
<td>The <strong>TRY/CATCH</strong> block can handle exceptions raised by the runtime system.</td>
</tr>
<tr>
<td>WHenever ... RAISE instructs the runtime system that an uncaught exception will be handled by the caller of the function.</td>
</tr>
<tr>
<td>NULL point exceptions can now be trapped as other exceptions: Error -8083 will be raised if you try to call an object method with a variable that does not reference an object (that contains NULL):</td>
</tr>
</tbody>
</table>

```
DEFINE x ui.Dialog
   -- x is NULL
CALL x.setFieldActive("fieldname",FALSE)
   -- raises -8083
```

In previous versions, the above code raised a fatal NULL pointer error.

The **base.Channel** class now provides a method to establish a client socket connection to a server, with the new openClientSocket() method.

For debugging purpose, get the stack trace of the program with the **base.Application.getStackTrace()** method.

Before version 2.10, it was only possible to assign a TEXT to a TEXT variable. It is now possible to assign STRING, CHAR and VARCHAR values to a TEXT variable.

The **fglr** run -e option now supports a comma-separated list of extensions, and -e can be specified multiple times:

```
fglr run -e ext1,ext2,ext3
   -e ext4,ext5 myprogram
```

See base.Channel.openClientSocket on page 2416

See base.Application.getStackTrace on page 2409.

See Type conversions on page 311.

See Loading C-Extensions at runtime on page 2234.
Overview

Get an action event when the user modifies the value of a field, with the predefined `dialogtouched` action, to detect first user modifications.

The `parse()` and `toString()` methods are now available for a `om.DomNode` object.

A `om.DomDocument` object can be created with `createFromString()`.

The `TEXT` and `BYTE` data types now support the methods `readFile(fileName)` and `writeFile(fileName)`.

<table>
<thead>
<tr>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>See <a href="#">Immediate detection of user changes</a> on page 1725.</td>
</tr>
<tr>
<td>See <a href="#">The DomNode class</a> on page 2584.</td>
</tr>
<tr>
<td>See <a href="#">The DomDocument class</a> on page 2577.</td>
</tr>
<tr>
<td>See <a href="#">BYTE</a> on page 290, <a href="#">TEXT</a> on page 307.</td>
</tr>
</tbody>
</table>

### Table 64: User interface

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>The new <code>DIALOG</code> instruction handles different parts of a form simultaneously.</td>
<td>See <a href="#">Multiple dialogs (DIALOG - inside functions)</a> on page 1585.</td>
</tr>
<tr>
<td><code>HBox</code> and <code>VBox</code> containers can now have a splitter.</td>
<td>See <a href="#">SPLITTER attribute</a> on page 1393.</td>
</tr>
<tr>
<td>The new <code>DOUBLECLICK</code> table allows to configure the action to be sent when the user double-clicks on a row.</td>
<td>See <a href="#">DOUBLECLICK attribute</a> on page 1364.</td>
</tr>
<tr>
<td>Define a timeout delay for front-end connections with the following <code>FGLPROFILE</code> entry:</td>
<td>See <a href="#">GUI connection timeout</a> on page 1118.</td>
</tr>
<tr>
<td><code>gui.connection.timeout = seconds</code></td>
<td></td>
</tr>
<tr>
<td>Before version 2.10, it was only possible to assign a <code>TEXT</code> to a <code>TEXT</code> variable. It is now possible to assign <code>STRING</code>, <code>CHAR</code> and <code>VARCHAR</code> values to a <code>TEXT</code> variable.</td>
<td>See <a href="#">Type conversions</a> on page 311.</td>
</tr>
<tr>
<td>Presentation styles have been extended:</td>
<td>See <a href="#">Presentation styles</a> on page 1165.</td>
</tr>
<tr>
<td>- The style attribute &quot;position&quot; for Windows™ can be set to &quot;previous&quot;.</td>
<td></td>
</tr>
<tr>
<td>- <code>TEXTEDIT</code> now has the &quot;textSyntaxHighlight&quot; attribute (value can be &quot;per&quot;, more to come...).</td>
<td></td>
</tr>
<tr>
<td>- All widgets can now use the &quot;localAccelerators&quot; global style attribute to interpret standard navigation and editor keys (like Home/End) without firing an action that uses the same keys as accelerators.</td>
<td></td>
</tr>
<tr>
<td>Get an action event when the user modifies the value of a field, with the predefined <code>dialogtouched</code> action, to detect first user modifications.</td>
<td>See <a href="#">Immediate detection of user changes</a> on page 1725.</td>
</tr>
<tr>
<td>Use the <code>validate=&quot;no&quot;</code> action default attribute to prevent data validation when executing an action.</td>
<td>See <a href="#">Data validation at action invocation</a> on page 1762.</td>
</tr>
<tr>
<td>Define a minimum width and height for forms with the <code>MINWIDTH</code>, <code>MINHEIGHT</code> attributes.</td>
<td>See <a href="#">MINHEIGHT attribute</a> on page 1380, <a href="#">MINWIDTH attribute</a> on page 1380.</td>
</tr>
<tr>
<td>In <code>INPUT ARRAY</code>, avoid the automatic creation of a temporary row with the new <code>AUTO APPEND = FALSE</code> dialog attribute.</td>
<td>See <a href="#">Appending rows in INPUT ARRAY</a> on page 1848.</td>
</tr>
</tbody>
</table>
### Table 65: SQL databases

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>The <code>fgldbsch</code> tool now supports the <code>x</code> conversion code to ignore table columns of a specific type. This is useful for ROWID-like columns such as SQL Server's <code>uniqueidentifier</code> columns.</td>
<td>See Data type conversion control on page 477.</td>
</tr>
<tr>
<td>Before version 2.10, SQL interruption was not supported well for some databases. SQL interruption is now available with all databases providing an API to cancel a long-running query.</td>
<td>See Using SQL interruption on page 601.</td>
</tr>
</tbody>
</table>

### Table 66: Web Services

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Genero Web Services XML library (xml) has been added. This library provides classes and methods to perform:</td>
<td>See The xml package on page 2931.</td>
</tr>
<tr>
<td>• XML manipulation with a W3C Document Object Model (DOM) API</td>
<td></td>
</tr>
<tr>
<td>• XML manipulation with a Streaming API for XML (StAX)</td>
<td></td>
</tr>
<tr>
<td>• Validation of DOM documents against XML Schemas</td>
<td></td>
</tr>
<tr>
<td>• Serialization of BDL variables in XML</td>
<td></td>
</tr>
<tr>
<td>• Creation of XML Schemas corresponding to BDL variables</td>
<td></td>
</tr>
<tr>
<td>New classes have been added to the Genero Web Services COM library to facilitate low-level XML and TEXT over HTTP and TCP Client communication (com).</td>
<td>See The HTTPRequest class on page 2871, The HTTPResponse class on page 2894, The TCPRequest class on page 2920 and The TCPResponse class on page 2920.</td>
</tr>
<tr>
<td>The Genero Web Services com library provides two classes, HTTPRequest and HTTPResponse, to perform low-level XML and TEXT over HTTP communications on the client side. Two more classes, TCPRequest and TCPResponse, are also provided to perform low-level XML and TEXT over TCP communications on the client side. This allows communication between applications using the core Web technology, taking advantage of the large installed base of tools that can process XML delivered plainly over HTTP or TCP, as well as SOAP over HTTP.</td>
<td></td>
</tr>
<tr>
<td>Specific streaming methods are also available to improve the communication by sending XML to the network even if the serialization process is not yet finished, as well as for the deserialization process.</td>
<td></td>
</tr>
<tr>
<td>It is also possible to prevent asynchronous requests from being blocked when waiting for a response, and to perform specific HTTP form encoded requests as specified in HTML4 or XForms1.0.</td>
<td></td>
</tr>
</tbody>
</table>
The `fglwsdl` tool now generates low-level and asynchronous client stubs from the WSDL.

The `fglwsdl` tool generates all client stubs with the low-level `HTTPRequest` and `HTTPResponse` classes of the `com` library to perform HTTP communications. The low-level generated stub also takes advantage of the streaming methods, if Document Style or RPC-Literal web services are performed. Streaming is not possible with RPC-Encoded web services, as nodes can have references to other nodes in the XML document, requiring the entire document in memory to perform serialization or deserialization.

The `fglwsdl` tool also generates two new BDL functions for each operation of a Web service. These two functions enable you to perform asynchronous web service operation calls by first sending the request, and retrieving the corresponding response later in the application. This allows you to prevent a BDL application from being blocked if the response of a web service operation takes a certain amount of time.

Genero Web Services provides an enhanced `fglwsdl` tool that is able to generate Genero data types from a XML schema. The data types can then be used in your application to be serialized or deserialized in XML. The resulting XML is a valid instance of that XML schema, and validation with a XML validator will succeed.

---

**BDL 2.02 new features**

Features added in 2.02 releases of the Genero Business Development Language.

**Important:** This page covers only those new features introduced with the Genero BDL version specified in the page title. Check prior new features pages if you migrate from an earlier version. Make sure to also read the upgrade guide corresponding to this Genero version.

Corresponding upgrade guide: [BDL 2.02 upgrade guide](#) on page 191.

**Table 67: Core language**

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Share global variables between the Genero source and the C Extension, by using the <code>-G</code> option of <code>fglcomp</code>.</td>
<td>Feature is desupported in version 3.20, see <a href="#">Sharing GLOBALS with C Extensions</a> on page 125.</td>
</tr>
<tr>
<td>Customize the runtime system error messages according to the current locale.</td>
<td>See <a href="#">Runtime system messages</a> on page 530.</td>
</tr>
<tr>
<td>New debugger commands (ptype).</td>
<td>See <a href="#">Integrated debugger</a> on page 2142.</td>
</tr>
<tr>
<td>Avoid switching into debug mode with SIGTRAP (Unix) or CTRL-Break (Windows™) with the new <code>fglrun.ignoreDebuggerEvent</code> FGLPROFILE entry.</td>
<td></td>
</tr>
</tbody>
</table>

**Table 68: User interface**

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specify a <code>TABINDEX</code> of zero to exclude the form item from the tagging list.</td>
<td>See <a href="#">TABINDEX attribute</a> on page 1395.</td>
</tr>
</tbody>
</table>
Table 69: SQL databases

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Some common SQL statements have been added to the static SQL syntax,</td>
<td>See Static SQL statements on page 694.</td>
</tr>
<tr>
<td>such as TRUNCATE TABLE, RENAME INDEX, CREATE / ALTER / DROP / RENAME</td>
<td></td>
</tr>
<tr>
<td>SEQUENCE.</td>
<td></td>
</tr>
<tr>
<td>With Oracle, specify the SELECT statement producing the unique session</td>
<td>See Oracle DB specific FGLPROFILE parameters on page 673.</td>
</tr>
<tr>
<td>identifier which is used for temporary table names.</td>
<td></td>
</tr>
<tr>
<td>To emulate Informix® temporary tables in Oracle, set the</td>
<td>See Using the global temporary table emulation on page 947.</td>
</tr>
<tr>
<td>temptables.emulation parameter to use GLOBAL TEMPORARY TABLES</td>
<td></td>
</tr>
<tr>
<td>instead of permanent tables.</td>
<td></td>
</tr>
</tbody>
</table>

BDL 2.01 new features

Features added in 2.01 releases of the Genero Business Development Language.

**Important:** This page covers only those new features introduced with the Genero BDL version specified in the page title. Check prior new features pages if you migrate from an earlier version. Make sure to also read the upgrade guide corresponding to this Genero version.

Corresponding upgrade guide: BDL 2.01 upgrade guide on page 191.

Table 70: Core language

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>The fglcomp compiler now supports a negative form for –w warning arguments.</td>
<td>See Compiling program code files (.4gl) on page 2108.</td>
</tr>
<tr>
<td>When using the RUN command, the ComSpec environment variable is now used</td>
<td>See RUN on page 490.</td>
</tr>
<tr>
<td>under Windows™ platforms.</td>
<td></td>
</tr>
</tbody>
</table>

Table 71: User interface

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>The layout tag syntax in grids has been extended to support an ending tag</td>
<td>See Layout tags on page 1254.</td>
</tr>
<tr>
<td>to get better control of form layout.</td>
<td></td>
</tr>
</tbody>
</table>

Table 72: SQL databases

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support for IBM® DB2® V9.x.</td>
<td>See IBM DB2 Linux-Unix-Windows on page 758.</td>
</tr>
<tr>
<td>Support for PostgreSQL 8.2.x.</td>
<td>See PostgreSQL on page 964.</td>
</tr>
</tbody>
</table>

BDL 2.00 new features

Features added in 2.00 releases of the Genero Business Development Language.

**Important:** This page covers only those new features introduced with the Genero BDL version specified in the page title. Check prior new features pages if you migrate from an earlier version. Make sure to also read the upgrade guide corresponding to this Genero version.

Corresponding upgrade guide: BDL 2.00 upgrade guide on page 191.
### Table 73: Core language

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>The runtime system <em>(fglrune)</em> now uses shared libraries for database</td>
<td>See Database driver specification (driver) on page 662.</td>
</tr>
<tr>
<td>drivers; there is no need to link anymore.</td>
<td></td>
</tr>
<tr>
<td>The <em>TYPE</em> instruction allows to define your own data type structures.</td>
<td>See Types on page 431.</td>
</tr>
<tr>
<td>File management function library provided as loadable extension.</td>
<td>See The os.Path class on page 2792.</td>
</tr>
<tr>
<td>Mathematical function library provided as loadable extension.</td>
<td>See The util.Math class on page 2759.</td>
</tr>
<tr>
<td>C extension support has been extended with Informix-like C API functions.</td>
<td>No longer applicable as of Genero 2.51.</td>
</tr>
<tr>
<td>The runtime system now shares several static elements among all processes,</td>
<td>See Runtime system basics on page 586.</td>
</tr>
<tr>
<td>reducing the memory usage. The shared elements are: Data type definitions,</td>
<td></td>
</tr>
<tr>
<td>string constants and debug information. For example, when a program</td>
<td></td>
</tr>
<tr>
<td>defines a string containing a long SQL statement, all fglrun processes</td>
<td></td>
</tr>
<tr>
<td>will share the same string, which is allocated only once.</td>
<td></td>
</tr>
<tr>
<td>The <em>IMPORT</em> instruction allows to declare a C extension module.</td>
<td>See IMPORT C-Extension on page 481.</td>
</tr>
<tr>
<td>New debugger commands (<em>call, ignore</em>).</td>
<td>See Debugger commands on page 2147.</td>
</tr>
<tr>
<td>The <em>base.Channel</em> class now has an <em>isEof()</em> method to detect end of</td>
<td>See Read and write text lines on page 2426.</td>
</tr>
<tr>
<td>file.</td>
<td></td>
</tr>
<tr>
<td>Ignoring the CTRL_LOGOFF_EVENT events on Microsoft™ Windows® platforms.</td>
<td>See Responding to CTRL_LOGOFF_EVENT on page 566.</td>
</tr>
<tr>
<td>New built-in function to set an environment variable: FGL_SETENV().</td>
<td>See fgl_setenv() on page 2294.</td>
</tr>
<tr>
<td>The XML reader and writer classes have been extended to properly support</td>
<td>See The XmlReader class on page 2625, The XmlWriter class on page 2630.</td>
</tr>
<tr>
<td>markup language entities (like HTML's  ).</td>
<td></td>
</tr>
</tbody>
</table>

### Table 74: User interface

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>New form item types (i.e. widgets): SLIDER, SPINEDIT, TIMEEDIT.</td>
<td>See ATTRIBUTES section on page 1330.</td>
</tr>
<tr>
<td>The WIDTH and HEIGHT attributes can be used for IMAGE form items, as a</td>
<td>See HEIGHT attribute on page 1368, WIDTH attribute on page 1411.</td>
</tr>
<tr>
<td>replacement for PIXELWIDTH/PIXELHEIGHT.</td>
<td></td>
</tr>
<tr>
<td>New debugger commands (<em>call, ignore</em>).</td>
<td>See Debugger commands on page 2147.</td>
</tr>
<tr>
<td>Presentation styles support now pseudo selectors such as focus, active,</td>
<td>See Pseudo selectors on page 1168.</td>
</tr>
<tr>
<td>inactive, input, display for fields and odd/even states for table rows.</td>
<td></td>
</tr>
</tbody>
</table>
New presentation style attributes were added:

- 'errorMessagePosition' can be used for windows to define how the ERROR message must be displayed;
- 'highlightTextColor' for tables allows you to change the color of the selected line;
- 'border' allows you to remove the border of some widgets like button, images;
- 'firstDayOfWeek' can be used for DateEdit widget to specify the first day of the week in the calendar;
- The auto-selection behavior for ComboBoxes and RadioGroup can be changed using 'autoSelectionStart'.

With X11 or Windows® TSE environments, you can now automatically start up the front-end with FGLPROFILE entries.

Up to fourth accelerators can now be defined for an action in actions defaults files or in the ACTION DEFAULTS section of form files.

Specify TTY attributes (COLOR, REVERSE) and conditional TTY attributes (COLOR WHERE) for all type of fields.

### Table 75: SQL databases

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database schema files have been extended to centralize form field definition with the new FIELD item type.</td>
<td>Important: This feature is deprecated in 2.51 and +. See EXECUTE (SQL statement) on page 711, Stored procedures on page 649. See CREATE TABLE on page 705.</td>
</tr>
<tr>
<td>Call database stored procedures with output parameters with the new IN/OUT keywords.</td>
<td>See Data type conversion control on page 477.</td>
</tr>
<tr>
<td>Primary key, foreign key and check constraints can be specified in static SQL CREATE TABLE statements:</td>
<td>Upgrade notes for database drivers. See Database drivers changes on page 194.</td>
</tr>
</tbody>
</table>

```sql
CREATE TABLE t1 (  
  col1 INTEGER PRIMARY KEY,  
  col2 CHAR(2),  
  col3 DATE,  
  FOREIGN KEY (col2) REFERENCES t2(col1)  
  REFERENCES t2(col1)
)
```

The fgl dbsch tool can now extract database tables with LVARCHAR columns. The LVARCHAR type is converted to VARCHAR2(n>255) in the .sch file.
**Table 76: Web Services**

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>You can now choose to use Document Style Service (Doc/Literal) or RPC Literal Style Service (RPC/Literal) with Genero Web Services (GWS), for .NET compatibility and WS-I compatibility (standards defined by the Web Services Interoperability organization).</td>
<td>See Choosing a web services style on page 3558 and Writing a Web server application on page 3348.</td>
</tr>
</tbody>
</table>
| • Document Style Service allows you to exchange complex data structures, such as database tables or word processing documents (MS.Net default)  
  • RPC Literal Style Service is usually used to execute a function, such as a service that returns a stock option |                                                                                                                                             |
| **Note:** RPC/Encoded Style Service (Traditional SOAP section 5) is available for backward compatibility.                                                                                                 |                                                                                                                                             |
| Genero Web Services now provides a tool, *fglwsdl*, to allow a Genero application that is accessing a Web Service to obtain the WSDL information for the service. It does not matter what language the Web Service is written in. The *fglwsdl* tool is installed in Genero as part of the Genero Web Services package. | See *fglwsdl* on page 2081.                                                                                                                |
| You no longer need to create a runner that includes the Genero Web Services package. Instead, your applications import the Genero Web Services library named *com*. This library provides classes and methods that allow you to perform tasks associated with creating GWS Servers and Clients, and managing the Web Services. | See The *com* package on page 2811.                                                                                                         |
| GWS now supports SOAP header management through the *CreateHeader* method in the Web Service class that is part of the Web Services library (*com*). | See The *WebService* class on page 2811.                                                                                                    |
| HTTPS support has been added on the client side. GWS supports secure communications through the use of encryption and standard X.509 certificates. Based on the OpenSSL engine, new security features allow a Web Services client to communicate with any secured server over HTTP or HTTPS. | See *fglpss* on page 2087, Encryption, BASE64 and password agent with *fglpss* tool on page 3290, and The FGLPROFILE file(s) on page 255. |
| A new tool is provided, *fglpss*, allowing you to encrypt a password from a standard X.509 certificate, and to decrypt a password you previously encrypted with a certificate. |                                                                                                                                             |
| Entries in the FGLPROFILE file are used to define the configuration for client security.                                                                                                                                                                      |                                                                                                                                             |
| You can configure a GWS Client to connect via an HTTP proxy by adding an entry in the FGLPROFILE file.                                                                                                                                                     | See Configure a WS client to connect via an HTTP Proxy on page 3340.                                                                      |
| You can define multiple Web Services in a single Genero DVM. When you start the Web Services engine, all registered Web Services are started.                                                                                                                      | See The *WebServiceEngine* class on page 2828.                                                                                               |
| You can remap the location of Genero Web Services using entries in the FGLPROFILE file, depending on the network configuration and the access rights management of the deployment site.                                                                                                                                  | See Using logical names for service locations on page 3334.                                                                                   |
| Serializing Genero data types: you can add optional attributes to the definition of data types. You can use these attributes to map the BDL data types in a Genero Web Services Client or Server application to their corresponding XML data types. | See Attributes to customize XML serialization on page 3523.                                                                                 |
The WSHelper.42m library included in the $FGLDIR/lib directory of the Genero Web Services package file contains internal BDL functions to handle SOAP requests and errors.

It is recommended that it is linked into every Genero Web Services Server or Client program.

Upgrade notes for web services.

See Compiling the client application on page 3324 and Compiling GWS server applications on page 3355.

Table 77: Core language

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>New <code>base TypeInfo</code> built-in class to serialize program variables.</td>
<td>See The TypeInfo class on page 2465.</td>
</tr>
<tr>
<td>The <code>base.Channel</code> class now supports a binary mode with the <code>\b</code> option, to control CR/LF translation when using DOS files.</td>
<td>See Line terminators on Windows and UNIX on page 2427.</td>
</tr>
</tbody>
</table>

Table 78: User interface

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up to three accelerators can now be defined for an action in actions defaults files or in the ACTION DEFAULTS section of form files.</td>
<td>See Defining keyboard accelerators for actions on page 1749.</td>
</tr>
</tbody>
</table>

Table 79: SQL databases

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic ODBC database driver is now available (code is generic ODBC database driver is now available (code is odc).</td>
<td>See Database driver specification (driver) on page 662.</td>
</tr>
<tr>
<td>MySQL version 5.0.x is now supported.</td>
<td>See Oracle MySQL / MariaDB on page 876.</td>
</tr>
<tr>
<td>PostgreSQL version 8.1.x is now supported.</td>
<td>See PostgreSQL on page 964.</td>
</tr>
<tr>
<td>Microsoft™ SQL Server 2005 is now supported.</td>
<td>See Microsoft SQL Server on page 827.</td>
</tr>
<tr>
<td>Pre-fetch rows by block with SQL Server to get better performance. Use the following <code>FGLPROFILE</code> entry to specify the maximum number of rows the driver can pre-fetch: <code>dbi.database.dbname.msv.prefetch.rows = count</code></td>
<td>See SQL Server (MS ODBC) specific FGLPROFILE parameters on page 676.</td>
</tr>
</tbody>
</table>

See "Database vendor specific parameters" in Connections for more details.
### BDL 1.32 new features
Features added in 1.32 release of the Genero Business Development Language.

**Important:** This page covers only those new features introduced with the Genero BDL version specified in the page title. Check prior new features pages if you migrate from an earlier version. Make sure to also read the upgrade guide corresponding to this Genero version.

Corresponding upgrade guide: [BDL 1.32 upgrade guide](#) on page 201.

#### Table 80: Core language

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>New debugger commands (<code>watch</code> with condition, <code>whatis</code>).</td>
<td>See <a href="#">Debugger commands</a> on page 2147.</td>
</tr>
<tr>
<td>The preprocessor is now part of the compilers and is always enabled. Preprocessing directives start with an ampersand character (<code>&amp;</code>).</td>
<td>See <a href="#">Source preprocessor</a> on page 2132.</td>
</tr>
</tbody>
</table>

#### Table 81: User interface

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>New built-in functions to transfer files from/to the front-end.</td>
<td>See <code>fgl_getfile()</code> on page 2286, <code>fgl_putfile()</code> on page 2291.</td>
</tr>
</tbody>
</table>

#### Table 82: SQL databases

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL version 8.0 is now supported (8.0.2 and higher).</td>
<td>See <a href="#">PostgreSQL</a> on page 964.</td>
</tr>
</tbody>
</table>

### BDL 1.31 new features
Features added in 1.31 release of the Genero Business Development Language.

**Important:** This page covers only those new features introduced with the Genero BDL version specified in the page title. Check prior new features pages if you migrate from an earlier version. Make sure to also read the upgrade guide corresponding to this Genero version.

Corresponding upgrade guide: [BDL 1.31 upgrade guide](#) on page 201.

#### Table 83: Core language

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>C extensions can be loaded dynamically, no need to re-link runner.</td>
<td>See <a href="#">C-Extensions</a> on page 2230.</td>
</tr>
<tr>
<td>The <code>FGL_WIDTH()</code> built-in function computes the number of print columns needed to represent a single or multibyte character.</td>
<td>See <code>fgl_width()</code> on page 2296.</td>
</tr>
</tbody>
</table>
Table 84: User interface

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>GUI protocol compression for slow networks.</td>
<td>See GUI protocol compression on page 1119.</td>
</tr>
<tr>
<td>Interruption handling with SSH port forwarding - only supported with GDC 1.31!</td>
<td>See User interruption handling on page 1711.</td>
</tr>
<tr>
<td>New method <code>ui.Form.setFieldStyle()</code> to set a style for a field.</td>
<td>See <code>ui.Form.setFieldStyle</code> on page 2509.</td>
</tr>
<tr>
<td>Improved front-end identification when connecting to GUI client.</td>
<td>See GUI front-end connection on page 1116.</td>
</tr>
</tbody>
</table>

Table 85: SQL databases

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL version 4.1.x is now supported, 3.23 is desupported.</td>
<td>See Oracle MySQL / MariaDB on page 876.</td>
</tr>
<tr>
<td>Oracle version 10g is now supported.</td>
<td>See Oracle Database on page 908.</td>
</tr>
</tbody>
</table>

BDL 1.30 new features

Features added in 1.30 releases of the Genero Business Development Language.

**Important:** This page covers only those new features introduced with the Genero BDL version specified in the page title. Check prior new features pages if you migrate from an earlier version. Make sure to also read the upgrade guide corresponding to this Genero version.

Corresponding upgrade guide: **BDL 1.30 upgrade guide** on page 201.

Table 86: Core language

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>First version of integrated preprocessor using <code>#</code> hash syntax for macros. Version 1.32 uses <code>&amp;</code> instead</td>
<td>See Source preprocessor on page 2132.</td>
</tr>
<tr>
<td>Localization support (multibyte character sets).</td>
<td>See Localization on page 512.</td>
</tr>
<tr>
<td>The <code>fglcomp</code> compiler now adds build information in 42m modules.</td>
<td>See 42m module information on page 2120.</td>
</tr>
<tr>
<td>Compiler version of a 42m module can be checked on site by using the <code>fglr run</code> with the <code>-b</code> option (line break added for documentation only):</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>$ fglrun -b module.42m</code></td>
<td></td>
</tr>
<tr>
<td>2004-05-17 10:42:05 1.30.2a-620.10</td>
<td></td>
</tr>
<tr>
<td>/devel/tests/module.4gl</td>
<td></td>
</tr>
<tr>
<td>The <code>fglmkmsg</code> tool now has the same behavior as other tools like <code>fglcomp</code> and <code>fglform</code>: If you give only the source file, the message compiler uses the same file name for the compiled output file, adding the <code>.iem</code> extension.</td>
<td></td>
</tr>
<tr>
<td>New <code>BREAKPOINT</code> instruction to stop a program at a given position when using the debugger. It is ignored when not running in debug mode.</td>
<td></td>
</tr>
<tr>
<td>New assignment operator <code>:=</code> has been added to the language. Assign variables directly within expressions: <code>IF (i:=(j+1)) ==2 THEN</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Compiling message files (.msg) on page 1161.</td>
</tr>
<tr>
<td></td>
<td>See Setting a breakpoint programmatically on page 2146.</td>
</tr>
<tr>
<td></td>
<td>See Assignment (:=) on page 366.</td>
</tr>
</tbody>
</table>
### Overview

**New fglcomp compiler option to detect non-standard SQL syntax:**

```bash
glcomp -W stdsql module.4gl
```

See SQL portability on page 607.

**New method base.StringBuffer.replace(), to replace a sub-string in a string:**

```sql
CALL s.replace("old","new",2)
```

Replaces two occurrences of "old" with "new"...

See base.StringBuffer.replace on page 2455.

**New methods to read/write complete lines in base.Channel built-in class:**

- `readLine()`
- `writeLine()`

See Read and write text lines on page 2426.

**The FGLLDPATH variable is now used during program linking.**

See Compiling source files on page 2106.

**The linker option --O (optimize) is de-supported (was ignored before). You now get a warning if you use this option.**

See Linking programs on page 2115.

**The [] array sub-script operator now returns the sub-array:**

```sql
DEFINE a2 DYNAMIC ARRAY
    WITH DIMENSION 2 OF INTEGER
LET a2[5,10] = 123
DISPLAY a2.getLength() -- displays 5
DISPLAY a2[5].getLength() -- displays 10
```

See Arrays on page 418.

### Table 87: User interface

**Overview**

New layout rules and form item attributes provide better control of form design.

Decoration attribute can be defined in a presentation style file to set fonts and colors.

Action defaults can be specified in forms in the ACTION DEFAULTS section.

New ui.Dialog built-in class to provide better control over interactive instructions.

COMBOBOX fields now support UPSHIFT and DOWNSHIFT attributes, to force character case when QUERYEDITABLE is used.

New presentation style attribute highlightCurrentRow for Tables, to indicate if the current row must be highlighted in a specific mode. By default, the current row is highlighted during a DISPLAY ARRAY.

New method appendElement() for ARRAYS, to append an element at the end of a dynamic array.

New assignment operator := has been added to the language. Assign variables directly within expressions: IF (i:=(j+1))==2 THEN

The new method ui.Dialog.setCellAttributes() allows you to define colors for each cell of a table.

The ui.Window class provides new methods to create or get a form object.

**Reference**

See Form rendering on page 1415.

See Presentation styles on page 1165.

See ACTION DEFAULTS section on page 1296.

See The Dialog class on page 2512.

See QUERYEDITABLE attribute on page 1387.

See Table style attributes on page 1204.

See Array methods on page 424.

See Assignment (:=) on page 366.

See Cell color attributes on page 1850.

See ui.Window methods on page 2490.
<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>When using a dynamic array in <code>INPUT ARRAY</code> or <code>DISPLAY ARRAY</code>, the number of rows is defined by the size of the dynamic array. The <code>SET_COUNT()</code> or <code>COUNT</code> attributes are ignored.</td>
<td>See Controlling the number of rows on page 1834.</td>
</tr>
<tr>
<td>The new form field attribute <code>TITLE</code> can be used to specify a table column label with a localized string.</td>
<td>See <code>TITLE</code> attribute on page 1397.</td>
</tr>
<tr>
<td>New class method <code>ui.Dialog.setDefaultUnbuffered()</code> to set the default for the <code>UNBUFFERED</code> mode.</td>
<td>See The buffered and unbuffered modes on page 1720.</td>
</tr>
<tr>
<td>Action defaults are now applied at element creation by the runtime system. In previous versions this was done dynamically by the front-end. Now, changing an action default node at runtime has no effect on existing elements.</td>
<td>See Configuring actions on page 1744.</td>
</tr>
<tr>
<td>The <code>DATEEDIT</code> field type now supports <code>DBDATE/CENTURY</code> settings and the <code>FORMAT</code> attribute.</td>
<td>See <code>FORMAT</code> attribute on page 1366.</td>
</tr>
<tr>
<td>New default action 'close' to control window closing:</td>
<td>See Implementing the close action on page 1772.</td>
</tr>
<tr>
<td><code>INPUT ARRAY</code> using <code>TABLE</code> container now needs <code>FIELD ORDER FORM</code> attribute to keep tabbing order consistent with visual order of columns.</td>
<td>See Defining the tabbing order on page 1729.</td>
</tr>
<tr>
<td>New instructions <code>ACCEPT INPUT / ACCEPT CONSTRUCT / ACCEPT DISPLAY</code> to validate a dialog by program.</td>
<td>See ACCEPT INPUT instruction on page 1495, ACCEPT DISPLAY instruction on page 1526, ACCEPT CONSTRUCT instruction on page 1579.</td>
</tr>
<tr>
<td>New dialog attribute <code>ACCEPT / CANCEL</code> to avoid creation of default actions 'accept' and 'cancel'.</td>
<td>See INPUT instruction configuration on page 1486.</td>
</tr>
<tr>
<td>New default action 'append' in <code>INPUT ARRAY</code>. Allows you to add a row at the end of the list.</td>
<td>See Default actions in <code>INPUT ARRAY</code> on page 1538.</td>
</tr>
<tr>
<td>New method <code>ui.Window.createForm()</code> to create an empty form object in order to build forms from scratch at runtime.</td>
<td>See <code>ui.Window.createForm</code> on page 2492.</td>
</tr>
<tr>
<td><code>TOPMENU</code> definition in forms now allows attributes in parenthesis.</td>
<td>See <code>TOPMENU</code> section on page 1297.</td>
</tr>
<tr>
<td>The form layout syntax now allows you to specify the real width of form items by using a dash '-' in the layout tag.</td>
<td>See Widget size within hbox tags on page 1430.</td>
</tr>
<tr>
<td>Important remark: Before build 530 the <code>MENU</code> has attached the window when returning from the <code>BEFORE MENU</code> actions. Since build 530 the <code>WINDOW</code> must exist before the <code>MENU</code> statement. So now the <code>Menu AUI tree node</code> is available in the <code>BEFORE MENU</code> block, but a window opened or made current in the <code>BEFORE MENU</code> block will NOT be used.</td>
<td></td>
</tr>
<tr>
<td>Layout <code>GRID</code> now accepts HBox tags to group items horizontally.</td>
<td>See Hbox tags on page 1261.</td>
</tr>
<tr>
<td>Form <code>VERSION</code> attribute to distinguish form revisions.</td>
<td>See <code>VERSION</code> attribute on page 1406.</td>
</tr>
<tr>
<td>Form layout <code>SPACING</code> attribute to define space between widgets.</td>
<td>See <code>SPACING</code> attribute on page 1393.</td>
</tr>
</tbody>
</table>
### Overview

**Form** `DEFAULT SAMPLE` instruction to define a default sample attribute for all form fields.

New form item attributes: `SAMPLE`, `JUSTIFY`, `SIZEPOLICY` ...

To hide form elements by default, that can be shown by the end user by option, use `HIDDEN=USER` as 'hidden to the user by default'.

Individual table columns now have new attribute `UNMOVABLE` to avoid moving.

`WANTCOLUMNSANCHORED` replaced by `UNMOVABLECOLUMN` and `WANTCOLUMNSVISIBLE` replaced by `UNHIDABLECOLUMNNS`.

Tables now accept a `WIDTH` and `HEIGHT` attribute to specify a size.

New `RADIOGROUP` attribute to define the orientation of the radio buttons: `ORIENTATION`.

The `MENU` `COMMAND` clause now generates action names in lowercase. This means, when you define `COMMAND "Open"`, it will bind to all actions views defined with the name 'open'.

New `ui.Interface.loadTopMenu()` method to load a global `TOPMENU`.

The `ON CHANGE` block is now invoked when the user clicks on a `CHECKBOX`, `RADIOGROUP`, or changes the item in a `COMBOBOX`.

New `DIALOG` keyword to reference the current dialog as a `ui.Dialog` object. This can be used for example to enable/disable fields during the dialog execution.

The `ui.Form` built-in class has new methods to handle form elements. The hidden attribute is now also managed at the model level, this allows you to hide form fields by name, instead of using the decoration node.

```lua
CALL myform.setElementHidden("formonly.field1",2)
CALL myform.setFieldHidden("field1",2)
-- prefix is optional
```

New methods are provided in `ui.Interface` to control the MDI children.

In `INPUT ARRAY`, `CANCEL INSERT` now supported in `AFTER INSERT`, to remove the new added line when needed.

`TOOLBAR` and `TOPMENU` elements now have the hidden attribute so you can create them and hide the options the user is not supposed to see.

**Important**: Hiding a toolbar or topmenu option does not prevent the use of the accelerator of the action. Use `ui.Dialog.setActionActive()` to disable an action.

### Reference

See `INSTRUCTIONS` section on page 1350.

See `SAMPLE attribute` on page 1389, `JUSTIFY attribute` on page 1377, `SIZEPOLICY attribute` on page 1391.

See `HIDDEN attribute` on page 1369.

See `UNMOVABLE attribute` on page 1399.

See `UNMOVABLECOLUMN` attribute on page 1400, `UNHIDABLECOLUMNNS` attribute on page 1398.

See `WIDTH attribute` on page 1411, `HEIGHT attribute` on page 1368.

See `ORIENTATION attribute` on page 1383.

See `COMMAND [KEY()] "option" block` on page 1476.

See `ui.Interface.loadTopMenu` on page 2484.

See `ON CHANGE block` on page 1492.

See `The Dialog class` on page 2512.

See `The Form class` on page 2496.

See `Window containers (WCI)` on page 2023.

See `CANCEL INSERT instruction` on page 1556.

See `ui.Form.setElementHidden` on page 2506.
New option NEXT FIELD CURRENT to gives control back to the dialog instruction without moving to another field.

See Giving the focus to a form element on page 1731.

### Table 88: SQL databases

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support for PostgreSQL 7.4 with parameterized queries.</td>
<td>See PostgreSQL on page 964.</td>
</tr>
<tr>
<td>A MySQL 3.23 driver is now provided for Windows™ platforms (was previously only provided on Linux®).</td>
<td>See Oracle MySQL / MariaDB on page 876.</td>
</tr>
</tbody>
</table>
| The fglcomp compiler now converts static SQL updates like:

```
UPDATE tab SET (c1,c2)=(v1,c2) ...
```

| to a standard syntax:

```
UPDATE tab SET c1=v1, c2=v2 ...
```

| On Windows™ platforms only, the ix drivers automatically set standard Informix® environment variables with ifx_putenv(). Values are taken from the console environment with getenv(). Additional variables can be specified with:

```
dbi.stdifx.environment.count = n
dbi.stdifx.environment.xx = "variable"
```

| No more available in recent versions. |

### BDL 1.20 new features

Features added in 1.20 releases of the Genero Business Development Language.

**Important:** This page covers only those new features introduced with the Genero BDL version specified in the page title. Check prior new features pages if you migrate from an earlier version. Make sure to also read the upgrade guide corresponding to this Genero version.

Corresponding upgrade guide: BDL 1.20 upgrade guide on page 210.

### Table 89: Core language

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integrated debugger with gdb syntax to interface with graphical tools like ddd.</td>
<td>See Integrated debugger on page 2142.</td>
</tr>
<tr>
<td>The program profiler can be used to generate statistics of program execution, to find the bottlenecks in the source code.</td>
<td>See Program profiler on page 2165.</td>
</tr>
<tr>
<td>Internationalize your application in different languages with localized strings, by using the %&quot;string&quot; notation.</td>
<td>See Localized strings on page 538.</td>
</tr>
<tr>
<td>The TERMINATE REPORT and EXIT REPORT can be used in reports to respectively stop a report from outside of the REPORT routine, or stop the report from inside the REPORT routine.</td>
<td>See TERMINATE REPORT on page 2038, EXIT REPORT on page 2052.</td>
</tr>
<tr>
<td>The fgl_getversion() function returns the version number of the runtime system.</td>
<td>See fgl_getversion() on page 2288.</td>
</tr>
</tbody>
</table>
### Overview

| Static arrays can be passed as parameters: all elements are expanded. | See [Static arrays on page 420](#). |
| New methods for `StringBuffer` class: `base.StringBuffer.replaceAt()` and `base.StringBuffer.insertAt()` | See [The StringBuffer class on page 2447](#). |
| Operators equal (= or ==) and not equal (<> or !=) now can be used with records: All record members will be compared. If two members are NULL the result of this member comparison results in TRUE. | See [RECORD on page 409](#). |
| New `-W` option for `fglform` to show warnings. | See [fglform on page 2068](#). |
| `LSTR()` operator, to get a localized string by name. Useful when the localized string identifier is known at runtime only. | See [LSTR() on page 360](#). |
| `SFMT()` operator, to format strings with parameter placeholders. Useful to localize application messages with parameters. | See [SFMT() on page 360](#). |
| The `base.StringTokenizer` class can be used to parse strings for tokens. | See [The StringTokenizer class on page 2462](#). |
| `CONSTANT` language elements can now be defined as `GLOBALS`. | See [Constants on page 405](#). |
| The `base.Application` class provides an interface to the program properties. | See [The Application class on page 2406](#). |
| Review of the definition of `base.Channel` class, now based on objects. | See [The Channel class on page 2412](#). |

### Table 90: User interface

| Interactive instructions support the UNBUFFERED mode, to synchronise data model and view automatically: When you set a variable, the value is automatically displayed to the field, and when the user fires and action, the field value is automatically assigned to the corresponding program variable. | See [The buffered and unbuffered modes on page 1720](#). |
| `DISPLAY ARRAY` can now work in paged mode, to avoid loading a large array of rows, with the `ON FILL BUFFER` clause. | See [Paged mode of DISPLAY ARRAY on page 1843](#). |
| Centralize default attributes for actions in `ACTION DEFAULTS`. | See [Configuring actions on page 1744](#). |
| Client side settings can now be saved by application name, with a specific API. By default it is the name of the program. | See [ui.Interface.setName on page 2486](#). |
| New attribute `APPEND ROW = TRUE/FALSE` attribute for the `INPUT ARRAY` instruction, to control the creation of the default append action. | See [INPUT ARRAY row modifications on page 1847](#). |
| New attribute `KEEP CURRENT ROW = TRUE/FALSE` for the `DISPLAY ARRAY` and `INPUT ARRAY` instructions, to defines if the current row must remain highlighted when leaving the dialog. The default is `FALSE`. | See [Handling the current row on page 1835](#). |
| You can now define a `TOOLBAR` in form specification files. | See [TOOLBAR section on page 1298](#). |
| You can now define a `TOPMENU` in form specification files. | See [TOPMENU section on page 1297](#). |
| The `fgl_gethelp()` function returns the help text for the given help number. | See [fgl_gethelp() on page 2285](#). |
### Overview

<table>
<thead>
<tr>
<th>The <code>fgl_set_arr_curr()</code> function changes the current row in DISPLAY ARRAY or INPUT ARRAY. Users can now send an interruption event to the program, to stop long running SQL queries, processing loops and reports. The <code>statusBarType</code> window style attribute to define the statusbar layout. The new FIELD ORDER FORM option can be used to follow the new TABINDEX attribute, to define the field tabbing order. FIELD ORDER FORM can also be used at the dialog level as dialog attribute. For COMBOBOX form items, a default ITEMS list is created by fglform when an INCLUDE attribute is used. The ON IDLE clause can be used to execute a block of instructions after a timeout. New logical order of execution for INPUT ARRAY triggers: 1. BEFORE INPUT 2. BEFORE ROW 3. BEFORE INSERT 4. BEFORE FIELD</th>
<th>See Handling the current row on page 1835. See User interruption handling on page 1711. See Window style attributes: Miscellaneous on page 1225. See Defining the tabbing order on page 1729. See COMBOBOX item type on page 1270. See Get program control if user is inactive on page 1712. See Editable record list (INPUT ARRAY) on page 1531.</th>
</tr>
</thead>
<tbody>
<tr>
<td>New ui.ComboBox class to configure COMBOBOX fields at runtime. DISPLAY ARRAY and INPUT ARRAY instructions now automatically use two predefined actions <code>nextrow</code> and <code>prevrow</code>, which allow binding action views for navigation. ON CHANGE field trigger can be used to detect field modification. Useful for fields such as CHECKBOX and COMBOBOX. Program icon definition with <code>ui.Interface.setImage()</code>. LABEL fields can now have a FORMAT attribute. Front-end function calls allow to execut code on the front-end side with the <code>ui.Interface.frontCall()</code> method. New ui.Form built-in class to handle forms. New ON ROW CHANGE clause in INPUT ARRAY, executed when if at least one value in the row has been modified, and the user moves to another row or validates the dialog. The ON ROW CHANGE block is executed before the AFTER ROW block. MENU instruction now supports ON ACTION clause, to write abstract menus as simple action handlers. New 'help' predefined action, to start help viewer for HELP clauses in dialog instructions.</td>
<td>See The ComboBox class on page 2561. See Predefined actions on page 1740. See Reacting to field value changes on page 1724. See ui.Interface.setImage on page 2485. See LABEL item type on page 1280. See Front calls on page 583. See The Form class on page 2496. See ON ROW CHANGE block on page 1544. See Ring menus (MENU) on page 1467. See Predefined actions on page 1740.</td>
</tr>
</tbody>
</table>
### Table 91: SQL databases

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL Server driver now supports the TINYINT data type.</td>
<td>See Numeric data types on page 842.</td>
</tr>
<tr>
<td>The fglcomp compiler supports now ANSI outer join syntax in SQL statements (LEFT OUTER JOIN), to replace the Informix specific OUTER() syntax.</td>
<td>See SELECT on page 700.</td>
</tr>
<tr>
<td>FOREACH that raises an error no longer loops infinitely.</td>
<td>See FOREACH (result set cursor) on page 724.</td>
</tr>
<tr>
<td>New SQLSTATE and SQLERRMESSAGE registers, to give SQL execution information.</td>
<td>See SQL error identification on page 595.</td>
</tr>
</tbody>
</table>

### BDL 1.10 new features

Features added in 1.10 releases of the Genero Business Development Language.

**Important:** This page covers only those new features introduced with the Genero BDL version specified in the page title. Check prior new features pages if you migrate from an earlier version. Make sure to also read the upgrade guide corresponding to this Genero version.


### Table 92: Core language

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>The language supports now built-in classes, a new object-oriented way to program in BDL.</td>
<td>See OOP support on page 570.</td>
</tr>
<tr>
<td>CONSTANT keyword to define constants in your programs.</td>
<td>See Constants on page 405.</td>
</tr>
<tr>
<td>The language now supports dynamic arrays with automatic memory allocation.</td>
<td>See Dynamic arrays on page 422.</td>
</tr>
<tr>
<td>A set of XML Utilities are provided in the runtime library as built-in classes.</td>
<td>See The om package on page 2577.</td>
</tr>
<tr>
<td>The STRING data type can be used to manipulate character strings without a length limit as with CHAR/VARCHAR.</td>
<td>See STRING on page 305.</td>
</tr>
</tbody>
</table>

### Table 93: User interface

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Dynamic User Interface is the major new concept in Genero. It is the basement for the new graphical user interface.</td>
<td>See User interface basics on page 1105.</td>
</tr>
<tr>
<td>Compared to classic IBM Informix 4gl, interactive instructions such as INPUT, DISPLAY ARRAY, have been extended with new control blocks and control instructions.</td>
<td>See Dialog instructions on page 1452.</td>
</tr>
<tr>
<td>Form specification files (.per) support now extended layout definition with the LAYOUT section.</td>
<td>See Form definitions on page 1131.</td>
</tr>
<tr>
<td>Defining Window Containers (a.k.a. MDI) is a simple way to group programs.</td>
<td>See Window containers (WCI) on page 2023.</td>
</tr>
</tbody>
</table>
### Upgrade Guides for Genero BDL

Provides information about product changes to be considered when migrating to a new version.

**Important:** Each upgrade guide is an incremental upgrade guide that covers only topics related to a specific version of Genero. It is important that you read all of the upgrade guides that sit between your existing version and the desired version.

### General BDL upgrade guide

These topics describe general considerations when upgrading to a new version of Genero BDL.

**Runtime system and front-end compatibility**

For better compatibility and GUI related bug fixes, use front-end and runtime system with the equivalent version number.

Graphical User Interface new features and bug fixes usually require modifications inside the runtime system (`fglrun`) and front-ends (GDC, GBC, GMA, GMI).

When upgrading the runtime system to the latest version, we strongly recommend that you upgrade to the latest front-end version as well. For example, when upgrading to a runtime system 2.50.xx, upgrade front-ends to the latest available 2.50.xx version.

**P-Code compatibility across versions**

P-Code incompatibility (within .42m files) may be introduced from version to version.

Recompilation is only needed when the p-code becomes incompatible. When executing a program with an older p-code version as expected, `fglrun` will raise the error -6201.

Recompile your `.4gl` sources when upgrading to a new features release. Recompilation is not required when upgrading to a bug-fix release.

Feature and bug-fix releases are distinguished by the product version number. The product version number can be found by executing the `fglrun` command with the `-V` option.

The product version number has the following format: `M.FF.BB`, where `M` stands for the major release number, `FF` is a feature number, and `BB` is the bug-fix number. For example: `2.31.14`.

- A new feature release is identified by the `M.FF` part of the product version number.
- A bug-fix release is identified by the `BB` part of the product version number.

For example, you must recompile your sources when upgrading from 2.50 to 3.00, from 2.40 to 2.50, or from 2.51 to 2.52.

**Related concepts**

- [Compiling source files](#) on page 2106
  Describes how to build the runtime files from source files.

### BDL 3.20 upgrade guide

These topics describe product changes you must be aware of when upgrading to version 3.20.

**Important:** This is an incremental upgrade guide that covers only topics related to the Genero BDL version specified in the page title. Check prior upgrade guides if you migrate from an earlier version. Make sure to also read about the new features for this Genero version.

### Table 94: SQL databases

<table>
<thead>
<tr>
<th>Overview</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>The new <code>SCHEMA</code> instruction allows you to specify a database schema, without having an implicit connection, when the program executes.</td>
<td>See Database schema on page 467.</td>
</tr>
</tbody>
</table>
Web Services changes
There are changes in support of web services in Genero 3.20.

XML serializer is case sensitive
From FGLGWS 3.20 on, the XML serializer is case sensitive, same as the JSON serializer. This means that the serializer uses the case of the variable name as defined in the 4GL file.

What impact has this on my existing Web services? It is recommended to:

1. Generate a new client and server stub via the fglwsdl tool.
   - The tool has an enhancement which does not generate XMLName attributes, unless they are really needed.
   - If you do not generate the stub, the XMLName attribute is still taken into account, so it should not have any impact.

2. Check on server sides that all published SOAP web services input and output records are in lower case. This ensures compatibility with older versions. If backward compatibility is not an issue, then you must query the server WSDL file and generate the client stub from it.

For more details, see Customizing XML serialization on page 3538.

Control HTTP Date header for GET, HEAD and DELETE requests
Starting with GWS 3.20.00, the FGLPROFILE entry http.global.request.date can be used to control whether GWS HTTP GET, HEAD and DELETE requests must send the HTTP Date header.

For more details, see HTTP configuration on page 3495.

Support for empty HTTP POST or PUT requests
GWS 3.20.01 allows now to POST or PUT empty requests and read them on server side:

```plaintext
DEFINE req com.HTTPRequest
LET req = com.HTTPRequest.Create("http://tempuri.org")
CALL req.setMethod("POST") # or PUT
CALL req.doRequest()
```

Prior to GWS 3.20.01, the doRequest() call returned error code -15555 and the error message "Unsupported request-response feature".

The server side can now read empty POST or PUT requests as well. For instance in the following code sample, the return value of readTextRequest() will be NULL:

```plaintext
DEFINE ser com.HTTPServiceRequest
DEFINE txt STRING
LET ser = com.WebServiceEngine.getHttpServiceRequest(-1)
IF ser.getMethod() == "POST" THEN
   LET txt = ser.readTextRequest()
   IF txt IS NULL THEN
      DISPLAY "No body"
   END IF
END IF
```

Notice that the Content-Type is still checked and may raise an error, if it doesn't match. So using readXmlRequest() must have a Content-Type of XML format, even if the body is empty. In practice, it is recommended to check the Content-Type before doing any read operation.

For more details, see com.HTTPRequest.doRequest on page 2879.
New high-level RESTful framework
Genero BDL 3.20 provides a new framework for RESTful Web services programming.
See RESTful Web services with high-level framework on page 3375.

Changes to security.global.protocol in FGLPROFILE
Starting with FGLGWS 3.20.07 GWS secured communication is based on the OpenSSL 1.1 engine. This version of OpenSSL always selects the security protocol. It no longer allows you to specify a specific Transport Layer Security (TLS) or Secure Sockets Layer (SSL).
The security.global.protocol entry in the fglprofile file is therefore no longer supported.
For instance, if you have set security.global.protocol = "TLSv1.2" to configure OpenSSL to use TLSv1.2 for HTTPS for earlier versions, you may encounter the following error message in your Web service:
OpenSSL 1.1 doesn't support specific protocol anymore
It is therefore recommended to remove the security.global.protocol entry from your fglprofile file. For more information on Web service security configuration see Web Services FGLPROFILE configuration on page 3489.

XMLChoice attribute nested option
The fglwsl tool now supports the attribute XMLChoice="nested" option to produce an XML representation supporting a substitutionGroup. A substitutionGroup in an XML schema is where one XML element can be replaced by another element that has a substitutionGroup with the same name.
This change applies if you have previously used XMLChoice on page 3544 for a WSDL containing a substitutionGroup. In the following XSD schema example, the "name" element, can be represented as a simple string type (simple-name) or as a more complex type (full-name).

```
<xs:element name="name" />
<xs:element name="simple-name" type="xs:string" substitutionGroup="name" />
<xs:element name="full-name" substitutionGroup="name">
  <xs:complexType>
    <xs:all>
      <xs:element name="first" type="xs:string" minOccurs="0"/>
      <xs:element name="middle" type="xs:string" minOccurs="0"/>
      <xs:element name="last" type="xs:string"/>
    </xs:all>
  </xs:complexType>
</xs:element>
<xs:element name="Identity"/>
<xs:sequence>
  <xs:element ref="name"/>
</xs:sequence>
</xs:element>
```

Previous versions using XMLChoice
In previous versions, the generated stub for a record with XMLChoice generated only one value for all available choices.

```
DEFINE Identity RECORD ATTRIBUTE(XMLSequence,XMLName="Identity")
  name RECORD ATTRIBUTE(XMLOptional),
  _SELECTOR_ SMALLINT ATTRIBUTE(XMLOptional),
  simple_name STRING,
  full_name RECORD ATTRIBUTE(XMLAll)
  first STRING ATTRIBUTE(XMLOptional),
  middle STRING ATTRIBUTE(XMLOptional),
```
**New version, using XMLChoice="nested"**

In the new version, by setting XMLChoice="nested" on the variable, you can choose at runtime one of the available options; name, simple-name, or full-name.

**Note:** In the example the first name (the substitutionGroup tag), is not serialized in XML.

The attribute can be used with a BDL RECORD defined to specify the XML representation for the schema as in the example:

```generated
DEFINE Identity RECORD ATTRIBUTE(XMSequence,XMLName="Identity")
    name RECORD ATTRIBUTE(XMChoice="nested")
        _SELECTOR_ SMALLINT ATTRIBUTE(XMSelector),
        name STRING,
        simple_name STRING ATTRIBUTE(XMLName="simple-name"),
        full_name RECORD ATTRIBUTE(XMLAll,XMLName="full-name")
            first STRING ATTRIBUTE(XMOptional),
            middle STRING ATTRIBUTE(XMOptional),
            last STRING
    END RECORD
END RECORD
END RECORD
```

This is an example of the XML representation when the _SELECTOR_ value is 3:

```xml
<Identity>
    <full-name>
        <first>John</first>
        <last>Smith</last>
    </full-name>
</Identity>
```

This is an example of the XML representation when the _SELECTOR_ value is 2:

```xml
<Identity>
    <simple-name>John Smith</simple-name>
</Identity>
```

**Important:** If in versions prior to GWS 3.20 you had a variable that was expected to serialize an XMLChoice value, you can set XMLChoice="nested" attribute on that variable to improve the serialization options and to avoid serializer errors.

It is therefore recommended to do the following:

- Recompile the Genero Web Service server to create a new WSDL that supports the XML XMLChoice="nested" feature.
- Regenerate all Genero Web Service client stubs from the newly-generated WSDL to get the support of XMLChoice="nested". Regenerate client stubs using the fglwsdl tool.

**Related concepts**

- **Web services** on page 3270
Create a Web service client or server with Genero BDL.

**Genero Mobile for Android™ (GMA) 1.40 changes**

Modifications to consider when using the Genero Mobile for Android™.

**Note:** This topic describes feature changes in the GMA 1.40 product. See also the Mobile section in Genero BDL 3.20 New Features page.
GMA 1.40 with FGLGWS 3.20

**Important:** The GMA version 1.40 is built on FGLGWS 3.20 and therefore, strongly tied to this Genero BDL version.

**Specifying the GBC for Universal Rendering**

Genero 3.20 introduces **Universal Rendering** based on the GBC web front-end.

The GBC to be bundled with the embedded app is specified with the new `gmabuildtool` option `--build-gbc-runtime`.

**Button.alignment style attribute**

GMA 1.40.03 supports the `alignment` attribute for `BUTTON` form items.

For the possible values of this attribute, see **Button style attributes** on page 1185.

**Cordova plugins from the FOURJS Cordova Github**

Before Genero 3.20, the Cordova plugins wrapper libraries and demos were shipped in the FGLGWS package (FGLDIR).

Cordova plugins, as well as demos and BDL wrapper libraries and demos, are now available from the FOURJS Cordova Github.

For more details, see **Installing Cordova plugins** on page 3645.

**New Cordova front calls listPlugins / getPluginInfo**

Since Genero 3.20 / GMA 1.40, the new `cordova.listPlugins` and `cordova.getPluginInfo` front calls can be used to get information about Cordova plugins bundled with the app.

**64-bit architecture in APK**

Google requires that the APK uploaded to the Play Store must contain a 64-bit version of the app, otherwise it will be rejected after August 1, 2019.

Starting with GMA 1.40.03, the `gmabuildtool` generates an APK with a 32-bit and 64-bit version of the app.

For more details, see **Building Android apps with Genero** on page 3591.

**Android 10 and feInfo (deviceid, iccid, imei)**

Starting with Android 10 (API level 29), to access deviceid, iccid and imei identifiers with the **feInfo** front call, the device must be configured manually with the `READ_PRIVILEGED_PHONE_STATE` privileged permission for fully managed devices, with the Android Enterprise Program. Third-party apps installed from the Google Play Store can't declare privileged permissions.

For more details see **Android permissions** on page 3595.

**Android Command Line Tools replacing SDK Tools**

Starting with GMA 1.40.10, you need to install the Android Command Line Tools, to install and/or update the Android SDK. For more details about this requirement, see the latest Android SDK Tools release note.

If you have an older Android SDK installed (without the `$ANDROID_SDK_TOOLS/cmdline-tools` directory), you can install the command line tools in the existing Android SDK directory, by following the steps described in **Install Genero Mobile for Android** on page 40.

See also the prerequisites to build Android Apps with Genero Mobile.
Genero Mobile for iOS (GMI) 1.40 changes
Modifications to consider when using Genero Mobile for iOS.

Note: This topic describes features changes in the GMI 1.40 product. See also the Mobile section in Genero BDL 3.20 New Features page.

GMI 1.40 with FGLGWS 3.20

Important: The GMI version 1.40 is built on FGLGWS 3.20 and therefore, strongly tied to this Genero BDL version.

Specifying the GBC for Universal Rendering
Genero 3.20 introduces Universal Rendering based on the GBC web front-end.
The GBC to be bundled with the embedded app is specified with the new gmibuildtool option --gbc.

Minimum required iOS and Xcode® versions
Starting with GMI 1.40, the minimal version of the mobile OS must be iOS 11.
With GMI 1.40.02, Xcode® 10 is required to build apps.
With GMI 1.40.05, Xcode® 11 is required to build apps (for iOS 11, 12 and 13).

iOS 13 and HTTPS X509 certificates
Since iOS 13, if you need to perform HTTPS connections with GWS or web components, the server X509 certificates have to follow apple rules defined here: https://support.apple.com/en-us/HT210176.

iOS 13 dark mode and presentation styles
iOS 13 introduced the dark mode. By default, GMI 1.40.05 supports both light and dark modes with the correct system colors. For example, the rendering of RED, GREEN, BLUE and REVERSE FGL color attributes is adapted when the device uses dark mode.
However, if custom presentation style background and foreground colors are used, one may want to force the light mode. This can be achieved by setting the UIUserInterfaceStyle entry in your Info.plist file, in the <dict/> element:

```
<dict>
  ...
  <key>UIUserInterfaceStyle</key>
  <string>Light</string>
</dict>
```

Cordova plugins from the FOURJS Cordova Github
Before Genero 3.20, the Cordova plugins wrapper libraries and demos were shipped in the FGLGWS package (FGLDIR).
Cordova plugins, as well as demos and BDL wrapper libraries and demos, are now available from the FOURJS Cordova Github.
For more details, see Installing Cordova plugins on page 3645.

New Cordova front calls listPlugins / getPluginInfo
Since Genero 3.20 / GMI 1.40, the new cordova.listPlugins and cordova.getPluginInfo front calls can be used to get information about Cordova plugins bundled with the app.
Returned barcode scan codes with `mobile.scanBarCode` front call
Since Genero 3.20 / GMI 1.40, the barcode reader used by GMI is a native built-in. Since then it matches also as close as possible with the GMA barcode types (using underscores). For more details on supported barcodes types, see `mobile.scanBarCode`.

Presentation styles changes
Modifications to consider when using presentation styles.

*common.imageCache* (GDC specific) style attribute is desupported
Starting with GDC 3.20.02, the `imageCache` style attribute is no longer supported.

The `imageCache` style attribute was only available with the GDC front-end, to indicate if the image resource of a form element had to be cached by the front-end. By default, images for image fields (typically changing at runtime) were not cached, while images of static form items (Button, TopMenu item, Toolbar item) were cached.

**Note:** The front-end file cache management has been improved in Genero 3.20. For more details about image resource handling, see *The resource file cache of the front-end* on page 1152.

Action views rendering in GBC chrome bar
Starting with GBC 1.00.51, the value "chrome" can be used for the following Window style attributes:

- `actionPanelPosition`
- `ringMenuPosition`
- `toolBarPosition`

When setting one of these style attributes to the value "chrome", the action views of the corresponding panel are rendered in the GBC chrome bar.

On mobile devices, this will follow Material Design specification.

For more details, see *Action views with GBC* on page 1769.

Collapsible groups with GBC
Since GBC 1.00.52, group elements in grid-based or stack-based form layouts can be defined as collapsible with new presentation style attributes:

- `collapsible`: When set to "yes", makes the group element collapsible by the end user.
- `initiallyCollapsed`: When set to "yes", forces the group to be collapsed when the form is displayed the first time.

Starting with GBC 1.00.54, value of `initiallyCollapsed` can be "never" to never collapse a group when it displays, or "always", to always collapse a group when it displays (ignoring stored settings or previous display state).

- Starting with GBC 1.00.53: `collapserPosition`: Can be set to "left" or "right" to define the position of the collapser icon.

For more details, see *Collapsible groups* on page 1277.

*folder.collapserPosition* (GBC)
Starting with GBC 1.00.53, the `collapserPosition` style attribute can be set to "left" or "right" to define the position of the collapser icon, when the `position` style attribute is defined to "accordion" for the folder.

For more details, see *FOLDER rendering* on page 1275.
**HBox.splitViewRendering (GBC)**

Starting with GBC 1.00.52, HBOX containers can be rendered as split views in GBC, when the width of the webview is lower than a pre-defined number of pixels.

For more details, see Rendering an HBOX as a splitview on page 1278.

**Button.alignment style attribute (GMA)**

The alignment style attribute for BUTTON form items is now supported by:

- GMA 1.40.03

For the possible values of this attribute, see Button.alignment.

**ScrollGrid.itemsAlignment style attribute (GBC)**

Starting with GBC 1.00.55, it is possible to control the alignment of elements inside a SCROLLGRID container with the itemsAlignment style attribute.

For more details, see Controlling element alignment inside a scrollgrid on page 1886.

**New DateTimeEdit.enableCalendar Style attribute (GDC)**

Since GDC version 3.20.09, DATETIMEEDIT fields can be rendered with a date/time picker, when defining the enableCalendar style attribute to "yes".

The date/time picker reacts like with DATEEDIT fields: ENTER key will close it and validate the date/time selection, ESC key will close the widget and cancel the date/time selection.

When using the date/time picker mode of DATETIMEEDIT, the content of the field can be cleared, and it makes the CONSTRUCT mode available.

When used with an INPUT, INPUT ARRAY (or in simple display mode), the format of the date part is defined by the DBDATE environment variable, and the format for the time part is defined by the GDC monitor language settings. In CONSTRUCT mode, the display format is set to ISO, when the variable associated to the field is a DATETIME.

For more details, see DateTimeEdit.enableCalendar.

**New ComboBox qtStyle Style attribute (GDC)**

Since GDC version 3.20.11, COMBOBOX fields can be rendered with a specific Qt widget style, by defining the qtStyle attribute. In order to get a background color defined by the backgroundColor style attribute, the qtStyle attribute must be set to "Windows".

For more details, see ComboBox.qtStyle.

**Front calls changes**

Modifications to consider when using front calls.

**monitor.update parameter for elevation prompt (GDC only)**

Starting with GDC 3.10.19, the monitor.update front call supports a third parameter to specify if the elevation prompt must be displayed, when the update process needs administrator permission.

**Related concepts**

Front calls on page 583
Front call functions execute on the platform where the front-end is installed.

Web components on page 1922
This section describes how to use web components in your application.

**Web components changes**
Modifications to consider when using web components.

There are no changes regarding web components in Genero BDL version 3.20.

**Related concepts**
- [Web components](#) on page 1922

This section describes how to use web components in your application.

**Database drivers changes**
New and desupported database drivers.

**New database server versions supported in Genero 3.20:**
Support for new database server releases, with existing ODI drivers:
- PostgreSQL 11 and 12 is supported with the existing dbmpgs_9 driver (linked to libpq.so.5).
- Microsoft SQL Server 2019 (v15) is supported with the existing dbmsnc_17 driver (linked to libmsodbcsql-17.so).

Support for new database server and client releases with new ODI drivers:
- Oracle® 18c database driver (dbmora_18/dbmora) for Oracle 18c database server.
  **Important:** The generic driver name dbmora maps now to dbmora_18. If your application is using Oracle 12c or Oracle 11g, specify respectively dbmora_12 or dbmora_11 drivers in the connection parameters.
- Oracle® 19c database server support, by using the (dbmora_18/dbmora) driver. On Unix platforms, the dbmora_18 driver is linked with libclntsh.so.18.1, which is provided by Oracle Instant Client 19c.

ODI driver replacements:
- Oracle® MySQL 5.6 driver (dbmmys_5_6) has to be used in place of the desupported dbmmys_5_5 driver.

For more details, see [Database driver specification (driver)](#) on page 662.

**Databases / ODI drivers desupported in Genero 3.20:**
Database drivers for old database client versions are removed in accordance with DB vendor de-support plans:
- Microsoft™ SQL Server 2008 / SQL Native Client V10 (dbmsnc_10)
- Oracle® MySQL 5.5.x (dbmmys_5_5) - for MySQL 5.6, use the new driver dbmmys_5_6.
  **Note:** The dbmmys_5_5 driver could also be used to connect to Mariadb 10.0 and 10.1. These Mariadb versions are no longer supported: Use Mariadb 10.2+ with dbmdmdb_10_2.

**Minimal required FreeTDS version is 1.00.104**
See [FreeTDS 1.00 for SQL Server](#) on page 123 for more details.

**FGLSQLDEBUG levels and output**
Starting with version 3.20, the FGLSQLDEBUG and fgl_sqldebug() maximum debug level is 3. If you set the SQL debug level to a higher value, it will be equivalent to level 3.

When FGLSQLDEBUG is enabled, the OPTIONS SQL INTERRUPT ON/OFF is traced.
**DATETIME SQL type mappings**
For some databases, the type mapping for `DATETIME HOUR TO MINUTE` has changed.

**Summary of DATETIME storage changes**
Starting with Genero 3.20, the native types used to store some of the `DATETIME` `qual1` TO `qual2` data types have changed.

**Important:** The native type mapping has not changed for the most common date/time types `DATETIME YEAR TO SECOND`, `DATETIME YEAR TO FRACTION(N)`, `DATETIME HOUR TO SECOND`, `DATETIME HOUR TO FRACTION(N)`. The `DATETIME` type concerned by this modification is `DATETIME HOUR TO MINUTE`, and only for the database brands listed in this topic.

`DATETIME HOUR TO MINUTE` data is now stored with the closest native time data type of the target database. Prior to version 3.20, such data was stored in a timestamp (`YYYY-MM-DD hh:mm:ss[.ffff]`), with the year/month/day parts set to `1900-01-01`. This was not consistent with `DATETIME HOUR TO SECOND` storage, using time (`hh:mm:ss[.ffff]`) types.

Using the native time data type for time-only data simplifies interoperability with other applications and database components (for example, with data load/unload tools).

**Note:** Only the database brands listed in this page are concerned by this `DATETIME` storage change.

**Database schema update**
Existing databases on development and production sites will need data type modification for columns concerned by this change.

Depending on the type of database, an `ALTER TABLE` statement can be used. For example, with SQL Server, it is possible to convert a `datetime2(0)` column to `time(0)`:  

```
CREATE TABLE t1 ( pk INT, dt DATETIME2(0) )
INSERT INTO t1 VALUES ( 101, '1900-01-01 12:33:44' )
SELECT * FROM t1
ALTER TABLE t1 ALTER COLUMN dt TIME(0)
SELECT * FROM t1
DROP TABLE t1
```

For databases that do not support such date/time type modification with `ALTER TABLE`, create a new table with the new types, write some SQL or BDL code to fill the new table with the old table rows by doing a `CAST()` , drop the old table and rename the new table to the original table name.

**Microsoft SQL Server**

**Table 95: Microsoft SQL Server date/time mapping changes**

<table>
<thead>
<tr>
<th>FGL Data Type</th>
<th>FGL 3.10</th>
<th>FGL 3.20</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DATETIME HOUR TO HOUR</code></td>
<td><code>DATETIME2(0)</code></td>
<td><code>TIME(0)</code></td>
</tr>
<tr>
<td><code>DATETIME HOUR TO MINUTE</code></td>
<td><code>DATETIME2(0)</code></td>
<td><code>TIME(0)</code></td>
</tr>
<tr>
<td><code>DATETIME HOUR TO SECOND</code></td>
<td><code>TIME(0)</code></td>
<td>No change</td>
</tr>
<tr>
<td><code>DATETIME HOUR TO FRACTION(N)</code></td>
<td><code>TIME(N)</code></td>
<td>No change</td>
</tr>
<tr>
<td><code>DATETIME MINUTE TO MINUTE</code></td>
<td><code>DATETIME2(0)</code></td>
<td><code>TIME(0)</code></td>
</tr>
<tr>
<td><code>DATETIME MINUTE TO SECOND</code></td>
<td><code>DATETIME2(0)</code></td>
<td><code>TIME(0)</code></td>
</tr>
<tr>
<td><code>DATETIME MINUTE TO FRACTION(N)</code></td>
<td><code>DATETIME2(n)</code></td>
<td><code>TIME(n)</code></td>
</tr>
<tr>
<td><code>DATETIME SECOND TO SECOND</code></td>
<td><code>DATETIME2(0)</code></td>
<td><code>TIME(0)</code></td>
</tr>
<tr>
<td>FGL Data Type</td>
<td>FGL 3.10</td>
<td>FGL 3.20</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>--------------</td>
<td>--------------</td>
</tr>
<tr>
<td>DATETIME SECOND TO FRACTION(N)</td>
<td>DATETIME2(n)</td>
<td>TIME(n)</td>
</tr>
<tr>
<td>DATETIME FRACTION TO FRACTION(N)</td>
<td>DATETIME2(n)</td>
<td>TIME(n)</td>
</tr>
</tbody>
</table>

See also Date/time support with Microsoft SQL Server.

**IBM DB2 LUW**

**Table 96: IBM DB2 LUW date/time mapping changes**

<table>
<thead>
<tr>
<th>FGL Data Type</th>
<th>FGL 3.10</th>
<th>FGL 3.20</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATETIME HOUR TO HOUR</td>
<td>TIMESTAMP</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME HOUR TO MINUTE</td>
<td>TIMESTAMP</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME HOUR TO SECOND</td>
<td>TIME</td>
<td>No change</td>
</tr>
<tr>
<td>DATETIME HOUR TO FRACTION(n)</td>
<td>TIMESTAMP (1)</td>
<td>No change</td>
</tr>
<tr>
<td>DATETIME MINUTE TO MINUTE</td>
<td>TIMESTAMP</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME MINUTE TO SECOND</td>
<td>TIMESTAMP</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME MINUTE TO FRACTION(n)</td>
<td>TIMESTAMP (1)</td>
<td>No change</td>
</tr>
<tr>
<td>DATETIME SECOND TO SECOND</td>
<td>TIMESTAMP</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME SECOND TO FRACTION(n)</td>
<td>TIMESTAMP (1)</td>
<td>No change</td>
</tr>
<tr>
<td>DATETIME FRACTION TO FRACTION(n)</td>
<td>TIMESTAMP (1)</td>
<td>No change</td>
</tr>
</tbody>
</table>

Notes:

1. Must use DB2 TIMESTAMP for fraction of seconds storage, DB2 TIME has no fractional part.

See also Date/time support with IBM DB2 LUW.

**IBM Netezza**

**Table 97: IBM Netezza date/time mapping changes**

<table>
<thead>
<tr>
<th>FGL Data Type</th>
<th>FGL 3.10</th>
<th>FGL 3.20</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATETIME HOUR TO HOUR</td>
<td>TIMESTAMP</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME HOUR TO MINUTE</td>
<td>TIMESTAMP</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME HOUR TO SECOND</td>
<td>TIME</td>
<td>No change</td>
</tr>
<tr>
<td>DATETIME HOUR TO FRACTION(n)</td>
<td>TIMESTAMP</td>
<td>TIME(1)</td>
</tr>
<tr>
<td>DATETIME MINUTE TO MINUTE</td>
<td>TIMESTAMP</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME MINUTE TO SECOND</td>
<td>TIMESTAMP</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME MINUTE TO FRACTION(n)</td>
<td>TIMESTAMP</td>
<td>TIME(1)</td>
</tr>
<tr>
<td>DATETIME SECOND TO SECOND</td>
<td>TIMESTAMP</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME SECOND TO FRACTION(n)</td>
<td>TIMESTAMP</td>
<td>TIME(1)</td>
</tr>
<tr>
<td>DATETIME FRACTION TO FRACTION(n)</td>
<td>TIMESTAMP</td>
<td>TIME(1)</td>
</tr>
</tbody>
</table>
Notes:
1. Netezza can store fraction of seconds in `TIME` columns, but when fetching data, the NZ ODBC driver produces an `SQL_TYPE_TIME`, returning `hh:mm:ss` without the fraction part. This is a lack of NZ ODBC. See issue FGL-4807.

See also Date/time support with IBM Netezza.

**SAP ASE**

Table 98: SAP ASE date/time mapping changes

<table>
<thead>
<tr>
<th>FGL Data Type</th>
<th>FGL 3.10</th>
<th>FGL 3.20</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATETIME HOUR TO HOUR</td>
<td>BIGDATETIME</td>
<td>BIGTIME</td>
</tr>
<tr>
<td>DATETIME HOUR TO MINUTE</td>
<td>BIGDATETIME</td>
<td>BIGTIME</td>
</tr>
<tr>
<td>DATETIME HOUR TO SECOND</td>
<td>BIGTIME</td>
<td>No change</td>
</tr>
<tr>
<td>DATETIME HOUR TO FRACTION(n)</td>
<td>BIGTIME</td>
<td>No change</td>
</tr>
<tr>
<td>DATETIME MINUTE TO MINUTE</td>
<td>BIGDATETIME</td>
<td>BIGTIME</td>
</tr>
<tr>
<td>DATETIME MINUTE TO SECOND</td>
<td>BIGDATETIME</td>
<td>BIGTIME</td>
</tr>
<tr>
<td>DATETIME MINUTE TO FRACTION(n)</td>
<td>BIGDATETIME</td>
<td>BIGTIME</td>
</tr>
<tr>
<td>DATETIME SECOND TO SECOND</td>
<td>BIGDATETIME</td>
<td>BIGTIME</td>
</tr>
<tr>
<td>DATETIME SECOND TO FRACTION(n)</td>
<td>BIGDATETIME</td>
<td>BIGTIME</td>
</tr>
<tr>
<td>DATETIME FRACTION TO FRACTION(n)</td>
<td>BIGDATETIME</td>
<td>BIGTIME</td>
</tr>
</tbody>
</table>

See also Date/time support with SAP ASE.

**Oracle MySQL and MariaDB**

Table 99: Oracle MySQL and MariaDB date/time mapping changes

<table>
<thead>
<tr>
<th>FGL Data Type</th>
<th>FGL 3.10</th>
<th>FGL 3.20</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATETIME HOUR TO HOUR</td>
<td>TIME</td>
<td>No change</td>
</tr>
<tr>
<td>DATETIME HOUR TO MINUTE</td>
<td>TIME</td>
<td>No change</td>
</tr>
<tr>
<td>DATETIME HOUR TO SECOND</td>
<td>TIME</td>
<td>No change</td>
</tr>
<tr>
<td>DATETIME HOUR TO FRACTION(n)</td>
<td>TIME(n)</td>
<td>No change</td>
</tr>
<tr>
<td>DATETIME MINUTE TO MINUTE</td>
<td>DATETIME</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME MINUTE TO SECOND</td>
<td>DATETIME</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME MINUTE TO FRACTION(n)</td>
<td>DATETIME(n)</td>
<td>TIME(n)</td>
</tr>
<tr>
<td>DATETIME SECOND TO SECOND</td>
<td>DATETIME</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME SECOND TO FRACTION(n)</td>
<td>DATETIME(n)</td>
<td>TIME(n)</td>
</tr>
<tr>
<td>DATETIME FRACTION TO FRACTION(n)</td>
<td>DATETIME(n)</td>
<td>TIME(n)</td>
</tr>
</tbody>
</table>

See also Date/time support with Oracle MySQL and MariaDB.
### PostgreSQL

**Table 100: PostgreSQL date/time mapping changes**

<table>
<thead>
<tr>
<th>FGL Data Type</th>
<th>FGL 3.10</th>
<th>FGL 3.20</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATETIME HOUR TO HOUR</td>
<td>TIMESTAMP(0)</td>
<td>TIME(0)</td>
</tr>
<tr>
<td>DATETIME HOUR TO MINUTE</td>
<td>TIME(0)</td>
<td>No change</td>
</tr>
<tr>
<td>DATETIME HOUR TO SECOND</td>
<td>TIME(0)</td>
<td>No change</td>
</tr>
<tr>
<td>DATETIME HOUR TO FRACTION(N)</td>
<td>TIME(N)</td>
<td>No change</td>
</tr>
<tr>
<td>DATETIME MINUTE TO MINUTE</td>
<td>TIMESTAMP(0)</td>
<td>TIME(0)</td>
</tr>
<tr>
<td>DATETIME MINUTE TO SECOND</td>
<td>TIMESTAMP(0)</td>
<td>TIME(0)</td>
</tr>
<tr>
<td>DATETIME MINUTE TO FRACTION(N)</td>
<td>TIMESTAMP(n)</td>
<td>TIME(n)</td>
</tr>
<tr>
<td>DATETIME SECOND TO SECOND</td>
<td>TIMESTAMP(0)</td>
<td>TIME(0)</td>
</tr>
<tr>
<td>DATETIME SECOND TO FRACTION(N)</td>
<td>TIMESTAMP(n)</td>
<td>TIME(n)</td>
</tr>
</tbody>
</table>

**Note:** All PostgreSQL time data type mappings use `TIME` and `TIMESTAMP` without time zone. See also [Date/time support with PostgreSQL](#).

### SAP HANA

**Table 101: SAP HANA date/time mapping changes**

<table>
<thead>
<tr>
<th>FGL Data Type</th>
<th>FGL 3.10</th>
<th>FGL 3.20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time class types</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DATETIME HOUR TO HOUR</td>
<td>TIMESTAMP</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME HOUR TO MINUTE</td>
<td>TIME</td>
<td>No change</td>
</tr>
<tr>
<td>DATETIME HOUR TO SECOND</td>
<td>TIME</td>
<td>No change</td>
</tr>
<tr>
<td>DATETIME HOUR TO FRACTION(n)</td>
<td>TIMESTAMP(1)</td>
<td>No change</td>
</tr>
<tr>
<td>DATETIME MINUTE TO MINUTE</td>
<td>TIMESTAMP</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME MINUTE TO SECOND</td>
<td>TIMESTAMP</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME MINUTE TO FRACTION(n)</td>
<td>TIMESTAMP(1)</td>
<td>No change</td>
</tr>
<tr>
<td>DATETIME SECOND TO SECOND</td>
<td>TIMESTAMP</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME SECOND TO FRACTION(n)</td>
<td>TIMESTAMP(1)</td>
<td>No change</td>
</tr>
<tr>
<td>DATETIME FRACTION TO FRACTION(n)</td>
<td>TIMESTAMP(1)</td>
<td>No change</td>
</tr>
</tbody>
</table>

Date/time class types (2)

<p>| DATETIME YEAR TO YEAR         | TIMESTAMP   | SECONDDATE  |
| DATETIME YEAR TO MONTH        | TIMESTAMP   | SECONDDATE  |
| DATETIME YEAR TO DAY          | TIMESTAMP   | SECONDDATE  |
| DATETIME YEAR TO HOUR         | TIMESTAMP   | SECONDDATE  |</p>
<table>
<thead>
<tr>
<th>FGL Data Type</th>
<th>FGL 3.10</th>
<th>FGL 3.20</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATETIME YEAR TO MINUTE</td>
<td>SECONDDATE</td>
<td>No change</td>
</tr>
<tr>
<td>DATETIME YEAR TO SECOND</td>
<td>SECONDDATE</td>
<td>No change</td>
</tr>
<tr>
<td>DATETIME YEAR TO FRACTION(n)</td>
<td>TIMESTAMP</td>
<td>No change</td>
</tr>
<tr>
<td>DATETIME MONTH TO MONTH</td>
<td>TIMESTAMP</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME MONTH TO DAY</td>
<td>TIMESTAMP</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME MONTH TO HOUR</td>
<td>TIMESTAMP</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME MONTH TO MINUTE</td>
<td>TIMESTAMP</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME MONTH TO SECOND</td>
<td>TIMESTAMP</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME MONTH TO FRACTION(n)</td>
<td>TIMESTAMP</td>
<td>No change</td>
</tr>
<tr>
<td>DATETIME DAY TO DAY</td>
<td>TIMESTAMP</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME DAY TO HOUR</td>
<td>TIMESTAMP</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME DAY TO MINUTE</td>
<td>TIMESTAMP</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME DAY TO SECOND</td>
<td>TIMESTAMP</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME DAY TO FRACTION(n)</td>
<td>TIMESTAMP</td>
<td>No change</td>
</tr>
</tbody>
</table>

Notes:
1. Must use SAP HANA TIMESTAMP for fraction of seconds storage, SAP HANA TIME has no fractional part.
2. To be consistent with time class storage, all FGL DATETIME types that can be stored in SECONDDATE will be stored with that type. Otherwise, we use TIMESTAMP for the fraction of seconds.

See also Date/time support with SAP HANA.

**SQLite**

With SQLite, the mapping rules for FGL DATETIME use custom type names such as SMALLTIME and TINYDATETIME, to keep type information, for more details see DATE and DATETIME data types on page 1013.

**Table 102: SQLite date/time mapping changes**

<table>
<thead>
<tr>
<th>FGL Data Type</th>
<th>FGL 3.10</th>
<th>FGL 3.20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time class types</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DATETIME HOUR TO HOUR</td>
<td>TIMESTAMP</td>
<td>SMALLTIME</td>
</tr>
<tr>
<td>DATETIME HOUR TO MINUTE</td>
<td>SMALLTIME</td>
<td>No change</td>
</tr>
<tr>
<td>DATETIME HOUR TO SECOND</td>
<td>TIME</td>
<td>No change</td>
</tr>
<tr>
<td>DATETIME HOUR TO FRACTION(n)</td>
<td>TIME(n)</td>
<td>No change</td>
</tr>
<tr>
<td>DATETIME MINUTE TO MINUTE</td>
<td>TIMESTAMP</td>
<td>SMALLTIME</td>
</tr>
<tr>
<td>DATETIME MINUTE TO SECOND</td>
<td>TIMESTAMP</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME MINUTE TO FRACTION(n)</td>
<td>TIMESTAMP(n)</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME SECOND TO SECOND</td>
<td>TIMESTAMP</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME SECOND TO FRACTION(n)</td>
<td>TIMESTAMP(n)</td>
<td>TIME</td>
</tr>
</tbody>
</table>
### FGL Data Type

<table>
<thead>
<tr>
<th>FGL Data Type</th>
<th>FGL 3.10</th>
<th>FGL 3.20</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATETIME FRACTION TO FRACTION(n)</td>
<td>TIMESTAMP(n)</td>
<td>TIME(n)</td>
</tr>
</tbody>
</table>

**Date/time class types**

<table>
<thead>
<tr>
<th>DATETIME YEAR TO YEAR</th>
<th>TIMESTAMP</th>
<th>TINYDATETIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATETIME YEAR TO MONTH</td>
<td>TIMESTAMP</td>
<td>TINYDATETIME</td>
</tr>
<tr>
<td>DATETIME YEAR TO DAY</td>
<td>TINYDATETIME</td>
<td>No change</td>
</tr>
<tr>
<td>DATETIME YEAR TO HOUR</td>
<td>TIMESTAMP</td>
<td>SMALLDATETIME</td>
</tr>
<tr>
<td>DATETIME YEAR TO MINUTE</td>
<td>SMALLDATETIME</td>
<td>No change</td>
</tr>
<tr>
<td>DATETIME YEAR TO SECOND</td>
<td>DATETIME</td>
<td>No change</td>
</tr>
<tr>
<td>DATETIME YEAR TO FRACTION(n)</td>
<td>DATETIME(n)</td>
<td>No change</td>
</tr>
<tr>
<td>DATETIME MONTH TO MONTH</td>
<td>TIMESTAMP</td>
<td>TINYDATETIME</td>
</tr>
<tr>
<td>DATETIME MONTH TO DAY</td>
<td>TIMESTAMP</td>
<td>TINYDATETIME</td>
</tr>
<tr>
<td>DATETIME MONTH TO HOUR</td>
<td>TIMESTAMP</td>
<td>SMALLDATETIME</td>
</tr>
<tr>
<td>DATETIME MONTH TO MINUTE</td>
<td>TIMESTAMP</td>
<td>SMALLDATETIME</td>
</tr>
<tr>
<td>DATETIME MONTH TO SECOND</td>
<td>TIMESTAMP</td>
<td>DATETIME</td>
</tr>
<tr>
<td>DATETIME MONTH TO FRACTION(n)</td>
<td>TIMESTAMP</td>
<td>DATETIME(n)</td>
</tr>
<tr>
<td>DATETIME DAY TO DAY</td>
<td>TIMESTAMP</td>
<td>TINYDATETIME</td>
</tr>
<tr>
<td>DATETIME DAY TO HOUR</td>
<td>TIMESTAMP</td>
<td>SMALLDATETIME</td>
</tr>
<tr>
<td>DATETIME DAY TO SECOND</td>
<td>TIMESTAMP</td>
<td>SMALLDATETIME</td>
</tr>
<tr>
<td>DATETIME DAY TO FRACTION(n)</td>
<td>TIMESTAMP</td>
<td>DATETIME</td>
</tr>
</tbody>
</table>

**Notes:**

1. To be consistent with TIME class storage, types that can be stored in TINYDATETIME, SMALLDATETIME, DATETIME and DATETIME(n) will be stored with such type.

See also [Date/time support with SQLite](#).

**FreeTDS 1.00 for SQL Server**

Genero 3.20 requires FreeTDS version 1.00+ to connect to SQL Server.

Starting with Genero 3.20, the dbmf.t.m driver to connect to SQL Server requires FreeTDS version 1.00.

**Important:** The minimal required version of FreeTDS is **1.00.104**.


**Note:** FreeTDS 1.00 fixes several bugs related to date/time data, allowing better support for date/time types.

For more details about FreeTDS / SQL Server configuration, see [SQL Server client environment](#).
SQL Server drivers performance
SQL Server ODI drivers based on FreeTDS, Easysoft and MS ODBC have been reviewed to achieve better execution times.

Starting with Genero BDL 3.20, the FTM, ESM and SNC drivers for Microsoft® SQL Server use now the ODBC API SQLExecDirect() for any sort of SQL statement. In prior versions, the drivers use SQLPrepare() + SQLExecute() for common SQL statements such as SELECT, INSERT, UPDATE, DELETE.

Other improvements have been done, such as setting the ODBC cursor name (SQLSetCursorName()) only when needed, namely for SELECT statements with FOR UPDATE clause.

This leads to a reduction in the number of sp_cursor* stored procedure calls and shows a noticeable improvement in performance, especially in a client/server configuration where each sp_cursor* call results in a network round trip.

Related concepts
Performances with SQL interruption on page 124
With some database drivers, performances can be impacted when using OPTIONS SQL INTERRUPT ON.

Performances with SQL interruption
With some database drivers, performances can be impacted when using OPTIONS SQL INTERRUPT ON.

Starting with Genero BDL 3.20, dbmsnc / MS ODBC SQL database drivers have been reviewed to improve support for SQL interruption, by using asynchronous ODBC calls.

This solution requires additional overhead, that can slow down regular SQL statements which do not need SQL interruption handling.

Note: Most of SQL statements of a program do not need SQL interruption handling.

Instead of enabling SQL interruption with OPTIONS SQL INTERRUPT ON at the beginning of the program, consider enabling this option only for the execution of SQL queries that can be interrupted. This programming pattern is anyway a good practice, since SQL interruption requires exception handling that needs to be treated.

Note: To easily identify when OPTIONS SQL INTERRUPT ON/OFF is used in your programs, the FGLSQLDEBUG output has been enhanced to trace this instruction:

```
SQL: SQL INTERRUPT ON
  4gl source      : sqlint.4gl line=5
  curr driver     : ident='dbmdefault'
  curr connection : ident='_1' (dbspec=[test1])
  Execution time  : 0 00:00:00.00003
...
SQL: SQL INTERRUPT OFF
  4gl source      : sqlint.4gl line=12
  curr driver     : ident='dbmdefault'
  curr connection : ident='_1' (dbspec=[test1])
  Execution time  : 0 00:00:00.00003
```

Related concepts
SQL Server drivers performance on page 124
SQL Server ODI drivers based on FreeTDS, Easysoft and MS ODBC have been reviewed to achieve better execution times.

PostgreSQL 12 notes
This topics contains notes about PostgreSQL 12 changes that affect Genero applications.

Desupport of OID columns
Genero BDL 3.20 supports PostgreSQL 12.

Starting with PostgreSQL version 12, the OID columns (similar to Informix ROWID columns) are no longer supported.
If your application is using ROWIDs, you must review the code. For more details, see ROWID columns on page 984.

**Related concepts**
PostgreSQL on page 964

**Sharing GLOBALS with C Extensions**
Sharing of global variables with a C Extension is no longer supported.

Before Genero BDL version 3.20, it was possible to produce a .h and .c files with the -G option of fglcomp to share global variables with C Extensions:

```
fglcomp -G myglobals.4gl
```

This command produced the myglobals.h and myglobals.c files, which could be used to build the C Extension library and share global variables defined in the myglobals.4gl file.

New features of the BDL language have required to review internals and deny global variable sharing with C Extensions. In addition, this practice was not recommended as stated in previous versions of the documentation.

To migrate existing code, implement a C Extension module that defines static variables in the C module, and set/get functions to assign or read the value of these variables.

**Related concepts**
C-Extensions on page 2230

With C-Extensions, you can bind your own C libraries in the runtime system, to call C function from the application code.

**Dynamic array assignment with .* notation**
The .* notation to assign dynamic arrays is discouraged.

Before Genero BDL version 3.20, the syntax to assign a dynamic array reference to another dynamic array variable was done with the .* notation:

```
LET dynarr2.* = dynarr1.*
```

Starting with version 3.20, the proper syntax to assign a dynamic array reference to another (or to copy a static array to another static array), is to use the array variable names without the .* suffix:

```
LET dynarr2 = dynarr1
```

**Important**: The .* notation to assign arrays is still supported for backward compatibility, to be able to compile legacy code. There is no need to change existing code. Use the new syntax without .* in new development.

**Related concepts**
Copying and assigning arrays on page 424
Arrays can be fully copied or assigned by reference.

**Record copy with .* notation**
The .* notation to assign records is discouraged.

Before Genero BDL version 3.20, the syntax to copy a record to another record was done with the .* notation:

```
LET record2.* = record1.*
```

Starting with version 3.20, the proper syntax to assign a record to another record defined with the same RECORD structure, or TYPE, is to use the record variable names without the .* suffix:

```
LET record2 = record1
```
**Important:** The `.*` notation to assign records is still supported for backward compatibility, to be able to compile legacy code. There is no need to change existing code. Use the new syntax without `.*` in new development.

The new syntax without the `.*` syntax has mainly been introduced for the following reason:

When the compiler sees `record.*` as function parameter, this is equivalent to list all members of the record: `record.first-element, ..., record.last-member`.

For example, the next statements are equivalent:

```plaintext
CALL func1(record.*)
CALL func2(record.field1, ..., record.fieldN) -- not nice, but legal
```

Genero 3.20 has introduced *record types with methods*, and records can be passed as read-only parameters without the `.*` notation. In this case, the compiler makes a stronger type checking for method parameters:

```plaintext
CALL var.method1(record) -- OK if method1 is defined with that record type
CALL var.method1(record.*) -- Denied!
CALL var.method1(record.field1, ..., record.fieldN) -- Denied!
```

**Related concepts**

- **Copying records** on page 412
  Records can be assigned to each other with the `=` operator.

- **Circular dependency with IMPORT FGL**
  The compiler allows that two modules reference each other with `IMPORT FGL`.

Circular module references occur when several modules reference each other with `IMPORT FGL`.

Before Genero BDL version 3.20, a circular reference was producing the compilation error -8402.

Starting with Genero BDL version 3.20, circular references with `IMPORT FGL` are now allowed.

**Related information**

- **Circular module references** on page 482

- **fglrun -m / -M options**
  The memory leak checking options of fglrun are not longer supported.

Before Genero BDL version 3.20, the `fglrun` command was supporting `-m` and `-M` options to check for memory leaks in internal runtime system structures.

Starting with version 3.20, the `fglrun` `-m` and `-M` options are no longer supported.

**Related concepts**

- **fglrun** on page 2065
  The `fglrun` tool is the runtime system program that executes p-code programs.

- **Case insensitive names with UI methods**
  Methods of built-in classes using user interface object names are now case insensitive.

Before Genero BDL version 3.20, built-in class methods such as `ui.Dialog.setActionActive()` were case sensitive. Since compilers convert names to lowercase, it was required to use lowercase names in the built-in class methods:

```plaintext
ON ACTION MyAction  -- converted to myaction
...
CALL DIALOG.setActionActive("MyAction",FALSE)  -- produced a runtime error!
CALL DIALOG.setActionActive("myaction",FALSE)  -- works
```

Starting with version 3.20, these methods are now case insensitive, so you can write:

```plaintext
ON ACTION MyAction  -- converted to myaction
```
CALL DIALOG.setActionActive("MyAction",FALSE)

**Note:** This change is backward compatible, there is no need to modify the existing code.

User interface object names defined in the form files can also be referenced with the exact name.

For example, in the .per form file:

```plaintext
EDIT f01 = Customer.CustAddr, ... ;
GROUP g1 : Group1, ... ;
... 
SCREEN RECORD CustRec (...);
...
```

In the .4gl source file:

```plaintext
CALL DIALOG.setFieldActive("Customer.CustAddr",FALSE)
...
CALL DIALOG.getForm().setElementHidden("Group1",1)
...
LET r = DIALOG.getCurrentRow("CustRec")
...
```

**Note:** The methods to build dynamic dialogs are still case sensitive. For example, to define an ON ACTION trigger:

```plaintext
CALL mydlg.addTrigger("ON ACTION accept").
```

**Related concepts**

- Identifying actions in ui.Dialog methods on page 2555
- Identifying fields in ui.Dialog methods on page 2556
- Identifying screen-arrays in ui.Dialog methods on page 2557
- Identifying elements in ui.Form methods on page 2510
- ui.Window.forName on page 2491
  Get a window object by name.
- ui.ComboBox.forName on page 2564
  Search for a combobox in the current form.

**FLOAT/SMALLFLOAT to string conversion**

New FGLPROFILE entry `fglr.run.floatToCharScale2` for FLOAT/SMALLFLOAT types.

Since Genero 2.50, the conversion to string from a `DECIMAL(P)`, FLOAT and SMALLFLOAT has been revised, to keep all significant digits and avoid data loss. See Floating point to string conversion on page 170 upgrade note for more details.

Until Genero 3.20, the number to string formatting could only be controlled for `DECIMAL(P)` types with the `fglr.run.decToCharScale2` FGLPROFILE entry.

Starting with version 3.20.06 (also backported in 3.10.19), a new FGLPROFILE entry can be used, to get the pre-2.50 behavior and round FLOAT/SMALLFLOAT values to 2 digits (this applies to all contexts):

```plaintext
fglr.run.floatToCharScale2 = true
```

Starting with version 3.20.09 (also backported in 3.10.20), another FGLPROFILE entry has been added, to get the 2 decimal digit formatting of FLOAT/SMALLFLOAT only in the context of the `PRINT` statement in reports:

```plaintext
fglr.run.floatToCharScale2.print = true
```

**Note:** Do not use the `fglr.run.floatToCharScale2*` configuration parameters, unless you have migration issues. These configuration parameters apply only to FLOAT and SMALLFLOAT: DECIMAL(P) conversions to string is not impacted (must use `fglr.run.decToCharScale2*` entries instead).
IBM DB2 BOOLEAN SQL type
The native BOOLEAN type is now used instead of CHAR(1) for DB2 11.x.
IBM DB2 LUW version 11.1.1.1 introduced the BOOLEAN SQL type.
Before Genero 3.20, the BOOLEAN type was converted to CHAR(1), storing the value '1' for TRUE and '0' for FALSE.
Starting with Genero 3.20, when connected to DB2 11.1 or +, the BOOLEAN keyword is left as is in DDL statements such as CREATE TABLE and ALTER TABLE. The fgldbsch database schema extraction tool will detect BOOLEAN columns and write the corresponding type into .sch files.

IBM DB2 deferred prepare
The DB2 FGLPROFILE option to control deferred prepare is desupported.
Before Genero 3.20, the DB2 LUW ODI driver was setting the CLI statement attribute SQL_ATTR_DEFERRED_PREPARE to SQL_DEFERRED_PREPARE_OFF or SQL_DEFERRED_PREPARE_ON, depending on the db2.prepare.deferred FGLPROFILE parameter. By default, deferred prepare was disabled by the ODI driver to make BDL PREPARE and DECLARE instructions fail when the SQL statement contains errors, by forcing the DB2 CLI client library to send the SQL to the server at prepare time. However, the common practice these days is to improve performances by enabling deferred prepare, sending the SQL only at execution time (EXECUTE/OPEN/FOREACH):

```sql
dbi.database.mydbname.db2.prepare.deferred = true
```
Starting with Genero 3.20, the db2.prepare.deferred FGLPROFILE option is ignored: The driver will use the DB2 CLI settings. By default, with DB2 CLI, deferred prepare is ON. If needed, you can control this feature with the DeferredPrepare DB2 CLI configuration setting of the db2cli.ini file:

```ini
DeferredPrepare = 0
```

ORACLE rowid in SQLCA.SQLERRM
With ORACLE, the rowid of the last affected row is available in SQLCA.SQLERRM.
Before Genero 3.20, the character extended (base 64) representation of the rowid of the row affected by an INSERT, UPDATE or DELETE could be found in the SQLCA.SQLERRM register, when the following FGLPROFILE entry was defined:

```ini
dbi.database.mydbname.ora.rowid.retrieve = true
```
This FGLPROFILE entry was required to fill the SQLCA.SQLERRM register with the result of a SELECT ROWIDTOCHAR(?) FROM DUAL statement, that was executed after each INSERT, UPDATE or DELETE SQL statement. For performance reasons, the database.dbname.ora.rowid.retrieve switch was by default set to false.

Note: For maximum SQL portability, replace rowids usage by primary keys. See Using ROWID columns on page 644.
Starting with Genero 3.20, to fill the SQLCA.SQLERRM register, the Oracle ODI driver uses now the OCIRowidToChar() API to convert the binary rowid value to the character extended (base 64) representation. Since the additional SELECT statement is not required anymore, the FGLPROFILE entry database.dbname.ora.rowid.retrieve does no longer exist.

The SQLCA.SQLERRM is now implicitly filled with the rowid of the row affected by an INSERT, UPDATE, DELETE statement, and after a FETCH of a cursor declared with SELECT FOR UPDATE.

**Related concepts**
- ROWID columns on page 935

**BDL 3.10 upgrade guide**

These topics describe product changes you must be aware of when upgrading to version 3.10.

**Important:** This is an incremental upgrade guide that covers only topics related to the Genero BDL version specified in the page title. Check prior upgrade guides if you migrate from an earlier version. Make sure to also read about the new features for this Genero version.

Corresponding new features page: [BDL 3.10 new features](#) on page 54.

**Web Services changes**

There are changes in support of web services in Genero 3.10.

**The XMLElementNillable, XMLOptional, and XMLNillable attributes**

The management of XML serialization when serializing nullable has changed. These attributes define how a NULL value is interpreted in XML.

When using the XMLOptional on page 3539 attribute, the behavior of the XML serialization has changed; it no longer handles XML nullable.

The new XMLNillable on page 3540 attribute can be used to specify that the XML representation of a NULL value must be xsi:nil="true".

In order to get the same result as when using XMLOptional in prior versions, set both XMLOptional and XMLNillable.

Instead of specifying each element individually with XMLNillable, the XMLElementNillable on page 3542 attribute can be used with a BDL RECORD defined as TYPE or DEFINE to specify the XML representation for nullable in all elements in the record. This eases migration to the new XMLNillable behavior, because now you can get the same behavior as before defining database records by adding the XMLElementNillable attribute.

**Important:** If in versions prior to GWS 3.10 you had the following expected XML serialization behavior, you need to take action to avoid serializer errors:

- A variable that was expected to serialize an xsi:nil value, you must set the XMLNillable attribute on that variable.
- A record defined with LIKE for fields in a database table which may allow null values, you must set the XMLElementNillable attribute as in the example.

```
DEFINE var RECORD ATTRIBUTE(XMLElementNillable) LIKE customer.*
```

It is therefore recommended to do the following:

- Recompile the Genero Web Service server to create a new WSDL that supports the XML nillable feature.
- Regenerate all Genero Web Service client stubs from the newly-generated WSDL to get the support of XML nillable. Regenerate client stubs using the fglwsdl tool.

**Better support for Web Services API on iOS/GMI**

Most of the Web Services APIs are not available on iOS mobile devices.
There are still some exceptions, related to iOS restrictions. For details see Web Services on mobile devices on page 3586.

**Note:** In versions before 3.10, the iOS app displayed a pop-up dialog to cancel a long running HTTP request. Starting with 3.10, the pop-up dialog is no longer used; only the activity indicator displays, and the app goes into background mode, if the HTTP request fails to complete after several minutes. The program gets a runtime error -15553 if the user taps the app icons, to bring the app back to foreground mode.

**OpenID Single sign-on (SSO) protocol not supported**
Support of the OpenID Single sign-on (SSO) protocol has been dropped. This service based on Genero REST is no longer delivered in the Genero Web Services package under $FGLDIR/web_utilities/services/openid.
If you have previously used OpenID to authenticate users launching applications, you must now use an alternate method, such as OpenID Connect. See the OpenID Connect SSO topics in Genero Application Server User Guide.

**Define server configuration based on regular expressions**
Starting with GWS 3.10.09, instead of using the `ws.myident.url` FGLPROFILE entry, you can now use a regular expression to identify several server URLs, by using the `ws.ident.regex.url` entry.
For more details, see Web Services FGLPROFILE: Server configuration.

**fglwsdl option -fRPCNamespace to support namespaces in RPC parameters**
Starting with GWS 3.10.09, the fglwsdl tool supports the new option `-fRPCNamespace` to produce BDL code supporting the `namespace` attribute for RPC parameters.
For more details, see fglwsdl on page 2081.

**fglpass option -gid to allow agent authentication through a UNIX user group**
Starting with GWS 3.10.10, the fglpass tool can be used with the `-gid` to allow agent authentication for all users which belong to the group of the current user executing the fglpass command.
For more details, see fglpass on page 2087, Use the password agent on page 3292.

**Changes to security.global.protocol in FGLPROFILE**
Starting with FGLGWS 3.10.22 GWS secured communication is based on the OpenSSL 1.1 engine. This version of OpenSSL always selects the security protocol. It no longer allows you to specify a specific Transport Layer Security (TLS) or Secure Sockets Layer (SSL).
The `security.global.protocol` entry in the fglprofile file is therefore no longer supported.
For instance, if you have set `security.global.protocol = "TLsv1.2"` to configure OpenSSL to use TLSv1.2 for HTTPS for earlier versions, you may encounter the following error message in your Web service:

```
OpenSSL 1.1 doesn't support specific protocol anymore
```
It is therefore recommended to remove the `security.global.protocol` entry from your fglprofile file. For more information on Web service security configuration, see HTTPS and password encryption on page 3489.

**Related concepts**
Web services on page 3270
Create a Web service client or server with Genero BDL.

**Genero Mobile for Android™ (GMA) 1.30 changes**
Modifications to consider when using the Genero Mobile for Android™.

**Note:** This topic describes feature changes in the GMA 1.30 product. See also the Mobile section in Genero BDL 3.10 New Features page.
**GMA 1.30 with FGLGWS 3.10**

**Important:** The GMA version 1.30 is built on FGLGWS 3.10 and therefore, strongly tied to this Genero BDL version.

**Desupport of GMA front-end application on Google Play**

The GMA front-end app is no longer available for download on Google Play.

In order to install the GMA front-end on your device, use the `gmabuiltool` as described in Genero mobile development client for Android on page 3568.

**ANDROID_SDK_ROOT replaces ANDROID_HOME**

When building apps with `gmabuildtool`, the ANDROID_HOME environment variable is deprecated.

To define the Android SDK installation directory, use ANDROID_SDK_ROOT instead.

If the `--android-sdk` option is not specified, the gmabuildtool will first use ANDROID_SDK_ROOT, then ANDROID_HOME as fallback.

Consider changing your environment settings to follow Android™ SDK specifications.

**Default directory for localized strings in GMA apps**

The .42s localized string files for the default language can be provided in the `appdir/defaults` directory.

Resource files such as .42s provided in `pwd` are always loaded by the runtime system. As GMA, `pwd` and `appdir` are the same, it was not possible to provide default strings files in `appdir`. It was required to provide localized string files for any language, in the corresponding `appdir/locale-code` directories.

For more details, see Localized string files on mobile devices on page 546 and Deploying mobile apps on page 3588.

**New --no-install-extras option**

By default, when updating the Android™ SDK with `gmabuildtool updatesdk`, the process installs also extra SDK modules.

The `--no-install-extras` option of `gmabuildtool updatesdk` can be used to skip installation of extra SDK modules when not needed.

**Unique package for all architectures**

The GMA bundle is now provided as a single package, supporting both ARM and x86 device architectures.

The `--build-types` option of `gmabuildtool build` is no longer available.

**Unique scaffolding archive**

Before GMA 1.30, two GMA binary archives where provided:

- `fjs-gma-*-android-scaffolding.zip`
- `fjs-gma-*-android-extension-project.zip`

Starting with GMA 1.30, the scaffolding and extension project have been merged in a single `fjs-gma-*-android-scaffolding.zip` archive.

For more details, see Executing Java code with GMA on page 2219.

**New scaffold command**

The `gmabuildtool scaffold` command has been added to manage scaffold archives.
In its initial version, the scaffold command provides the --list-plugins option, to show available plugins, and the --install-plugins option, to install plugins in the scaffold archive (for Cordova support).

**No more gma/temp directory**

The Genero Mobile Android™ project directory does not longer need the gma/temp directory to build an app.

Since GMA 1.30, the --build-distribution option of gmabuildtool is no longer available.

**gmabuildtool options use --build-app-genero-program as base directory by default**

Starting with GMA 1.30, the gmabuildtool options listed below use the application program files directory (--build-app-genero-program option) as base directory for their default values.

In previous versions, the default base directory was the current working directory. Note however, that the default value for the --build-app-genero-program option is still the current working directory.

- `-bp / --build-project`
- `-bih / --build-app-icon-hdpi`
- `-bim / --build-app-icon-mdpi`
- `-bixh / --build-app-icon-xhdpi`
- `-bixxh / --build-app-icon-xxhdpi`
- `-bsh / --build-status-icon-hdpi`
- `-bsm / --build-status-icon-mdpi`
- `-bsxh / --build-status-icon-xhdpi`

**GMA scaffolding archive usage (--build-force-scaffold-update option)**

During the manual installation procedure (to build GMA apps from the command line without GST), it was required to unzip the scaffolding zip archive in a dedicated directory (gma-scaffold-project), which could be referenced with the --build-project option of the gmabuildtool build command.

Starting with GMA 1.30, if not yet done, the APK build process will automatically unzip the original GMA scaffold archive (gma-install-dir/artifacts) into the --build-project directory. If needed, you can force a cleanup and update these scaffold files with the --build-force-scaffold-update option.

**New --build-quietly option**

The --build-quietly option of gmabuildtool build allows to build the APK silently, by answered yes to all questions asked during the build process.

**Image.alignment style attribute**

GMA 1.30.01 now supports the alignment style attribute for IMAGE form items.

For the possible values of this attribute, see Image style attributes on page 1198.

**Dangerous permissions no longer set by default**

Starting with GMA 1.30, the Android™ Dangerous Permissions such as android.permission.WRITE_EXTERNAL_STORAGE are no longer set by default when building an APK. Such permissions must be specified explicitly with the --build-app-permissions option of gmabuildtool, depending on the frontcalls used by the app.

For more details, see Android permissions on page 3595.

**Push Notification: Firebase Cloud Messaging replaces Google Cloud Messaging**

Starting with GMA 1.30.18, the Firebase Cloud Messaging framework replaces Google Cloud Messaging to implement push notifications.
What you need to consider:

1. A new Firebase Cloud Messaging project must be created from the Firebase console.
2. The package name of the FCM app must match the package name used to build your APK (with the --build-app-package-name option of `gmabuildtool`)
3. You must download the `google-services.json` configuration file and put it in the `appdir`.
4. The `sender-id` parameter of the `registerForRemoteNotifications`, `getRemoteNotifications` and `unregisterFromRemoteNotifications` frontcalls is no longer needed (all information is in the `google-services.json` parameter file).
5. In the server application sending push notifications, replace the Google API key by the Server Key found in the Firebase Cloud Message project parameters.

For more details, see Push notifications on page 3619.

Setting debug ports with `adb`

The debug ports 6400 and 6480 need to be forwarded with the `adb` command, the `telnet/redir` solution is no longer supported.

For more details, see Debugging on a mobile device on page 2145.

Related concepts

Building Android apps with Genero on page 3591

Genero provides a command-line tool to create applications for Android™ devices.

Genero Mobile for iOS (GMI) 1.30 changes

Modifications to consider when using Genero Mobile for iOS.

Note: This topic describes features changes in the GMI 1.30 product. See also the Mobile section in Genero BDL 3.10 New Features page.

GMI 1.30 with FGLGWS 3.10

Important: The GMI version 1.30 is built on FGLGWS 3.10 and therefore, strongly tied to this Genero BDL version.

Default directory for localized strings in GMI apps

The `.42s` localized string files for the default language can be provided in the `appdir/defaults` directory.

Resource files such as `.42s` provided in `pwd` are always loaded by the runtime system. As GMA, `pwd` and `appdir` are the same, it was not possible to provide default strings files in `appdir`. It was required to provide localized string files for any language, in the corresponding `appdir/locale-code` directories.

For more details, see Localized string files on mobile devices on page 546 and Deploying mobile apps on page 3588.

Building GMI apps with C extensions

Before GMI 1.30, in order to build an iOS app with C extensions, it was required to create your own `makefile` based on the generic `$GMIDIR/lib/Makefile-gmi` file, to add your own C extension libraries or custom front calls with the `USEREXTENSION` variable:

```bash
...
GMI_OPTIONS = \  
   APPNAME=MyApp \  
   BUNDLE_IDENTIFIER=com.mycomany.myapp \  
   IDENTITY="iPhone Developer" \\  
```
Starting with GMI 1.30, it is now possible to build your static library with the `staticlib` target of `$GMIDIR/lib/Makefile-gmi`, and pass your library to the `gmibuildtool` with the `--extension-libs` option:

```
$ gmibuildtool ... -extension-libs "-lz libBPush.a" ...
```

For more details, see Building iOS apps with Genero on page 3606.

### Registering custom front calls in GMI

Before GMI 1.30, it was required to implement a `frontCalls()` function, in order to declare the Objective-C class implementing your custom front calls:

```
NSArray* frontCalls()
{
    return @[
        [ExtensionFrontCall class]
    ];
}
```

Starting with GMI 1.30, upon startup, the GMI detects the extension by enumerating all descendants of the Frontcall classes.

Thus, there is no need to implement this function anymore.

For more details, see Implement front call modules for GMI on page 2259.

### Managing plugins in GMI

Starting with GMI 1.30, the `gmibuildtool` allows you to handle plugins with the `--install-plugins` and `--list-plugins` options.

For more details, see Cordova plugins on page 3644.

### C Extension changes

Modifications to consider when using C Extensions

#### Informix ESQL/C header files no longer distributed

Prior to version 3.10.11, Informix ESQL/C header files such as `decimal.h` were provided in `$FGLDIR/include/esql` directory.

Starting with BDL 3.10.11, the Informix ESQL/C header files are no longer distributed in the BDL packages: Genero provides its own header files defining the C structures for `DECIMAL`, `DATETIME`, `INTERVAL` and `TEXT/BYTE` types.

Simply include the `fglExt.h` header file in your C extension source.

If you need additional type definitions that are not provided in Genero C Extension header files, install the latest Informix CSDK, and include the Informix header files before `fglExt.h`.

For more details, see Header files for ESQL/C typedefs on page 2230.
C Extension API functions for bigint
Genero BDL 2.51 has desupported C Extension stack functions, that were introduced again in 3.10 (also backported in 3.00.10).

The following C API functions are available again:
• popbigint(bigint *dst)
• pushbigint(bigint val)

The following C type definitions are available again:
• bigint: Defines a 8-byte signed integer

For more details, see Runtime stack functions on page 2235

Related concepts
C-Extensions on page 2230
With C-Extensions, you can bind your own C libraries in the runtime system, to call C function from the application code.

Java Interface changes
There are changes in support of Java in Genero 3.10.

Genero BDL supports Java SE 6 to Java SE 10 and 11
Starting with version 3.10.14, Genero BDL supports now Java SE 10 and 11 when using the Java Interface.

Important: The minimal required version to use the BDL Java Interface is Java SE 6.
For more details, see Java software requirements for FGL on page 2193.

Related concepts
The Java interface on page 2192
The Java interface allows you to import Java classes and instantiate Java objects in your programs.

Presentation styles changes
Modifications to consider when using presentation styles.

Edit.dataTypeHint style attribute is deprecated
Starting with Genero BDL 3.10, the Edit presentation style attribute dataTypeHint is deprecated.
Consider using the KEYBOARDHINT form attribute instead.
Reference: Edit style attributes on page 1194.

Image.alignment style attribute
The alignment style attribute for IMAGE form items is now supported by:
• GMA 1.30.01
For the possible values of this attribute, see Image style attributes on page 1198.

TextEdit.customWidget style attribute
With Genero BDL 3.10, the TextEdit presentation style customWidget is no longer required with the GBC front-end.
This style attribute was supported by GWC-HTML5, when using textFormat="html".
Reference: TextEdit style attributes on page 1212.
Note: Genero 3.10 introduces the fglrichtext web component, an alternative to the richtext option in TEXTEDIT fields, which is supported by all front-ends.

**Window.commentPosition style attribute is deprecated**

Since Genero BDL 3.10, the Window presentation style `commentPosition` is deprecated. This style attribute is only supported by the GDC front-end.

Reference: Window style attributes on page 1220.

**Window.actionPanelButtonSize and Window.ringMenuButtonSize style attributes**

Starting with Genero BDL 3.10, the action panel and ring menu button sizes can be defined by specifying a size relative to the height of the current font, using the "em" unit, for example:

```
<StyleAttribute name="actionPanelButtonSize" value="20em" />
```

Note: The abstract values such as "small", "medium", "large" are supported for backward compatibility. Reference: Window.actionPanelButtonSize, Window.ringMenuButtonSize.

**Table.allowWebSelection with GBC**

To select table/tree content with the GBC front-end, by default, the user must press the CTRL key and do a click-and-drag. Starting with GBC 1.00.28, you can use the "allowWebSelection" style attribute, to allow content selection from a table (or a treeview), with a simple click-and-drag (without pressing the CTRL key):

```
<StyleAttribute name="allowWebSelection" value="yes" />
```

Note: In GWC-HTML5, the "allowWebSelection" style attribute was supported for the Window elements, to allow the user to copy the content of disabled fields to the clipboard. Text selection and copy in disabled fields is allowed by default with GBC.

Reference: Table style attributes on page 1204, Tree style attributes on page 1209.

**Table.resizeFillsEmptySpace and Table.resizeFillsEmptySpace with GBC**

Starting with GBC 1.00.33, the `resizeFillsEmptySpace` style attribute can be used for Table and Tree elements:

```
<StyleAttribute name="resizeFillsEmptySpace" value="yes" />
```

Reference: Table style attributes on page 1204, Tree style attributes on page 1209.

**Window.tabbedContainer with GBC**

Since GBC 1.00.34, the `tabbedContainer` style attribute can be used to render hosted applications embedded in a tab container, as was done with GWC-HTML5:

```
<StyleAttribute name="tabbedContainer" value="yes" />
```

If the window of an application uses `tabbedContainer=yes`, every subsequent application is embedded in a tab container generated by GBC. When using this style attribute, the GBC side bar is replaced by Start menus on page 2019, which is mandatory when `tabbedContainer=yes`.

Reference: Window.tabbedContainer style attribute.
Window.position value "center" with GBC

Starting with GBC 1.00.35, you can use the window position "center" for the position style attribute:

```xml
<StyleAttribute name="position" value="center"/>
```

**Note:** Using the center position mimics HTML5 centering windows when setting the sizable style attribute to "no". The main difference with HTML5 is that centering was the default behavior with HTML5. With GBC, it has to be specified explicitly.

Reference: [Window.position style attribute](#).

Window.sizable with GBC for modal windows

Since GBC 1.00.37, the sizable style attribute can also be used for modal windows:

```xml
<StyleAttribute name="sizable" value="no"/>
```

Reference: [Window.sizable style attribute](#).

**CheckBox.customWidget style attribute**

Starting with GBC 1.00.38, the CheckBox presentation style attribute customWidget can be defined to "toggleButton", to get a toggle switch rendering:

```xml
<StyleAttribute name="customWidget" value="toggleButton"/>
```

Reference: [CheckBox style attributes](#) on page 1187.

**DateEdit.buttonIcon and DateTimeEdit.buttonIcon style attributes**

Starting with GBC 1.00.41, the buttonIcon presentation style attribute can be used for DateEdit and DateTimeEdit classes, to defined the icon of the calendar button.

Reference: [DateEdit style attributes](#) on page 1190, [DateTimeEdit style attributes](#) on page 1191.

**DateEdit.calendarType style attribute**

Since GBC 1.00.45, the DateEdit presentation style attribute calendarType altered its behaviour for the value modal. When set to modal, the user must use the calendar to validate (or cancel) a date change, and clicking outside the calendar closes the calendar. Previous behavior included a greyed overlay when the calendar was opened as modal, and clicking outside the calendar did nothing.

Reference: [DateEdit style attributes](#) on page 1190.

**Folder.position value "accordion"**

Starting with GBC 1.00.47, you can use the value "accordion" for the position style attribute of FOLDER containers, to render the children pages as a set of collapsible group boxes.

**Note:** The "accordion" position is only supported by the GBC front-end.

Reference: [Folder style attributes](#) on page 1195.
Front calls changes
Modifications to consider when using front calls.

**session.* front call are desupported**

The `session.setVar` and `session.getVar` front calls are no longer supported in Genero 3.10: The `session` front calls rely on the GAS "wa" protocol (html5Proxy). This protocol is only supported only by GWC-HTML5 client. html5Proxy is no longer delivered in Genero 3.10.

In order to store information on the front-end side, use the `localStorage` front calls. For more details, see New `localStorage` front calls on page 149.

**standard.setWebComponentPath**

The `standard.setWebComponentPath` front call is deprecated in Genero BDL 3.10. To deploy your web components, consider using the solutions described in Deploying the `gICAPI` web component files on page 1947.

**standard.feInfo: outputMap**

Starting with Genero BDL 3.10, the `outputMap` information can no longer be used with the `standard.feInfo` front call.

**standard.cbSet supported by GBC**

Since GBC 1.00.35, the GBC supports the `standard.cbSet` front call.

**mobile.isForeground to check app foreground mode**

Since Genero BDL 3.10.11 (GMA 1.30.10, GMI 1.30.11), the `mobile.isForeground` front call can be used to check if the app is in foreground mode.

For more details, see `mobile.isForeground` front call.

Related concepts

Front calls on page 583
Front call functions execute on the platform where the front-end is installed.

Web components on page 1922
This section describes how to use web components in your application.

Web components changes
Modifications to consider when using web components.

Only SIZEPOLICY=FIXED is supported for WEBCOMPONENT fields

When defining a WEBCOMPONENT field in the form, the only valid option for the SIZEPOLICY attribute is FIXED. This is by the way the default for WEBCOMPONENT fields.

For more details, see Controlling the web component layout on page 1924.

fglrichtext: New implementation

Up to FGLGWS version 3.10.09 (WCG 1.00.11), the fglrichtext built-in web component was based on draft.js Rich Text Editor Framework.

Starting with FGLGWS 3.10.11 (WCG 1.00.12), the fglrichtext web component is implemented with the quill.js Rich Text Editor.

Available features and configuration options have changed.

For more details, see The `fglrichtext` web component on page 1972.
**fglrichtext: Support of emojis**

Starting with FGLGWS version 3.10.13 (WCG 1.00.13), the emoji toolbar option allows users to include an emoji in the text of an fglrichtext web component.

**Important:** Since emojis are Unicode characters and not plain images, your application needs to use UTF-8. Emojis rendering will differ depending on the device you're running it on. You might need to install fonts that handle emoji characters as well if your device doesn't handle it.

For more details, see Configuring the toolbar on page 1974.

**fglrichtext: Defining a default font**

Starting with FGLGWS version 3.10.16 (WCG 1.00.15), a default font family and font size can be specified for a fglrichtext web component with new properties.

For more details, see Specifying the default font on page 1977.

**fglrichtext: Localizing texts**

Starting with FGLGWS version 3.10.16 (WCG 1.00.15), it is possible to localize the texts used by the fglrichtext web component.

Several properties have been added to the fglrichtext web component, to define the strings for each toolbar button tooltips, combobox items, and popup dialog labels.

For more details, see Localizing rich text component strings on page 1977.

**fglsvgcanvas: Mouse hovering events**

Starting with FGLGWS version 3.10.11 (WCG 1.00.12), the fglsvgcanvas built-in web component supports new properties to detect onmouseover/onmouseout mouse hovering SVG events, and trigger ON ACTION blocks.

For more details, see The fglsvgcanvas web component on page 1988.

**fglsvgcanvas: title() and createChars() functions**

Starting with FGLGWS version 3.10.16 (WCG 1.00.15), the fglsvgcanvas library provides the title() function to create a title SVG element, and the createChars() function to create a text node.

For more details, see fglsvgcanvas: SVG drawing module on page 2332.

**fglgallery: Aspect ratio for image elements**

Using FGLGWS version 3.10.18 (WCG 1.00.16), image elements are now aligned properly by using the same size, in case if picture resources have different sizes.

By default, the images are displayed with a square (1:1) aspect ratio. To define a different aspect ratio, the fglgallery library provides the setAspectRatio() function.

For more details see fglgallery image aspect ratio handling.

**fglgallery: Custom CSS file handling**

Starting with FGLGWS version 3.10.18 (WCG 1.00.16), to avoid missing resource errors in web viewers, the custom CSS file $FGLDIR/webcomponents/fglgallery/css/fglgallery-custom.css is no longer included by default in the fglgallery.html file.

If you want to provide you own CSS file for fglgallery, uncomment the <link /> line in fglgallery.html.

Note also that CSS names have changed, since the HTML content of fglgallery has been reviewed for images aspect ratio handling.
Debugging web components with GDC

Starting with GDC 3.10.24, the new command line option `--webengine-remote-debugging=[address]:[port]` can be used to define the debug port for QT WebEngine Developer Tools, like when using the QTWEBENGINE_REMOTE_DEBUGGING environment variable.

For more details, see Debugging a web component on page 1928.

Related concepts

Web components on page 1922
This section describes how to use web components in your application.

Database drivers changes

New and desupported database drivers.

New database server versions supported in Genero 3.10:

Support for new database server releases, with existing ODI drivers:

- PostgreSQL 10 is supported with the existing `dbmpgs_9` driver (linked to `libpq.so.5`).

Support for new database server and client releases with new ODI drivers:

- Microsoft® ODBC for SQL Server v13 (`dbmsnc_13`) and v17 (`dbmsnc_17`).
- Oracle® MySQL 5.7 (`dbmmys_5_7`) and MySQL 8.0 (`dbmmys_8_0`).
- Oracle® 12c on macOS® with `dbmora_12`.
- MariaDB 10.2 with `dbmmdb_10_2`.
- SAP HANA® 2.0 database with `dbmhdb_2`.

Databases / ODI drivers desupported in Genero 3.10:

Database drivers for old database client versions are removed in accordance with DB vendor de-support plans:

- SQL Server 2005 / SQL Native Client V9 (`dbmsnc_9`).
- Oracle® MySQL 5.1.x (`dbmmys_5_1`)

Related concepts

Database driver specification (driver) on page 662

(GDC) Local Actions

The concept of "Local Actions" is now deprecated.

Starting with Genero 3.10, the concept of "Local Actions" is deprecated.

Local Actions are available in the GDC front-end only, to bind local operations to action views.

For example, the typical copy/cut/paste operations can be bound to ToolBar items by using the `editcopy`, `editcut`, `editpaste` action names.

Local Actions can be considered as an over-engineered feature. Business applications do not have to implement basic copy/cut/paste editor actions. The user interface must focus on application functions (create new record, print order, etc).

Related concepts

Dialog actions on page 1739
Describes how to program action handling when the end user triggers an action on the front-end.

Toolbars on page 1436
Toolbars define a bar of buttons that appears at the top of application forms.

Configuring actions on page 1744
Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with *action attributes*.

**Microsoft ODBC Driver for SQL Server**
Support for Microsoft® ODBC Driver for SQL Server

**Feature summary**
Genero BDL 3.10 now supports Microsoft® ODBC Driver v13 and v17 for SQL Server, respectively with the dbmsnc_13 and dbmsnc_17 ODI drivers.

These ODI drivers are available on Windows® and Linux® platforms, to connect to:

- Microsoft SQL Server 2016,
- Microsoft SQL Server 2017,
- Microsoft Azure SQL Database.

**Using Microsoft® ODBC on Windows® platforms**
The new ODI drivers for Microsoft SQL Server can be used to connect from Windows platforms to Microsoft SQL Server 2016, 2017 and Azure DB.

The Windows ODI drivers for Microsoft ODBC for SQL Server need the following DLLs:

- The dbmsnc_13 ODI driver requires MSODBCSQL13.DLL.
- The dbmsnc_17 ODI driver requires MSODBCSQL17.DLL.

**Note:** Microsoft ODBC 11 for SQL Server (MSODBCSQL11.DLL) is not supported.

For backward compatibility, the dbmsnc_11 driver using Microsoft SQL Native Client 11 (SQLNCLI11.DLL) is still supported. However, it is recommended to upgrade to the dbmsnc_13 or dbmsnc_17 drivers using Microsoft ODBC for SQL Server.

**Important:**
With Genero BDL 3.10, the generic ODI driver named dbmsnc maps to the most recent SNC driver dbmsnc_17, requiring Microsoft ODBC 17 for SQL Server (MSODBCSQL17.DLL) in the ODBC data source definition. If you do not set up the ODBC data source configuration to use MSODBCSQL17.DLL, the SNC ODI driver will report an invalid ODBC driver error.

With Genero 3.00, you still have to use dbmsnc_11 to connect to SQL Server 2016. For a cross-version configuration of Genero, use explicit ODI driver names dbmsnc_11, dbmsnc_13 or dbmsnc_17, instead of the generic driver name "dbmsnc". This is required because dbmsnc maps to dbmsnc_17 in BDL 3.10 and dbmsnc maps to dbmsnc_11 in BDL 3.00.

**Using Microsoft® ODBC on Linux® platforms**
The new ODI drivers for Microsoft SQL Server can be used to connect from Linux platforms to Microsoft SQL Server 2016, 2017 and Azure DB.

The Linux ODI drivers for Microsoft ODBC for SQL Server are linked to the following shared libraries:

- The dbmsnc_13 ODI driver requires libmsodbcsql-13.so.
- The dbmsnc_17 ODI driver requires libmsodbcsql-17.so.

On Linux® platforms, the dbmsnc_nn drivers are directly linked to the corresponding libmsodbcsql-nn.so ODBC driver library. There is no need to install the unixODBC software. The SNC drivers will be able to connect to SQL Server, as long as the dynamic linker can find the Microsoft ODBC driver library. The libmsodbcsql-nn.so shared library is a symbolic link located in /usr/lib64, which points to the real ODBC 13 shared library.
Azure SQL Database

The ODI drivers for Microsoft ODBC for SQL Server can be used to connect to Microsoft Azure SQL Databases, from Windows or Linux platforms.

To establish a TCP connection to a server created through the Azure portal, you need to configure the firewalls on the server side and on your computer and network.

For more details, check http://azure.microsoft.com/en-us/.

Related concepts
Microsoft SQL Server on page 659
Database driver specification (driver) on page 662

Unique TABINDEXes in a form
The TABINDEX values must be unique in a given form file.

Starting with Genero 3.10, the TABINDEX attribute must be unique in a form layout:

```
LAYOUT
GRID
{
  [f1     ]
  [f2     ]
  [f3     ]
}
END
END
ATTRIBUTES
EDIT f1 = FORMONLY.cust_id;
EDIT f2 = FORMONLY.cust_name, TABINDEX=2;
EDIT f3 = FORMONLY.cust_address, TABINDEX=2;
# TABINDEX has to be unique.
# See error number -6847.
END
```

Related concepts
Defining the tabbing order on page 1729
Control the order of tabbing through the fields with the TABINDEX attribute.

TABINDEX attribute on page 1395
The TABINDEX attribute defines the tab order for a form item.

Optional SCREEN RECORD size for lists
A SCREEN RECORD definition can omit the number of rows of the corresponding list container.

Starting with Genero 3.10, a SCREEN RECORD definition can be defined without the size of the corresponding list container (TABLE, SCROLLGRID, TREE or static field list).

When specifying the size in SCREEN RECORD, it must match the exact number of rows of the corresponding list container, otherwise fglform will throw error -2029:

```
LAYOUT
GRID
{
  <TABLE t1     >
  [c1  |c2       ]
  [c1  |c2       ]
  <     |     >
}
END
END
ATTRIBUTES
```
c1 = FORMONLY.cust_id;
c2 = FORMONLY.cust_name;
END
INSTRUCTIONS
SCREEN RECORD sr_cust[10](FORMONLY.*);
  # Screen record array 'sr' has different component sizes.
  # See error number -2029.
END

Related concepts
Binding tables to arrays in dialogs on page 1857
Program arrays act as data model that are bound to form tables, when implementing list dialogs.

Defining tables in the layout on page 1854
Define table views in the LAYOUT section of the form definition file.

Type checking with fglcomp compiler
The fglcomp compiler is more strict regarding type checking.

Starting with Genero 3.10, the fglcomp compiler is now more strict when assigning complex types.
Any assignment potentially throwing runtime error -1260 is subject to the new compiler error -6631.

Note: Assignments using primitive types such as CHAR to INTEGER are not checked, except for DATETIME, INTERVAL, TEXT and BYTE assignments, which are candidates for runtime error -1260.

Type checking errors are raised on:
- incompatible assignments in LET variable = value.
- passing incompatible values to functions (user functions, built-in functions, C extensions).
- returned types of functions when the FUNCTION is defined with the RETURNS clause.

Better type checks include:
- better predictions (for autocompletion in vim and Studio)
- being able to call methods on return values of methods.

For example, when assigning invalid object references to variables defined with a different class:

```
$ cat tcl.4gl
MAIN
  DEFINE sb base.StringBuffer
  LET sb = "foo" -- illegal: assigns a String to a StringBuffer
END MAIN
```

With versions prior to 3.10, you get no compiler error, but an error at runtime:

```
$ fglcomp -V
fglcomp 3.10.14
... $ fglcomp tcl
$ fgldr tcl
Program stopped at 'tcl.4gl', line number 3.
FORMS statement error number -1260.
It is not possible to convert between the specified types.
```

With version 3.10, you get now a compiler error:

```
$ fglcomp -V
fglcomp 3.10.03
... $ fglcomp -M tcl
```
In the following example, the type returned by `base.Channel.create()` does not match the variable definition:

```
$ cat tc2.4gl
MAIN
  DEFINE sb base.StringBuffer
  LET sb = base.Channel.create() -- what's the return type of base.Channel.create()?  
END MAIN

$ fglcomp -M tc2
```

This example tries to pass a `STRING` to a method whereas an `om.DomNode` is expected:

```
$ cat tc3.4gl
MAIN
  DEFINE doc om.DomDocument
  DEFINE n1, n2 om.DomNode
  LET doc = om.DomDocument.createFromString("<Foo><Bar/></Foo>")
  LET n1 = doc.getDocumentElement()
  LET n2 = doc.getDocumentElement().getFirstChild()
  LET n2 = n1.getFirstChild()
  CALL n1.removeChild(n2) -- legal: n2 is an om.DomNode
  CALL n1.removeChild("Bar") -- illegal: n2 is a STRING
END MAIN

$ fglcomp -M tc3.4gl
```

This example uses the return value of FGL-functions:

```
$ cat tc4.4gl
MAIN
  DEFINE s STRING
  LET s = function1() -- unchecked: return type unknown
  LET s = function2() -- illegal: returns a base.Channel
  LET s = function3() -- legal: returns a STRING
END MAIN

FUNCTION function1()
  RETURN base.Channel.create()
END FUNCTION

FUNCTION function2() RETURNS base.Channel
  RETURN base.Channel.create()
END FUNCTION

FUNCTION function3() RETURNS STRING
  RETURN "foo"
END FUNCTION

$ fglcomp tc4.4gl
The compilation was not successful. Errors found: 1.
The file 'tc4.err' has been written.

$ cat tc4.err
MAIN
DEFINE s STRING
LET s = function1() -- unchecked: return type unknown
LET s = function2() -- illegal: returns a base.Channel
incompatible types, found: base.Channel, required: STRING.
See error number -6631.
LET s = function3() -- legal: returns a STRING
END MAIN

Related concepts
Primitive Data types on page 289
Selecting the correct data type assists you in the input, storage, and display of your data.

fglcomp on page 2071
The fglcomp tool compiles .4gl source files into .42m p-code modules, and does various other tasks.

Calling functions on page 442
Functions can be invoked, to execute the code they define.

Function references on page 449
Function can be referenced and invoked dynamically in a CALL instruction, or in an expression.

Oracle DB Proxy Authentication
Specifying a proxy user when connecting to Oracle® DB.

Oracle® DB supports proxy authentication, a feature that allows a given DB user "A" to connect with the credentials of another DB user "B" and to be seen as user "A" for the rest of the SQL session.

In order to link a proxy user to another user, issue the following command:

ALTER USER app_user GRANT CONNECT THROUGH proxy_user

An application can then connect with the credentials of proxy_user, and be identified as app_user during the SQL session.

See Oracle® DB documentation for more details about proxy authentication.

Starting with Genero 3.10, it is possible to specify a "proxy client" at connection time, by using the /PROXY_CLIENT:user_name suffix in the user name parameter of the CONNECT TO instruction.

In the example, the application connects with the proxy user credentials, and the proxy client is defined as app_user:

CONNECT TO "myserver"
   USER "proxy_user/PROXY_CLIENT:app_user"
   USING "proxy_pswd"

A subsequent SELECT USER FROM dual would return "APP_USER".

Related concepts
Database users on page 917

Temp table emulation with Oracle DB
Controlling the Oracle® schema and the tablespace for tables created by Informix® temporary table emulation.

In order to support Informix® CREATE TEMP TABLE and SELECT ... INTO TEMP statements, the Oracle® ODI driver converts the SQL text to Oracle® syntax. Two emulation methods are supported by Genero: "default" and "global" emulation. First method creates regular tables, while second method uses Oracle® global temporary tables. The emulation method used is defined by the following FGLPROFILE entry:

dbi.database.mydb.ifxemultemptables.emulation = "default" the "global"
If your Genero application requires real Oracle® database users for each end user, you need to specify the common schema where application tables are defined, to access them without a schema prefix in SQL statements. You typically do this with the following FGLPROFILE entry:

```
    dbi.database.mydb.ora.schema = "app_owner"
```

However, if your application programs create Informix-style temporary tables, each Oracle® DB user needs CREATE ANY TABLE and DROP ANY TABLE privileges to create the tables in the common schema. This is not suitable in an organization using a strong security policy.

The Oracle® tables created by the "default" emulation are by default created in the TEMPTABS tablespace, while global temporary tables are created in the TEMPTABS schema. Before Genero 3.10, it was not possible to control this.

Starting with Genero 3.10, it is possible to specify the schema and the tablespace where Oracle® tables are created for temporary table emulation:

```
    dbi.database.dbname.ora.temptables.schema.source = { "login" | "command" }
    dbi.database.dbname.ora.temptables.schema.command = "select-statement"
    dbi.database.dbname.ora.temptables.tablespace = "tablespace-name"
```

These FGLPROFILE parameters apply to both default and global temporary table emulation methods. When using "default" emulation, you typically configure these entries to add the current user name as schema for the CREATE TABLE Oracle® statements and avoid granting CREATE/DROP ANY TABLE privileges to all users. With the "global" emulation, you can specify a common schema (different from the default TEMPTABS schema), to share the global temporary table among several database users.

**Related concepts**

Temporary tables on page 943  
Oracle DB specific FGLPROFILE parameters on page 673

**Oracle® MySQL 5.7 and 8.0 support**
Support for Oracle® MySQL version 5.7 and 8.0

**ODI driver for Oracle® MySQL 8.0**
Genero BDL 3.10.08 now supports Oracle® MySQL 8.0 with the dbmmys / dbmmys_8_0 driver.  
**Important:** MySQL 8.0 uses the version number .21 for the libmysqlclient.so library. The dbmmys generic driver name maps now to the dbmmys_8_0 driver, which is linked to libmysqlclient.so.21. In prior Genero versions, the generic driver name dbmmys was an alias for dbmmys_5_5 or dbmmys_5_7. When using MySQL versions older than 8.0, you need to use the exact driver name with the version number. See Database driver specification (driver) on page 662 for more details.

**ODI driver for Oracle® MySQL 5.7**
Genero BDL 3.10 supports Oracle® MySQL 5.7 with the dbmmys_5_7 driver.  
**Important:** MySQL 5.7 uses the version number .20 for the libmysqlclient.so library. The dbmmys_5_7 driver is linked to libmysqlclient.so.20. With MySQL 5.7, you need to use the dbmmys_5_7 driver. See Database driver specification (driver) on page 662 for more details.

**Oracle® MySQL JSON data type**
MySQL 5.7 introduces the JSON data type to store and handle JSON documents.  
This new type can be used in Genero BDL programs, by using TEXT data type:

```
    MAIN
        DEFINE p_js TEXT
```
LOCATE p_js IN MEMORY
CONNECT TO "test1+driver='dbmmys'") USER "mysuser" USING "fourjs"
EXECUTE IMMEDIATE "create table t1 ( pk int, js json )"
LET p_js = '{"id": "9999", "name": "Tom Baker"}'
INSERT INTO t1 VALUES ( 1, p_js )
LET p_js = NULL
SELECT js INTO p_js FROM t1 WHERE pk = 1
DISPLAY p_js
END MAIN

When producing a database schema file from a MySQL 5.7+ database, the fgldbsch tool will convert JSON columns to TEXT.

Related concepts
Oracle MySQL / MariaDB on page 876
What is JSON? on page 575
JSON (JavaScript Object Notation) is a well known lightweight data-interchange format for JavaScript.

fgldbsch on page 2079
The fgldbsch tool generates the database schema files from an existing database.

fglhint_* in SQL comments
Using SQL comment hints to control statement execution.

Starting with version 3.10, you can now specify fglhint_* keywords in C-style SQL comments, to give an indication to the database driver about the type of SQL statement to be executed.

Note: C-style SQL comments can only be used in dynamic SQL statements and SQL blocks.

For example, to force an INSERT statement to be treated as a regular SELECT returning a result set, use the fglhint_select hint in a C-style comment:

DECLARE c1 CURSOR FROM "/* fglhint_select */ INSERT INTO table1 OUTPUT INSERTED.* SELECT * FROM customers"

Important: If you are using /* */ comments, these will now be parsed and any unknown keyword will be ignored: Comments such as /* INSERT */ or /* SELECT */ must be replaced by /* fglhint_insert */ and /* fglhint_select */ respectively.

Furthermore, Informix® emulation can be disabled with the fglhint_no_ifxemul hint.

Related concepts
fglhint_* SQL comments on page 714
Using special SQL comment hints to control statement execution.

Static SQL statements on page 694
Describes static SQL statements supported in the language.

Dynamic SQL management on page 709
Explains how to execute and manage SQL statements at runtime.

SCROLLGRID content is checked by fglform
When using a SCROLLGRID, fglform compiler checks that it does not hold other list containers.

Before version 3.10, fglform on page 2068 did not give an error when using TABLE, TREE or SCROLLGRID as sub-elements in a SCROLLGRID container, for example:

SCROLLGRID gr2
{<TABLE tb4 >
[ab8 ]
}
This layout construction is illegal: It can not be rendered by the GUI and causes undefined front-end behavior or crash.

Starting with 3.10, fglform gives a compilation error -6846 , if SCROLLGRID contains elements other than simple form fields or labels.

**Related concepts**

SCROLLGRID container on page 1325
Defines a scrollable grid view widget, in a grid-based layout.

**Default resource file search path**
Search rules for program resource files have been enhanced in 3.10.

**Default search in application directory**
Starting with version 3.10, if resource files such as .42f form files are not found in the current working directory, or in the directories specified in the FGLRESOURCEPATH environment variable, or in $FGLDIR/lib, the runtime system also does a lookup in the directory where the MAIN .42m module or the .42r program file resides.

As a result, if all program files are in the directory where the main program module resides, there is no need to set FGLRESOURCEPATH.

For more details about resource file directory search, see FGLRESOURCEPATH on page 279.

**Form file path using absolute path**
If the file name specified in OPEN FORM / OPEN WINDOW WITH FORM is an absolute path, the runtime system must not search for the form in FGLRESOURCEPATH.

Before version 3.10, to load a form file specified with an absolute path in OPEN FORM or OPEN WINDOW WITH FORM, the runtime system tried to find the file by appending the absolute path to the directories defined by FGLRESOURCEPATH.

This could end up in loading unexpected resource files.

For example, with:

```
OPEN FORM f FROM "/dir/form"
```

The form "/dir/form" was searched in each element of FGLRESOURCEPATH.

Since version 3.10, FGLRESOURCEPATH is only used, if the file name is not an absolute path.

For more details, see WITH FORM clause on page 1134.

**Related concepts**

Providing the image resource on page 1149
There are several things you need to know about providing an image resource in a Genero program.

os.Path.pathType on page 2805
Checks if a path is a relative path or an absolute path.

**Using os.Path.pathType() on Windows**
How to identify absolute file paths on Windows®, whether using a drive letter or not.

Before version 3.10, on Windows® platforms, a file name starting with a directory-separator (slash or backslash) was not interpreted by os.Path.pathType() as an absolute name. As result, such files where considered as relative path and searched by concatenating the path to the elements defined in FGLLLDPATH, FGLRESOURCEPATH, FGLIMAGEPATH.
Starting with version 3.10, a file name starting with a directory-separator or starting with a drive letter, is considered an absolute file path.

This change has also an impact on the `os.Path.join` on page 2804 method, which returns the second parameter only if it is identified as an absolute path.

**Related concepts**

- **FGLLDPATH** on page 278
  Defines the search paths to load program modules.
- **FGLRESOURCEPATH** on page 279
  Defines search path for resource files.
- **FGLIMAGEPATH** on page 276
  Defines the search paths for VM server image files.
- **os.Path.pathType** on page 2805
  Checks if a path is a relative path or an absolute path.

**New localStorage frontcalls**

New localStorage frontcalls replace GAS specific `session.setVar` and `session.getVar` calls.

The `session` module front calls `setVar` and `getVar` are deprecated. Starting with version 3.10, you can use the new localStorage front calls, supported by all Genero front-ends.

For more details, see [Local storage front calls](#) on page 2695.

**Related concepts**

- **Front calls** on page 583
  Front call functions execute on the platform where the front-end is installed.

**Wide Char mode of SNC driver**

The `snc.widechar` FGLPROFILE entry defaults to the right setting for the current application locale.

SQL Server performs better, when using the correct ODBC types in `SQLBindParameter()`, depending on the database columns types:

- When using CHAR/VARCHAR/TEXT SQL types (typically, when the application locale uses a single-byte encoding), the recommendation is that ODBC SQL types SQL_[VAR]CHAR (single-char ODBC SQL types) are used
- When using NCHAR/NVARCHAR/NTEXT SQL types (when the application locale uses a multibyte encoding such as UTF-8), it is recommended to use the ODBC SQL types SQL_W[VAR]CHAR (these are wide-char ODBC SQL types)

The `dbi.database.dbname.snc.widechar` FGLPROFILE entry controls the ODBC SQL types used by the SNC driver.

Before version 3.10, the `snc.widechar` option was set to `true` by default, assuming that the application locale uses a multibyte encoding like UTF-8. If the database columns are defined with CHAR/VARCHAR/TEXT types, this parameter had to be set to `false`.

Starting with 3.10, to simplify application configurations when using a single-byte encoding, the `snc.widechar` option defaults to the expected setting, based on the current application locale (assuming that the database column types fit the application locale):

- If the application locale defines a single-byte encoding (such as ISO8859-1), we assume that the database columns are defined with CHAR/VARCHAR/TEXT types to store single-byte characters, and the SNC driver will use SQL_[VAR]CHAR.
- If the application locale defines a multibyte encoding (such as UTF-8 or BIG5), we assume that the database columns are defined with NCHAR/NVARCHAR/NTEXT types to store UNICODE characters, and the SNC driver will use SQL_W[VAR]CHAR.

**Note:** Set the `dbi.database.dbname.snc.widechar` to `false`, only if you are using a multibyte encoding such as BIG5, with CHAR/VARCHAR/TEXT column types in the database.
Note: Easysoft (ESM) and FreeTDS (FTM) drivers do not support the snc.widechar option: The SQL char type binding mode is automatic, depending on the current application locale, and it cannot be changed.

Related concepts
CHAR and VARCHAR data types on page 838
The FGLPROFILE file(s) on page 255
FGLPROFILE environment variable defines Genero BDL configuration files
Multibyte character sets (MBCS) on page 517

MariaDB 10.2 support
The new ODI driver dbmmdb_10_2 is provided to connect to MariaDB 10.2.

MariaDB 10.2 comes with a new client library name libmariadb.so. In prior versions of MariaDB, the client library name was libmysqlclient.so.

Starting with Genero 3.10, MariaDB 10.2 is supported by using the new dbmmdb_10_2 ODI driver. This driver is linked to libmariadb.so.3, the MariaDB 10.2 client library.

Important: The driver-specific FGLPROFILE entries must use the "mdb" code, for example:

```plaintext
dbi.database.test1.mdb.config = "/opt/var/app/my.cnf"
```

To connect to older MariaDB version 10.0 and 10.1, you can still use the dbmmys_5_5 ODI driver, linked to the libmysqlclient.so.18 shared library, provided in MariaDB 10.0 and 10.1 distributions.

Important: The driver-specific FGLPROFILE entries must use the "mys" code, for example:

```plaintext
dbi.database.test1.mys.config = "/opt/var/app/my.cnf"
```

Related concepts
Database driver specification (driver) on page 662

Related tasks
Prepare the runtime environment - connecting to the database on page 877

Program stop error message box
In GUI mode, runtime errors stopping the program are now displayed to the end user.

Starting with Genero 3.10.11, when using the GUI mode, runtime errors that stop the program execution are displayed to the end user in a pop-up message box.

Tip: Runtime errors stopping the program execution are either non-trappable errors, or trappable errors that occur in code pieces not protected by WHENEVER or TRY/CATCH.

It is now much easier to identify the reason of the error stopping the program. Before this feature, a error stopping the program could not be seen by the end user, and the program windows just disappear with any further feedback. The only way to identify the problem was to inspect the fglrun output or errorlog on the application server.

The end user can now take a screenshot and report the issue to the application provider.

Note: This new behavior is also available with older front-end versions.

Feature extension: Starting with version 3.20.11, an FGLPROFILE entry can define a generic/static message to be displayed to the end user, to overwrite the original error message and hide issue details. See Default exception handling on page 508.

Related concepts
Non-trappable errors on page 509
Non-trappable errors are fatal errors that generally prevent further program execution.

WHENEVER instruction on page 503
Use the `WHENEVER` instruction to define how exceptions must be handled for the rest of the module.

**TRY - CATCH block** on page 506
Use `TRY / CATCH` blocks to trap runtime exceptions in a delimited code block.

**startlog()** on page 2298
Initializes error logging and opens the error log file passed as the parameter.

**BUTTONEDIT and ON ACTION INFIELD**
With **ON ACTION INFIELD**, a BUTTONEDIT action is now always considered as a field-qualified action.

Before version 3.10, when implementing the action handler for a BUTTONEDIT action with **ON ACTION action-name INFIELD field-name**, it was mandatory to add the "field-name." prefix in the **ACTION** attribute, to make the button always active, even when the focus was not in the field:

```plaintext
-- Form file:
BUTTONEDIT f1 = customer.cust_city, ACTION = cust_city.zoom;

-- Program file:
ON ACTION zoom INFIELD cust_city
```

When specifying only the action name in the **ACTION** attribute, the BUTTONEDIT button was only enabled when the focus was in the field. However, this is not the expected behavior of a BUTTONEDIT button.

Starting with Genero 3.10, the expected behavior is now implicitly achieved, even when the field name is not specified in the **ACTION** attribute:

```plaintext
BUTTONEDIT f1 = customer.cust_city, ACTION = zoom;
```

For more details see **Field-specific actions (INFIELD clause)** on page 1767, **BUTTONEDIT item type** on page 1267.

**BDL 3.00 upgrade guide**
These topics describe product changes you must be aware of when upgrading to version 3.00.

**Important**: This is an incremental upgrade guide that covers only topics related to the Genero BDL version specified in the page title. Check prior upgrade guides if you migrate from an earlier version. Make sure to also read about the new features for this Genero version.

Corresponding new features page: **BDL 3.00 new features** on page 61.

**Web Services changes**
There are changes in support of web services in Genero 3.00.

**Migration to 3.00 on client side**
If migrating from a version 2.xx of a GWS client application to version 3.00, you need to regenerate all client stubs in your application using the `fglwsdl` tool.

**Important**: It is mandatory to regenerate the client stubs, to support fault response with HTTP error code of 200. For more information see **SOAP fault handling in client stub** on page 152.

See also **Change client behavior at runtime**.

**Default SSL/TLS protocol on server side**
The default for the `FGLPROFILE` entry `security.global.protocol` is now SSLv23, enabling all supported SSL/TLS protocols, including TLSv1.2 as required by the Federal Law of USA. In prior versions, the default was TLSv1 (v1.0). It is up to the web server administrator to restrict the SSL/TLS protocol to TLSv1.2.

**Important**: 
Starting with FGLGWS 3.00.24 GWS secured communication is based on the OpenSSL 1.1 engine. This version of OpenSSL always selects the security protocol. It no longer allows you to specify a specific Transport Layer Security (TLS) or Secure Sockets Layer (SSL).

The security.global.protocol entry in the fglprofile file is therefore not supported.

For instance, if you have set security.global.protocol = "TLSv1.2" to configure OpenSSL to use TLSv1.2 for HTTPS for earlier versions, you may encounter the following error message in your Web service:

OpenSSL 1.1 doesn't support specific protocol anymore

It is therefore recommended to remove the security.global.protocol entry from your fglprofile file.

For more details, see HTTPS and password encryption on page 3489

Server socket read/write timeout on server side

Before version 3.00, when a WS client did not send all the HTTP body (for instance, after connection has been accepted), by default the WS server would wait indefinitely, and this could end up in a denial of service.

The com.WebServiceEngine class supports now a new option called server_readwritetimeout, to define the server socket read/write timeout: If a timeout occurs, the WS server program will raise the BDL exception -15553. By default this timeout is defined as 5 seconds.

For more details, see WebServiceEngine options on page 2840.

HTTPPart header default settings with com.HTTPPart.CreateAttachment()

The com.HTTPPart.CreateAttachment() method now by default creates header fields based on the file name and file extension.

For more details, see com.HTTPPart.CreateAttachment on page 2905.

File path returned by com.HTTPPart.getAttachment()

Before version 3.00, the com.HTTPPart.getAttachment() method returned the path to a temporary file. Starting with Version 3.00, this method will now return the absolute path location of the received part file name, based on the "Content-Disposition" header.

For more details, see com.HTTPPart.getAttachment on page 2907.

XForms characters in com.HTTPServiceRequest.readFormEncodedRequest()

Starting with version 3.00, if the result string of the HTTP request contains & or = XForms special characters, these are escaped by doubling them.

For more details, see com.HTTPServiceRequest.readFormEncodedRequest on page 2861.

Specific exception -15575 when GAS disconnects web service server

The GWS methods listed below raise an exception with a specific error code -15575, when the GAS disconnects properly from the web service server. Before version 3.00, the generic error -15565 was raised. A specific error code allows you to distinguish fully a normal disconnection from other errors, in a TRY/CATCH block. See code examples in method reference pages:

- com.WebServiceEngine.GetHTTPServiceRequest on page 2832
- com.WebServiceEngine.HandleRequest on page 2833

SOAP fault handling in client stub

Web Services client stub generation has been changed to support fault response with HTTP error code of 200.

The generated code supports SOAP fault with HTTP error code of 200 and 500. To enable this new feature in your client stub code, regenerate the stubs with the fglwsdl tool.
For more details, see Client side SOAP fault handling.

Optional multipart handling in client stub

In the generated client stub code, all functions handling the SOAP request with multipart get an additional input parameter and/or return parameter as a DYNAMIC ARRAY OF com.HTTPPart, to pass and return optional parts.

When generating client stubs managing multipart, you get an extra input and/or output variable called "AnyInputParts" and "AnyOutputParts" that is a DYNAMIC ARRAY of com.HTTPPart objects. Those variables may contain additional input and/or output HTTP parts not specified in the WSDL. You will have to adapt your client program by handling those dynamic arrays in any Genero functions calling such stubs.

Request example prior to 3.00:

```plaintext
FUNCTION xxx_g(InputHttpPart_1, ..., InputHttpPart_n)
  DEFINE InputHttpPart_1 com.HTTPPart
  ...
  DEFINE InputHttpPart_n com.HTTPPart
  ...
  RETURN wsstatus
END FUNCTION
```

Request example 3.00 and greater, with extra input variable AnyInputParts:

```plaintext
FUNCTION xxx_g(InputHttpPart_1, ..., InputHttpPart_n)
  DEFINE InputHttpPart_1 com.HTTPPart
  ...
  DEFINE InputHttpPart_n com.HTTPPart
  DEFINE AnyInputParts DYNAMIC ARRAY OF com.HTTPPart
  ...
  RETURN wsstatus
END FUNCTION
```

Note: This change has also been backported in 2.50.25.

For more details, see Multipart in the client stub on page 3332.

Removal of FGLWSNOINFO environment variable

Before version 3.00, the GWS library displayed by default a message about certificates used by the program:

```plaintext
—
WS-INFO (Certificate authority) | Loading from Windows keystore
—
```

To avoid this message, it was possible to set the FGLWSNOINFO environment variable to TRUE.

Starting with version 3.00, this message is no longer displayed by the GWS library, and the FGLWSNOINFO is no longer required.

Desupported Web Services APIs

The methods listed in the following table are desupported in Genero 3.00.

**Table 103: Table of desupported methods (with their alternative)**

<table>
<thead>
<tr>
<th>Methods desupported as of 3.00</th>
<th>Alternative method to use</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.Util.CreateDigestString</td>
<td>security.Digest.CreateDigestString on page 3146</td>
</tr>
</tbody>
</table>
Form definitions for mobile applications
Genero version 3 supports grid-based layout with all front-ends, and introduces STACK layout.

Support for grid-based and stack-based layout
Before Genero version 3.00 (i.e., with Genero Mobile version 1.1), the GMI front-end could only support a stack-based layout and it was required to create different forms for iOS apps and other front-ends supporting grid-based layout. In fact, to get a stack-based layout, grid-based .pem forms were automatically transformed on the fly when displayed on the GMI front-end.

Starting with Genero 3.00, all mobile front-ends support now grid-based layout and stack-based layout, and a new STACK layout container was introduced to define stack-based layout forms explicitly. Therefore, you can now use the same form definition for all mobile front-ends, by implementing the layout type of your choice. It is even possible to mix grid-based or stack-based forms in the same app.

Loading different forms based on front-end type
If you want to use a grid-based or stack-based form for the front-end, you can load the form with OPEN FORM (or OPEN WINDOW) and implement the layout type based on the front-end name returned by the ui.Interface.getFrontEndName() method:

```
MAIN
  ...
  OPEN FORM f1 FROM IIF( ui.Interface.getFrontEndName()=="GMI",
    "myform_stack", "myform_grid")
  DISPLAY FORM f1
  ...
END MAIN
```

Related concepts
Form rendering on page 1415
The section explains the layout rules to render forms on graphical front-ends.

Database drivers changes
Desupported database drivers.

Databases / ODI drivers desupported in version 3.00:

- SAP ASE 15.x (dbase_15). New SAP® ASE 16.x version is now supported.
- Oracle® Database 10.1 and 10.2 (dbmora_10)

  **Note:** This driver is no longer available for Linux® PowerPC 32/64 platforms, because Oracle® has desupported this platform.
- IBM DB2® UDB 9.x (dbmdb2_9)

Related concepts
Database driver specification (driver) on page 662

Oracle DB NUMBER type
The NUMBER/FLOAT Oracle® data type can now be extracted by fgldbsch to create .sch files.

Before Genero 3.00, columns using the native Oracle® NUMBER/NUMBER(p>32) type (with up to 38 significant digits), or the FLOAT(b) type (when (b/3)>32), were not allowed by the fgldbsch schema extractor. This

<table>
<thead>
<tr>
<th>Methods desupported as of 3.00</th>
<th>Alternative method to use</th>
</tr>
</thead>
</table>
restriction was applied to avoid the risk of overflow errors, if the Oracle® NUMBER/FLOAT column contains values that do not fit into a BDL DECIMAL(32, s) type.

Starting with Genero 3.00, fgldbsch can map NUMBER/FLOAT native Oracle® types to BDL DECIMAL(32) or DECIMAL(32, s) types, when specifying the flag B with the –cv option for these native types:

- NUMBER (floating point number) is extracted as DECIMAL(32)
- NUMBER(p>32) (scale defaults to 0) is extracted as DECIMAL(32.0)
- NUMBER(p>32, s) or NUMBER(*, s) is extracted as DECIMAL(32, s)
- FLOAT(b) is extracted as DECIMAL(b/3) or FLOAT

For more details about Oracle® type conversion rules and –cv type positions, run fgldbsch with the –cx ora option.

**Note:** This new behavior has been introduced to simplify integration with existing Oracle® databases, to extract .sch schema from databases using column types that have no exact equivalent BDL type. When designing new database tables, it is recommended that you only use DECIMAL(p, s), with p≤32 to achieve maximum portability. When fetching numeric values with more than 32 significant digits into BDL decimals, values will be rounded for DECIMAL(32), or raise an overflow error -1226 for DECIMAL(32, s).

**Related concepts**

- **Numeric data types** on page 924
- **Database schema** on page 467

Defines database table structures with column type information to be reused in program variable definitions.

**Oracle DB scroll cursor emulation removal**

The scroll cursor emulation has been removed in the Oracle® DB driver.

Before Genero 3.00, it was possible to enable scrollable cursor emulation (with temporary files) by defining the following FGLPROFILE entry:

```
  dbi.database.mydbname.ora.cursor.scroll.emul = true
```

This feature was supported to workaround an Oracle® DB bug in versions 8 and 9i. The bug no longer exist in recent Oracle® DB versions and therefore the default native scrollable cursor feature can be safely used.

If this FGLPROFILE entry is set, the runtime system will print a warning to stderr.

**Related concepts**

- **FGLPROFILE entries for core language** on page 256

This is a summary of FGLPROFILE entries supported by the core BDL.

- **Scrollable cursors** on page 623

How scrollable cursors can be supported on different databases.

**MySQL VARCHAR size limit**

MySQL 5 VARCHAR columns can be used to store VARCHAR(N>255) values.

Before Genero 3.00, the Oracle® MySQL driver converted a VARCHAR(N>255) type to a MySQL TEXT type, because MySQL versions before 5.0.3 only allowed up to 255 characters for a VARCHAR column. MySQL TEXT type is a large object type with specific semantics and constraints, but it was the only available type to store character data above the 255 character limit. As a result, data type information was lost when extracting the database schema with fgldbsch from a MySQL database: When creating a table in a Genero BDL program, the original VARCHAR(N>255) type was converted to TEXT (with a fixed size of 65535 characters), and then converted by fgldbsch back to a VARCHAR2(65535) type in the .sch file. The original size of the VARCHAR type was lost.

Starting with Genero 3.00, when creating a table in a BDL program with CREATE TABLE, the MySQL driver leaves any VARCHAR(N) as is, even if the size is greater than 255.

**Note:** The MySQL driver does not distinguish MySQL server 5.0.x (5.0.2 / 5.0.3) versions. It assumes that we are connected to a server version 5.0.3 or above, supporting large VARCHAR types.
If your application is using VARCHAR(N) types with N>255 and your MySQL server version is 5.0.3 or above, it is recommended that you review your database creation scripts to use VARCHAR(N) instead of TEXT.

**Note:** The CHAR(N>255) types are still mapped to a MySQL TEXT type, because MySQL CHAR type has a limit of 255 characters. When designing a database, consider using CHAR only for short character string data storage (less than 50 characters), and use VARCHAR for larger character string data storage (name, address, comments).

**Related concepts**

- [CHAR and VARCHAR data types](#) on page 883
- [Primitive Data types](#) on page 289

Selecting the correct data type assists you in the input, storage, and display of your data.

- [TEXT and BYTE (LOB) types](#) on page 893
- [Using portable data types](#) on page 614

Only a limited set of data types are really portable across several database engines.

**MySQL DATETIME fractional seconds**

MySQL 5.6.4 TIME and DATETIME types support fractions of seconds that can be used to store DATETIME HOUR TO FRACTION(N) or DATETIME YEAR TO FRACTION(N).

Before Genero 3.00, the Oracle® MySQL driver converted DATETIME types as follows:

- DATETIME HOUR TO SECOND was converted to MySQL TIME.
- Other DATETIME types were converted to MySQL DATETIME.

Starting with Genero 3.00, when creating a table in a BDL program with the CREATE TABLE statement, if the MySQL server version is greater or equal to 5.6.4, the types are converted differently, as follows:

- DATETIME HOUR TO MINUTE is converted to MySQL TIME (seconds set to 00).
- DATETIME HOUR TO SECOND is converted to MySQL TIME.
- DATETIME HOUR TO FRACTION(n) is converted to MySQL TIME(n).
- DATETIME YEAR TO MINUTE is converted to MySQL DATETIME (seconds set to 00).
- DATETIME YEAR TO SECOND is converted to MySQL DATETIME.
- DATETIME YEAR TO FRACTION(n) is converted to MySQL DATETIME(n).

This change has no impact your application when using DATETIME HOUR TO SECOND or DATETIME YEAR TO SECOND.

However, it is now possible to store DATETIME HOUR TO FRACTION(n) and DATETIME YEAR TO FRACTION(n) data. The DATETIME YEAR TO FRACTION(n) is typically used to implement data modification timestamps to track user changes.

**Important:** More changes on DATETIME type mapping take place in Genero BDL version 3.20.

**Related concepts**

- [DATE and DATETIME data types](#) on page 888
- [Using portable data types](#) on page 614

Only a limited set of data types are really portable across several database engines.

- [Primitive Data types](#) on page 289

Selecting the correct data type assists you in the input, storage, and display of your data.

- [Date/time literals in SQL statements](#) on page 635

Good practices for date and time handling in SQL.

**PostgreSQL DATETIME type mapping change**

Conversion of DATETIME type with fractional seconds to PostgreSQL TIME(N)/TIMESTAMP(N) was invalid and has been reviewed.

Before Genero 3.00, the PostgreSQL driver converted DATETIME types as follows:

- DATETIME HOUR TO MINUTE was converted to TIMESTAMP(3) WITHOUT TIME ZONE
• DATETIME HOUR TO SECOND was converted to TIME(0) WITHOUT TIME ZONE
• DATETIME HOUR TO FRACTION(n) was converted to TIME(n+1) WITHOUT TIME ZONE
• DATETIME YEAR TO MINUTE was converted to TIMESTAMP(3) WITHOUT TIME ZONE
• DATETIME YEAR TO SECOND was converted to TIMESTAMP(3) WITHOUT TIME ZONE
• DATETIME YEAR TO FRACTION(n) was converted to TIMESTAMP(n+1) WITHOUT TIME ZONE

Starting with Genero 3.00, when creating a table in a BDL program with CREATE TABLE, the types are converted in a different way:
• DATETIME HOUR TO MINUTE is converted to TIME(0) WITHOUT TIME ZONE (seconds set to 00).
• DATETIME HOUR TO SECOND is converted to TIME(0) WITHOUT TIME ZONE.
• DATETIME HOUR TO FRACTION(n) is converted to TIME(n) WITHOUT TIME ZONE.
• DATETIME YEAR TO MINUTE is converted to TIMESTAMP(0) WITHOUT TIME ZONE (seconds set to 00).
• DATETIME YEAR TO SECOND is converted to TIMESTAMP(0) WITHOUT TIME ZONE.
• DATETIME YEAR TO FRACTION(n) is converted to TIMESTAMP(n) WITHOUT TIME ZONE.

This bug fix introduces an incompatibility and can have an impact on applications using DATETIME HOUR TO MINUTE, DATETIME HOUR TO FRACTION(n) or DATETIME YEAR TO FRACTION(n). If you are using one of these types, consider reviewing your database schema, to modify the column types according to the new SQL type conversion rules.

Important: More changes on DATETIME type mapping take place in Genero BDL version 3.20.

Related concepts
DATE and DATETIME data types on page 977
Using portable data types on page 614
Only a limited set of data types are really portable across several database engines.
Primitive Data types on page 289
Selecting the correct data type assists you in the input, storage, and display of your data.

Date/time literals in SQL statements on page 635
Good practices for date and time handling in SQL.

MariaDB support
The MariaDB database is now supported by Genero 3.00.

MariaDB is the open source brand of Oracle’s MySQL that has been adopted by several major organizations.
The purpose of the MariaDB project is to be a drop-in replacement for MySQL.

MariaDB supported versions are 10.0 and higher.

To connect to MariaDB, use the MySQL database driver (dbmmys), and follow MySQL adaptation guide for configuration and SQL portability issues.
Depending on the libmysqlclient library compatibility, you might need to configure Genero to use a version-stamped driver. As of Genero version 3.00, the generic driver name “dbmmys” can be used to connect to MariaDB 10.0. See Database driver specification (driver) on page 662 for more details.

Related concepts
Oracle MySQL / MariaDB on page 876

The FreeTDS driver can now be used for SQL Server versions > 2005.

Before Genero version 3.00, the FreeTDS driver could only be used to connect to SQL Server 2005. Starting with Genero 3.00 the dbmftm driver can connect to SQL Server 2008, 2012 and 2014.

With SQL Server version >= 2008, date/time types used to store DATE and DATETIME values are different to those used with SQL Server version 2005. See DATE and DATETIME data types on page 844 for more details.
**Important:** For SQL Server version 2008, 2012 and 2014, you must set TDS_Version=7.3 in odbc.ini. Using TDS version 8.0 introduces problems (tested with FreeTDS 0.95.5 to 0.95.19)

**Related tasks**
Prepare the runtime environment - connecting to the database on page 828

**FGL_GETVERSION() built-in function**
The FGL_GETVERSION() function now returns the product version number (for example: 3.00.00).

Prior to Genero 3.00, the fgl_getversion() built-in function returned the internal build number.

Starting with Genero 3.00, the function returns the product version number as a string, such as 3.00.00.

**Related concepts**
Built-in functions on page 2268
A *built-in function* is a predefined function that is part of the runtime system, or provided as a library function automatically loaded when a program starts. The built-in functions are part of the language.

**Built-in front-end icons desupport**
Image resources included in front-ends are desupported with Genero 3.00.

Starting with Genero 3.00, the icon files distributed in front-end packages are no longer provided (as before in the GDC-installation-dir/pics for example)

Common icons for buttons, toolbars, topmenus, and other items using icons can be centralized on the application side where the program executes. It is recommended that this feature be used to provide the same icons on different types of front-ends, or use specific icons, but from the same central icon directory. For more details, see Providing the image resource on page 1149.

Note that mobile front-ends will display default icons, for default action views, if no IMAGE attribute is specified for the action. See Action views on mobile devices on page 1777 for more details.

**Related concepts**
FGLIMAGEPATH on page 276
Defines the search paths for VM server image files.

**Presentation styles changes**
Deprecated and renamed presentation style attributes.

Starting with version 3.00:

The following presentation style attributes are desupported:

- **CheckBox:** customWidget
  
  **Important:** The Checkbox.customWidget style attribute has been reintroduced in version 3.10 for GBC with the "toggleButton" value.

The following presentation style attributes are deprecated:

- **Image:** imageContainerType ( = "browser")
  
  Replace IMAGE fields using this style attribute with URL-based WEBCOMPONENT fields.
- **ComboBox:** completorCompleter
  
  Consider using the COMPLETER form field attribute instead.

**Related concepts**
Style attributes reference on page 1182
A presentation style attribute may be a common attribute that can be applied to any graphical element. Most presentation style attributes apply only to a specific graphical element.

**Modifications in front calls**
Describes changes applied to front calls.

Front call modifications in BDL version 3.00:
- Before version 3.00, the `connectivity` front call accepted a host name as parameter. Starting with version 3.00, this front call no longer uses a host name. It will only check the available network type. For more details, see the `mobile.connectivity` front call.

**Related concepts**
- [Front calls on page 583](#)
- [SERIAL emulation with SQL Server](#)

**SERIAL emulation with SQL Server**
The SERIAL and BIGSERIAL types can be emulated with triggers and sequences when using SQL Server 2012 and higher.

By default when using SQL Server, the SERIAL and BIGSERIAL types are emulated with IDENTITY columns. This native sequence generator is the fastest and preferred solution. However, it requires removing the serial column in all INSERT statements, which can lead to a large change in your legacy code.

Until version 3.00, it was possible to workaround this limitation by using the "regtable" serial emulation. But this solution required using a dedicated SERIALREG table that needed to be updated for each INSERT statement. This resulted in poor performances, when concurrent programs create rows in the same tables (locking issues in SERIALREG).

Starting with Genero 3.00, it is now possible to use a serial emulation based on triggers and sequences. Sequences were introduced in SQL Server version 2012, so you need at least a 2012 server in order to use this emulation:

```
 dbi.database.mydb.ifxemul.datatype.serial.emulation = "trigseq"
```

**Related concepts**
- [SERIAL and BIGSERIAL data types on page 847](#)
- [FGLPROFILE entries for core language on page 256](#)
- [Auto-incremented columns (serials) on page 626](#)

**Improved compilation time**
The `fglcomp` and `fglform` compilers have been reviewed to achieve faster compilation.

A Genero project can be very large, with thousands of .4gl source files to compile. Compilation time can be an issue when the whole set of sources needs to be compiled every day, or several times a day.

In Genero 3.00, the `fglcomp` on page 2071 compiler has been improved to deliver better performances. Depending on the content of the source file, the compiler can complete the process twice as fast.

Loading .sch database schema files has also been improved. Using huge schema files with several thousands lines is no longer an issue. This is especially useful when compiling forms that define fields based on database columns in a schema file.

**Related concepts**
- [fglform on page 2068](#)
- [Database schema on page 467](#)
Defines database table structures with column type information to be reused in program variable definitions.

**Preprocessor changes**
Several bugs have been fixed in the preprocessor, that can now result in a compilation error.

**String token expansion**
Before version 3.00, the following preprocessor syntax was used to expand a string macro parameter:

```c
#define T(x) DISPLAY "head_"#x"_tail"
-- macro usage:
T(body)
```

This produced the following result (after preprocessing):

```c
"head_""body""_tail"
```

It was accepted by the compiler, because it was interpreted as a single string literal.
The new preprocessor now produces (as expected):

```c
"head_ " "body" "_tail"
```

However, this will now result in a compiler error, because this is not a valid string literal.
To solve an issue such as this and get the same result string as before version 3.00, use the `||` concatenation operator in the preprocessor macro and add (escaped) double quotes before and after the `#ident` placeholder:

```c
#define T(x) DISPLAY "head_" || #x || "_tail"
```
or, by using single quotes as border strings delimiters:

```c
#define T(x) DISPLAY 'head_' || #x || '_tail'
```

**Identifier concatenation**
Before version 3.00, the following type of macro:

```c
#define FOO() foo
-- macro usage:
FOO()bar
```

was producing a single identifier token (accepted by the compiler):

```c
foobar
```

But it will now produce two distinct identifier tokens (as expected):

```c
foo bar
```

And this will result in a compilation error.

**Backslash in macro parameters**
Before version 3.00.00 it was possible to use the backslash to escape a comma in preprocessor macro parameters.
This syntax is no longer allowed by the preprocessor, it is not a valid usage. To solve this issue, replace parameters by real string literals in the macro:

```c
-- bad coding
```
The ## paste operator

Before version 3.00.00, the ## paste operator was used to construct code with two elements that did not result in a valid token, for example:

```plaintext
&define FOO(name) rec_ ## [ x ]
FOO(x)
```

produced:

```plaintext
rec_[ x ]
```

This kind of preprocessor macro is no longer allowed in version 3.00.00 and will result in a compiler error:

```plaintext
x.4gl:2:1:2:1:error:(-8042) The operator '##' formed 'rec_[', an invalid preprocessing token.
```

The ## paste operator must be used to join two identifiers, to create a new identifier:

```plaintext
&define REC_PREFIX(name) rec_ ## name
LET REC_PREFIX(customer) = NULL
```

This will produce:

```plaintext
LET rec_customer = NULL
```

File inclusion search path

Before version 3.00, the compiler option -I was allowing you to specify a list of paths separated by the OS path separator.

This behavior was unexpected and wrong: The -I option must define a single path (or directory).

Command lines using the followin (UNIX) notation:

```plaintext
fglcomp -I path1:path2 ...
```

must be reviewed by specifying each directory in a separate -I option:

```plaintext
fglcomp -I path1 -I path2 ...
```

Related concepts

Source preprocessor on page 2132

A typical preprocessor like in the C language.

Current system time in UTC

Use the util.Datetime.getCurrentAsUTC() method to get the current system date/time in UTC.

Starting with Genero version 3.00, you can use the util.Datetime.getCurrentAsUTC() method to get the current system time in UTC (Coordinated Universal Time).
This method has been added to solve the issue when using `util.Datetime.toUTC(CURRENT)` during the daylight saving time transition period in the fall, as described in `util.Datetime.toUTC` on page 2733.

**Structured ARRAYs in list dialogs**

ARRAYs with sub-records can be used in list dialogs, to simplify array definition based on database tables, requiring additional information at runtime.

Starting with Genero version 3.00, ARRAY variables defined with sub-records can be bound to DISPLAY ARRAY and INPUT ARRAY screen records.

This is especially useful when you need to define arrays from database tables, and handle additional row information at runtime, for example, to hold an image resource for each row, to be displayed with the IMAGECOLUMN attribute.

An array is usually defined with a flat list of members:

```plaintext
SCHEMA shop
DEFINE a_items DYNAMIC ARRAY OF RECORD LIKE items.*
...
```

With version 3.00, arrays structured with sub-records can now be used within a DISPLAY ARRAY or INPUT ARRAY dialog. The array members and the form fields used by the screen array are bound by position:

```plaintext
SCHEMA shop
DEFINE a_items DYNAMIC ARRAY OF RECORD
    item_data RECORD LIKE items.*,
    it_image STRING,
    it_count INTEGER
END RECORD
...
DISPLAY ARRAY a_items TO sr.*
...
```

For more details about program variable to form field binding in dialogs, see Binding variables to form fields on page 1715, Example 4: DISPLAY ARRAY with structured array on page 1530.

**BDL 2.51 upgrade guide**

These topics describe product changes you must be aware of when upgrading to version 2.51.

**Important:** This is an incremental upgrade guide that covers only topics related to the Genero BDL version specified in the page title. Check prior upgrade guides if you migrate from an earlier version. Make sure to also read about the new features for this Genero version.

Corresponding new features page: BDL 2.51 new features on page 66.

**Database drivers changes**

Desupported database drivers.

**Databases / ODI drivers desupported in version 2.51:**

- Genero DB is no longer supported (`dbmads*`).
- Oracle® MySQL 4.1 and 5.0 (`dbmmy41x`, `dbmmy50x`)
- PostgreSQL 8.3, 8.4 (`dbmpgs83x`, `dbmpgs84x`)
- Oracle® Database 9.2 (`dbmora92x`)
- IBM® DB2® UDB version 7.x and 8.x (`dbmdb27x`, `dbmdb28x`)

Note also that database driver naming convention has changed in 2.51, for more details see New database driver name specification on page 163.

**Related concepts**

Microsoft SQL Server on page 827
Database driver specification (driver) on page 662

New database driver name specification

Allows database driver specification without target database version information.

Starting with version 2.51, the database drivers follow a new file name convention, which allows you to specify a generic name based on the target database type without any database version information.

Important: Most database driver names have changed. You need to re-configure the "driver" entry in your FGLPROFILE settings (or database connection string parameters), to match the new driver names. If you are using the default driver (dbmdefault), there is no configuration change needed. To simplify upgrading, the runtime system identifies old driver names and converts them to new names. However, it is recommended that you consider using the generic driver name corresponding to the type of database your applications connect to. The error -6366 occurs if the runtime system is not able to load the specified database driver, or cannot identify an old driver name.

Before version 2.51, it was required to specify the exact database type and version, to match both the database client and the server version. For example, when using Oracle® 11.2 (server and client):

```plaintext
dbi.database.stores.driver = "dbmoraB2x"
```

Starting with 2.51, you can now, for example, specify a generic driver name without version, which can connect to any database server version supported by the DB vendor client/server protocol. The generic name defines a database driver for the latest database client version that is available on the platform:

```plaintext
dbi.database.stores.driver = "dbmora"
```

Each generic database driver name has also a human-readable alias, such as "informix" or "oracle".

```plaintext
dbi.database.stores.driver = "oracle"
```

To simplify driver specification, install the latest database client software that corresponds to the generic driver name, especially if it does not require a database server upgrade.

For some database client software, additional database drivers are still provided for older database client versions (if available on the platform). In such cases, the driver file name gets a version identifier.

For example:

- dbmora_11 (Oracle® 11g client)
- dbmmsys_5_1 (Oracle® MySQL client 5.1.x)
- dbmsnc_10 (SQL Server Native Client 10 (SQLNCLI10.DLL))
- dbmsnc_9 (SQL Server Native Client 9 (SQLNCLI.DLL))

Such database drivers with version info are provided to follow db client library dependency rules, as defined by the database vendors. For example, on a Linux® platform, Oracle® MySQL version 5.1.x provides the db client library named libmysqlclient.so.16. In this file name, "16" is the version number that defines the shared library compatibility. The database driver that was compiled and linked in a compatible db client environment is dbmmsys_5_1. This database driver is linked to libmysqlclient.so.16. Starting with Oracle® MySQL version 5.5.x, the db client library version number has been incremented to 18 (linked to libmysqlclient.so.18). The driver to be used with that library version is dbmmsys_5_5, which was compiled and linked with a 5.5.x environment.

Related concepts

Database driver specification (driver) on page 662
FGLPROFILE entries for core language on page 256
This is a summary of FGLPROFILE entries supported by the core BDL.

**The FIELD form item type and .val schema file**

Form files using the FIELD item type and/or .val attribute definitions must be reviewed.

Starting with version 2.51, the FIELD item type defining abstract fields in forms, based on .val schema file attributes is deprecated.

Furthermore, any non-I4GL attribute defined in the .val schema file must be avoided. Reading attributes in the .val is now only supported for compatibility with I4GL projects.

With Genero, it is recommended to define all form item attributes in the form definition file.

**Related concepts**

Form specification files on page 1237

Form specification files are the source files defining the layout and content of application forms.

**TRY/CATCH and ERROR LOG**

Errors are no longer logged when raised in a TRY/CATCH block.

Before version 2.51, exceptions occurring in a TRY/CATCH block were logged if the error log is initiated with the startlog() function. With version 2.51, if an exception is raised in a TRY/CATCH block, it will no longer be logged in the error log file. In other words, the TRY/CATCH block will behave like WHENEVER ERROR CONTINUE, regarding error logging.

Example:

```sql
CALL startlog("errors.txt")
...
TRY
  INSERT INTO customer ...  
CATCH
  -- Handle errors and write to error log with errorlog() if needed.
  IF SQLCA.SQLCODE == -8634 THEN
    ...
  END IF
END TRY
```

**Important:** In order to get this new behavior, the pcode is no longer compatible with older versions (<=2.50). All programs must be recompiled.

**Related concepts**

Exceptions on page 502

Describes exception (error) handling in the programs.

**DATETIME types with SQLite**

Better support for Informix® DATETIME types emulation within SQLite.

Before version 2.51, DATETIME SQL types where converted to SQLite types as follows:

- DATETIME HOUR TO SECOND type was translated to TIME (hh:mm:ss).
- DATETIME YEAR TO FRACTION and all other combinations (except HOUR TO SECOND) were translated to TIMESTAMP (YYYY-MM-DD hh:mm:ss.fff).

Since most DATETIME types were converted to TIMESTAMP, it was not possible to distinguish common date/time types such as DATETIME HOUR TO MINUTE or DATETIME YEAR TO MINUTE, especially when extracting the database schema with fgldbsch. Type information was lost and this prevented schema-base variable definitions with DEFINE LIKE.

Starting with version 2.51, common DATETIME SQL types are now mapped to different types in SQLite, to provide better support for these types. In fact, SQLite allows you to define table columns with custom types (you can use any type name). However, the number of tokens in the syntax is limited so it's not possible to use, for example, the tokens DATETIME YEAR TO SECOND directly. The Genero database driver uses this SQLite SQL language feature to
map Informix-style DATETIME types to specific custom types. For example, a DATETIME HOUR TO MINUTE becomes a SMALLTIME, a DATETIME YEAR TO FRACTION(2) becomes a DATETIME(2), etc. Furthermore, the data values inserted in the database now match exactly the precision of the original DATETIME type. For more details about date/time mapping and emulation, see DATE and DATETIME data types on page 1013.

**Related concepts**
- SQLite on page 1004
- fgl dbsch on page 2079

The fgl dbsch tool generates the database schema files from an existing database.

**Desupport of C-Extension API functions**

BIGINT and BOOLEAN stack functions and C API functions for C-Extensions are no longer supported.

**Note:** Starting with version 3.10 (also backported in 3.00.10), the popbigint() and pushbigint() function are again available.

Since version 2.51:

The C-Extension stack functions to handle BIGINT and BOOLEAN types have been removed:

**Table 104: Desupported FGL stack functions**

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>popboolean()</td>
</tr>
<tr>
<td>popbigint()</td>
</tr>
<tr>
<td>pushboolean()</td>
</tr>
<tr>
<td>pushbigint()</td>
</tr>
</tbody>
</table>

The C API functions such as decadd(), risnull(), rsetnull(), have been removed. These functions are part of the IBM® Informix® ESQL/C product and cannot be part of the Genero BDL product. The Genero runtime system provides only the C functions to push and pop data on the Genero BDL stack.

Below is the list of C API functions that have been removed, check your C extension code for the usage of these functions. If such functions are required, link your C-Extensions with the IBM® Informix® ESQL/C libraries.

**Table 105: Desupported C API functions**

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>bycmpr()</td>
</tr>
<tr>
<td>byleng()</td>
</tr>
<tr>
<td>bycopy()</td>
</tr>
<tr>
<td>byfill()</td>
</tr>
<tr>
<td>risnull()</td>
</tr>
<tr>
<td>rsetnull()</td>
</tr>
<tr>
<td>rgetmsg()</td>
</tr>
<tr>
<td>rgetlmsg()</td>
</tr>
<tr>
<td>rtypalign()</td>
</tr>
<tr>
<td>rtypmsize()</td>
</tr>
<tr>
<td>rtypname()</td>
</tr>
<tr>
<td>rtypwidth()</td>
</tr>
<tr>
<td>rdatestr()</td>
</tr>
<tr>
<td>rdayofweek()</td>
</tr>
</tbody>
</table>
rdefmtdate()
ifx_defmtdate()
rfmtdate()
rjulmdy()
rlleapyear()
rmdyjul()
rstrdate()
ifx_strdate()
rtoday()
ldchar()
rdownshift()
rfmtdouble()
rfmttint4()
rstod()
rstoi()
rstol()
rupshift()
scat()
stchar()
stcmp()
stcopy()
stleng()
deadd()
deccmp()
deccopy()
deccvasec()
deccvdbl()
deccvflt()
deccvint()
deccvlong()
declong()
dececmul()
decelecv()
decdecvt()
decfcvt()
decfcvt()
decmul()
decround()
decsub()
dectoasc()
Related concepts

C-Extensions on page 2230

With C-Extensions, you can bind your own C libraries in the runtime system, to call C function from the application code.

BDL 2.50 upgrade guide

These topics describe product changes you must be aware of when upgrading to version 2.50.

Important: This is an incremental upgrade guide that covers only topics related to the Genero BDL version specified in the page title. Check prior upgrade guides if you migrate from an earlier version. Make sure to also read about the new features for this Genero version.

Corresponding new features page: BDL 2.50 new features on page 69.
Web Services changes
There are changes in support of web services in Genero 2.50.

Client stubs managing multipart changes on client side

Important: This change has been backported from version 3.00

Starting with version 2.50.25, when generating client stubs managing multipart, you get an extra input and/or output variable called "AnyInputParts" and "AnyOutputParts" that is a DYNAMIC ARRAY of com.HTTPPart objects. Those variables may contain additional input and/or output HTTP parts not specified in the WSDL. You will have to adapt your client program by handling those dynamic arrays in any Genero functions calling such stubs.

Request example prior to 2.50.25:

```FUNCTION xxx_g(InputHttpPart_1, ..., InputHttpPart_n)
  DEFINE InputHttpPart_1 com.HTTPPart
  ...
  DEFINE InputHttpPart_n com.HTTPPart
  ...
  RETURN wsstatus
END FUNCTION```

Request example 2.50.25 and greater, with extra input variable AnyInputParts:

```FUNCTION xxx_g(InputHttpPart_1, ..., InputHttpPart_n)
  DEFINE InputHttpPart_1 com.HTTPPart
  ...
  DEFINE InputHttpPart_n com.HTTPPart
  DEFINE AnyInputParts DYNAMIC ARRAY OF com.HTTPPart
  ...
  RETURN wsstatus
END FUNCTION```

Web Services deprecated APIs
The methods listed in the following table are deprecated in version 2.50.

Table 106: Table of deprecated methods (with their alternative)

<table>
<thead>
<tr>
<th>Method deprecated as of 2.50</th>
<th>Alternative method to use</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.Util.CreateDigestString</td>
<td><code>security.Digest.CreateDigestString</code> on page 3146</td>
</tr>
</tbody>
</table>

Related concepts

The [HTTPPart class](#) on page 2902
The `com.HTTPPart` class provides an interface to manage the HTTP attachment sent or received in HTTP.

Multipart in the client stub on page 3332
You can generate a client stub for a Web service that has multiple parts.

Dynamic arrays on page 422
Database drivers changes
Desupported database drivers.

Databases / ODI drivers desupported in versions 2.50:

- SQL Server MDAC drivers (Code MSV, name: dbmmsv*):
  
  On a Microsoft™ Windows® platform, use the SQL Server Native Client driver instead (Code SNC).
  
  With the SNC drivers, set the dbi.database.dbname.snc.widechar FGLPROFILE entry to false when using CHAR/VARCHAR/TEXT in the SQL Server database. Note that this is only required for versions up to 3.00: Starting with version 3.10, the char mode of SNC driver is selected automatically, depending on the current application locale.

- Oracle® MySQL 5.4 (dbmmys54x)
- Oracle® Database 8.1 (dbmora81x)
- Oracle® Database 9.0 (dbmora90x)

Related concepts
Microsoft SQL Server on page 827
Oracle MySQL / MariaDB on page 876
Oracle Database on page 908

TEXT/BYTE support with FTM/ESM database drivers
FTM and ESM database drivers TEXT/BYTE type mapping has changed.

Since version 2.50, the TEXT and BYTE data types are now converted respectively to VARCHAR(MAX) and VARBINARY(MAX) data types, the recommended LOB types introduced in SQL Server 2005. Before version 2.50, the TEXT and BYTE data types were converted to TEXT and IMAGE data types, respectively, in SQL Server.

It is still possible to use SQL Server TEXT and IMAGE types, but if you create or alter tables in an FGL program, the VARCHAR(MAX) and VARBINARY(MAX) types will be used instead.

Related concepts
TEXT and BYTE (LOB) types on page 854

Presentation styles changes
Deprecated and renamed presentation style attributes.

Starting with version 2.50:

The following presentation style attributes are deprecated (still implemented, but not to be used):

- Window: backgroundImage
- TextEdit: textSyntaxHighlight

The next presentation style attributes have been replaced by a new style attribute, or have been renamed:

- CheckBox: nativeLook => customWidget (with same possible values)

  Important: In version 3.00, the Checkbox.customWidget style attribute has been desupported, but reintroduced in version 3.10 for GBC with the "toggleButton" value.

Related concepts
Style attributes reference on page 1182
A presentation style attribute may be a common attribute that can be applied to any graphical element. Most presentation style attributes apply only to a specific graphical element.

**Floating point to string conversion**

The default formatting of a `DECIMAL(P)`, `SMALLFLOAT` and `FLOAT` adapts to the significant digits of the value.

Floating point decimal types (like `DECIMAL(5)`) can store a large range of values, with a variable number of digits after the decimal point: For example, a `DECIMAL(5)` can store 12345 as well as 0.12345. See `DECIMAL(p,s)` on page 298 for more details about floating point decimal types.

With Genero 2.50, the conversion to string from a `DECIMAL(P)`, `FLOAT` and `SMALLFLOAT` has been revised, to keep all significant digits and avoid data loss.

Before Genero 2.50, floating point decimals converted to strings were formatted with 2 decimal digits by default, which could lead to data loss. See following example using a `DECIMAL(12)`:

```plaintext
MAIN
    DEFINE str STRING, dec12, dec12_bis DECIMAL(12)
    LET dec12 = 10.12999
    LET str = dec12
    DISPLAY str
    LET dec12_bis = str
    DISPLAY (dec12 == dec12_bis)
END MAIN
```

Prior to Genero 2.50, the above code displayed:

```
10.13
0
```

Starting with Genero 2.50, all significant digits are kept, which allows for proper decimal data serialization:

```
10.12999
1
```

Prior to Genero 2.50, floating point decimal values conversion of huge values could also lose digits in the whole part of the number; the width of the result was never longer than p + 2. Starting with Genero 2.50, all significant digits of a floating point decimal are kept in the result string:

<table>
<thead>
<tr>
<th>Values</th>
<th>Vers&lt;2.50</th>
<th>Vers&gt;=2.50</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.23456e123</td>
<td>1.23456e123</td>
<td>1.23456e123</td>
</tr>
<tr>
<td>1.23456e40</td>
<td>1.235e40</td>
<td>1.23456e40</td>
</tr>
<tr>
<td>123.456</td>
<td>123.46</td>
<td>123.456</td>
</tr>
<tr>
<td>123456.0</td>
<td>123456.0</td>
<td>123456.0</td>
</tr>
<tr>
<td>0.123456</td>
<td>0.12</td>
<td>0.123456</td>
</tr>
<tr>
<td>0.0123456</td>
<td>0.01</td>
<td>0.0123456</td>
</tr>
<tr>
<td>0.000123456</td>
<td>0.00</td>
<td>0.000123456</td>
</tr>
<tr>
<td>1.23456e-08</td>
<td>0.00</td>
<td>1.23456e-08</td>
</tr>
</tbody>
</table>

If you expect that any `DECIMAL(P)` to string conversion rounds to 2 digits, define the following FGLPROFILE entry (this applies to all contexts):

```
fgltrun.decToCharScale2 = true
```

Starting with Genero version 3.10.20, another FGLPROFILE entry has been added, to get the 2-digit rounding of `DECIMAL(P)` only in the context of the `PRINT` statement in reports. (the `fgltrun.decToCharScale2` and `fgltrun.decToCharScale2.print` configuration parameters are exclusive):

```
fgltrun.decToCharScale2.print = true
```
Note: Do not use the fglrun.decToCharScale2* configuration parameters, unless you have migration issues. These configuration parameters apply only to DECIMAL(P) types: FLOAT and SMALLFLOAT conversions to string is not impacted.

Related concepts
Data type conversion reference on page 311
This topic lists type conversion rules for all data types.

Implicit creation of certificates for HTTPS
Certificates for HTTPS are now created implicitly when nothing is specified in FGLPROFILE.
Before version 2.50, certificates for HTTPS had to be specified explicitly in FGLPROFILE.
Starting with 2.50, no HTTPS certificate is defined in FGLPROFILE, when a web services program starts, the creation is implicit.

Related concepts
HTTPS configuration on page 3305
If no client certificate is provided, Genero Web Services (GWS) does the HTTPS request transparently.
FGLPROFILE entries for Web Services on page 3489
The FGLPROFILE entries relating to Genero Web Services are divided between five categories: security, basic or digest HTTP authentication, proxy configuration, server configuration, and XML cryptography.

PostgreSQL schema extraction needs namespace
To extract a database schema from PostgreSQL, the fgldbsch tool now requires db namespace specification.

In version 2.50, the fgldbsch database schema extractor can only extract the schema from a PostgreSQL database if you specify the -ow option.

PostgreSQL distinguishes table owners from table schemas (in other words table namespaces). The real table namespace is defined by the pg_class.relnamespace column: it contains the oid of a namespace defined in pg_namespace.

For PostgreSQL, the fgldbsch -ow option will specify the namespace, instead of the owner of the table, because a db user can create several schemas/namespaces and use the same table name in those different namespaces. As result, filtering on user name can mix up table definitions from different schemas/namespaces.

When extracting a database schema from a PostgreSQL database, you must specify the namespace of tables with the -ow option. If no -ow option is specified and the -un option is specified, fgldbsch will use the login name of the -un option as namespace. If neither -ow, nor -up options are specified, fgldbsch will use the PostgreSQL "public" namespace/schema by default.

Since database tables are usually created in the "public" namespace, you typically specify this namespace with the -ow option:

```
fgldbsch -db test1 -dv dbmpgs -un pgsuser -up fourjs -v -ow public
```

Related concepts
Specifying the table owner on page 478
fgldbsch on page 2079
The fgldbsch tool generates the database schema files from an existing database.

Client stubs managing multipart changes
You must update client programs that call client stubs managing multipart.

Important: This change has been backported from version 3.00

Starting with version 2.50.25, when generating client stubs managing multipart, you get an extra input and/or output variable called "AnyInputParts" and "AnyOutputParts" that is a DYNAMIC ARRAY of com.HTTPPart objects. Those variables may contain additional input and/or output HTTP parts not specified in the WSDL. You will have to adapt your client program by handling those dynamic arrays in any Genero functions calling such stubs.
Request example prior to 2.50.25:

```
FUNCTION xxx_g(InputHttpPart_1, ..., InputHttpPart_n)
  DEFINE InputHttpPart_1 com.HTTPPart
  ...
  DEFINE InputHttpPart_n com.HTTPPart
  ...
  RETURN wsstatus
END FUNCTION
```

Request example 2.50.25 and greater, with extra input variable AnyInputParts:

```
FUNCTION xxx_g(InputHttpPart_1, ..., InputHttpPart_n)
  DEFINE InputHttpPart_1 com.HTTPPart
  ...
  DEFINE InputHttpPart_n com.HTTPPart
  DEFINE AnyInputParts DYNAMIC ARRAY OF com.HTTPPart
  ...
  RETURN wsstatus
END FUNCTION
```

**Related concepts**

- The HTTPPart class on page 2902
  The `com.HTTPPart` class provides an interface to manage the HTTP attachment sent or received in HTTP.
- Multipart in the client stub on page 3332
  You can generate a client stub for a Web service that has multiple parts.
- Dynamic arrays on page 422

**BDL 2.41 upgrade guide**

These topics describe product changes you must be aware of when upgrading to version 2.41.

**Important:** This is an incremental upgrade guide that covers only topics related to the Genero BDL version specified in the page title. Check prior upgrade guides if you migrate from an earlier version. Make sure to also read about the new features for this Genero version.

There is no upgrade note with this version.

Corresponding new features page: BDL 2.41 new features on page 73.

**BDL 2.40 upgrade guide**

These topics describe product changes you must be aware of when upgrading to version 2.40.

**Important:** This is an incremental upgrade guide that covers only topics related to the Genero BDL version specified in the page title. Check prior upgrade guides if you migrate from an earlier version. Make sure to also read about the new features for this Genero version.

Corresponding new features page: BDL 2.40 new features on page 73.

**Web Services changes**

There are changes in support of web services in Genero 2.40.

**Modification to server location at runtime on client side**

**Important:**

It is recommended to regenerate all client stubs in your application using the `fglwsdl` tool.

If you have modified the server location at runtime via the generated global variable in your client application, you MUST apply following modification:
Prior to version 2.40, you had something similar to the following:

```
LET Calculator_CalculatorPortTypeLocation = "http://host:port/Calculator"
```

Starting with version 2.40, you must have something similar to the following:

```
LET Calculator_CalculatorPortTypeEndPoint.Address.Uri = "http://host:port/Calculator"
```

See Change client behavior at runtime.

**Database drivers changes**

Desupported database drivers.

**Databases / ODI drivers desupported in versions 2.40:**

- PostgreSQL versions 8.0, 8.1 and 8.2 (dbmpgs80x, dbmpgs81x, dbmpgs82x).
- Sybase Adaptive Server Anywhere (ASA) version 8 driver (dbmasa8x)

**Note:** The dbmasa8x driver was provided for the Sybase ASA desktop database engine (renamed Sybase SQL Anywhere when writing these lines). Since version 2.30, Genero Business Development Language supports a new driver to connect to SAP Adaptive Server Enterprise (ASE): dbmase*.

**Related concepts**

SAP ASE on page 1032

**Program size option removal (fglrun -s)**

The -s option of fglrun is no longer available.

Before version 2.30 the -s option of fglrun on page 2065 was used to compute the size of program elements such as global and module variables, p-code and structured data types. Starting with version 2.30, this option reported a size of zero. With version 2.40 the -s option is now fully desupported.

The -s option was mainly implemented for internal use. Regarding the amount of memory used by a program, it is recommended that you consider the memory allocated dynamically at runtime: If you fill large dynamic arrays, or leave a lot of SQL cursors open without freeing them, the memory footprint of a program can be much larger than the actual size of static elements that may be reported by the -s option.

**Related concepts**

Optimization on page 586
Programming tips and tricks to make your programs run faster.

Optimize your programs on page 587
This section contains programming tips to optimize the execution of your application.

**Informix® SERIAL emulation with SQL Server**

SERIAL type emulation has been enhanced for SQL Server.

**Using SCOPE_IDENTITY() to get the last sequence**

Before version 2.40, the SQL Server drivers (SNC, MSV, ESM, FTM) used the @@IDENTITY expression to retrieve the last generated identity column, if the native serial emulation is configured. But @@IDENTITY is not recommended, because it can return an identity value generated for another table in a trigger of the main table.

Starting with 2.40, the SQL Server drivers use the `SCOPE_IDENTITY()` function, which returns the last number generated in the current scope (ignoring identity numbers generated in triggers).
Regtable serial emulation trigger code change
When using the "regtable" serial emulation, the code of the triggers has changed in version 2.40, using now the SET NOCOUNT ON instruction. Existing serial triggers created by prior versions must be reviewed, to have the same trigger body in all tables, otherwise an SQL error is raised when executing INSERT statements.

Related concepts
SERIAL and BIGSERIAL data types on page 847

SIZEPOLICY attribute removal for containers
The SIZEPOLICY attribute is no longer available for layout containers like TABLE / GRID.

Before version 2.40 it was possible to specify a SIZEPOLICY attribute for several types of form elements, including containers such as TABLE, GRID. The SIZEPOLICY attribute makes no sense in containers and is only meaningful for leaf nodes (widgets such as EDIT, COMBOBOX). The form compiler will now report a syntax error if the SIZEPOLICY attribute is used in the definition of elements that are not widgets.

Related concepts
SIZEPOLICY attribute on page 1391
The SIZEPOLICY attribute is a sizing directive based on the content of a form item.

The LVARCHAR type in IBM® Informix® databases
Native LVARCHAR type of Informix is now mapped by default to a large VARCHAR in schema file.

Starting with version 2.40, the fgldbsch database schema extractor converts now by default IBM® Informix® LVARCHAR(N) types to VARCHAR2(N) with type code 201. Before 2.40, you had to pass -cv AAAB... option to avoid a conversion error when generating the schema file.

The static SQL syntax has been enhanced, to support the LVARCHAR type name in DDL statements such as CREATE TABLE. The non-Informix® ODI drivers have been adapted to convert LVARCHAR type names to VARCHAR.

Two-Pass reports can now use VARCHAR types with a size greater than 255 bytes (the temporary table will be created with an LVARCHAR column). However, the index is created as well, and IBM® Informix® IDS (version 11 when writing these lines) has a size limitation for indexes. You may get an SQL error -517 if the VARCHAR variable used to group / order rows in the report routine exceeds ~350 bytes (see IDS SQL error -517 for details).

Related concepts
Database schema extractor options on page 476
The fgldbsch tool extracts the schema description for an existing database.

Command reference on page 2065
Command line tools provided for BDL.

Right-trim collation for character types in SQLite
CHAR and VARCHAR columns in SQLite need to be defined with a TRIM collation to ignore trailing spaces in comparisons.

Since version 2.40, the SQLite database driver adds the COLLATE RTRIM keywords after the CHAR(N) and VARCHAR(N) types in CREATE TABLE statements, when Informix® emulation is enabled (the default). This collation clause forces SQLite to use right-trim comparison rules instead of the default binary mode. The binary mode requires to have the same number of trailing spaces in both character values to be equal. By using COLLATE RTRIM clause, the trailing blanks are trimmed and thus ignored. It is recommended that you also use [VAR]CHAR(N) COLLATE RTRIM in database creation scripts.

Related concepts
SQLite on page 1004
Message files support now 4-bytes integer message numbers

2-byte .msg message number limitation was removed.

Before version 2.40, message files entries were only defined with numbers in the range -32767 to 32767 (SMALLINT). This limitation is no longer true in 2.40; message numbers can now be in the range -2147483648 to 2147483647 (INTEGER).

Related concepts
Message files on page 1160
Message files centralize strings and larger texts identified by a number, that can be used in programs.

MySQL client library version change in MySQL 5.5.11

Shared library version number of the MySQL client library must match the library used to link the ODI driver.

Starting with Oracle® MySQL 5.5.11, the client library version number was changed from 16 to 18. In fact the libmysqlclient.so.16 file was renamed to libmysqlclient.so.18. From a cross-5.5.x compatibility point of view, this may not be the best thing to do, since the major shared library version has changed, client applications using the C API (such as Genero ODI MySQL drivers) need to be recompiled and re-linked in order to use the latest library.

In Genero version 2.40, the dbmmys55x ODI driver is linked with libmysqlclient.so.18 on the platforms where MySQL 5.5.11+ is available. That is: Linux®, Solaris and Mac OS X® platforms, when writing these lines. On other UNIX™ platforms such as HP, the client library is still libmysqlclient.so.16. This may change in future Genero versions, following the availability of MySQL 5.5.11+ versions.

Therefore, you must pay attention to the MySQL 5.5 version you have installed. You need to upgrade your MySQL 5.5 client software to match the client library used to build the dbmmys55x.so shared library. On Linux®, you can run the ldd command to check what libmysqlclient.so version is required. If it's not possible to upgrade your MySQL client software, please contact the support channel.

Related concepts
Oracle MySQL / MariaDB on page 876

New compiler warning to avoid action shadowing

Prevent the same action name at different levels of ON ACTION handlers in a dialog.

The fglcomp compiler of version 2.40 will now print warning -8409, if a dialog block defines ON ACTION handlers at different levels with the same action name.

It is not good practice to use the same action names at different levels of a dialog. For example, you can define several ON ACTION INFIELD handlers using the action name "zoom", but do not define an ON ACTION zoom at the sub-dialog or dialog level.

If the warning occurs during compilation, modify your code in order to use specific action names at each level, and do not forget to rename the actions of the corresponding action views in the forms.

Related concepts
Multilevel action conflicts on page 1768

Command reference on page 2065
Command line tools provided for BDL.

Runtime error raised when report dimensions are invalid

Report page length checking error -4375 might occur at compile time or runtime.

Several fixes related to PAGE LENGTH error -4375 have been done in subsequent releases.

Below is the history of changes for this report definition error:

1. Version 2.40 fixes bug FGL-3035: A START REPORT instruction raises error -4375 at runtime, when the top/bottom margin sizes do not fit the page length. In this version, the error is not returned at compile time, because report dimensions can be specified with variables in START REPORT.
2. Version 2.50.00 fixes bug FGL-3711: The compiler also raises error -4375, if PAGE LENGTH is too short to cover the specified page header and trailer lengths.

3. Version 2.51.07 fixes bug FGL-651: FIRST PAGE HEADER blocks can have the same number of rows as the PAGE LENGTH.

4. Version 2.50.26 fixes bug FGL-4223: Reports with PAGE LENGTH = 1 TOP MARGIN 0 BOTTOM MARGIN 0 are often used to produce reports without any page formatting. A page length of 1 denies printing in PAGE HEADER and PAGE TRAILER. However, it is possible to print any number of lines in FIRST PAGE HEADER and ON LAST ROW is possible.

**Related concepts**

Reports on page 2026

**Linker checks all referenced functions**

The linker checks definition of all functions referenced in all modules provided in the link command.

Starting with version 2.40, any reference to a function has to be resolved by the linker: When linking a 42r program, if an unused module references an undefined function, the linker (fglrun -l or fgllink) will stop with the error -1338. Before version 2.40, the undefined function was ignored.

**Note:** Complete function reference is only checked by the linker when creating a 42r program file. When creating a 42x library, there can be references to undefined functions.

In the example, the main.4gl module does not call any function, but the module used in the link line (module.4gl) defines an unused function (f1) calling an undefined function (f2):

```
main.4gl:
MAIN
   DISPLAY "In main..."
END MAIN

module.4gl:
FUNCTION f1() -- Unused in program
   DISPLAY "In f1..."
   CALL f2() -- Undefined
END FUNCTION
```

Compiling and linking:

```
$ fglcomp main.4gl
$ fglcomp module.4gl
$ fgllink -o prog.42r main.42m module.42m
ERROR(-1338):The function 'f2' has not been defined in any module in the program.
```

**Related concepts**

Linking programs on page 2115
Describes how to link .42m modules together to build a .42r program file.

Command reference on page 2065
Command line tools provided for BDL.

**BDL 2.32 upgrade guide**

These topics describe product changes you must be aware of when upgrading to version 2.32.

**Important:** This is an incremental upgrade guide that covers only topics related to the Genero BDL version specified in the page title. Check prior upgrade guides if you migrate from an earlier version. Make sure to also read about the new features for this Genero version.

Corresponding new features page: BDL 2.32 new features on page 77.
**Front-end protocol compression disabled**

GUI communication does not require protocol compression on LAN networks.

Until version 2.32.00, front-end protocol compression was enabled by default, to speed up GUI communication on slow networks. However, on regular networks, compression is useless and can be disabled to save processing resources. With version 2.32.00, the compression is now disabled by default. If needed, compression can be enabled with this FGLPROFILE entry:

```text
gui.protocol.format = "zlib"
```

Note also that compression needs the zlib library to be present on the computer where fglrun executes. Starting with 2.32.00, the product package no longer includes the fallback zlib library ($FGLDIR/lib/libzfgl.so or %FGLDIR%\bin\libzfgl.dll). If no standard zlib is installed on your system, compression will not be possible.

**Related concepts**

GUI protocol compression on page 1119
The FGLPROFILE file(s) on page 255
FGLPROFILE environment variable defines Genero BDL configuration files

**SQLite driver no longer needs libiconv on Windows®**

UTF-8 string data storage in SQLite requires conversion when the application is not UTF-8.

Starting with version 2.32, the SQLite driver (dbmsqt3xx) no longer needs the LIBICONV.DLL library on Windows® platforms to do charset conversion, when the application locale is not UTF-8.

**Related concepts**

SQLite on page 1004
Localization on page 512
Localization support allows you to implement programs that follow specific language and cultural rules.

**Need for Informix® CSDK to compile C extensions**

Compiling C Extensions requires now the Informix CSDK.

**Note:** This upgrade note is related to C Extensions or ESQL/C Extensions, and can be ignored if your application does not use such extensions.

To compile C or ESQL/C extensions manipulating data types like DECIMAL, you need IBM® Informix® data type structure definitions such as dec_t, dtime_t, intrvl_t, as well as macros like DECLEN() or TU_ENCODE(). Before version 2.32, these C structure and macros where provided in the files of the $FGLDIR/include/f2c directory.

The Genero BDL version 2.32 no longer provides the IBM® Informix® ESQL/C structure definitions in $FGLDIR/include/f2c files, because we have identified that some of the definitions are platform specific. However, to compile your C extensions, you need these definitions if your extensions use complex data types such as DECIMAL, DATETIME/INTERVAL, BYTE/TEXT. The definitions are not required if you use standard C types such as int or char[].

Starting with version 2.32, you need to install an IBM® Informix® CSDK on your development machine in order to get the structure and macro definitions to compile your C extensions. The IBM® Informix® CSDK is only required on the development platform. It is not required to install the CSDK on the production machines, except of course if you want to connect to an IBM® Informix® database server.

**Related concepts**

C-Extensions on page 2230
With C-Extensions, you can bind your own C libraries in the runtime system, to call C function from the application code.

**FESQLC tool removal**

The ESQL/C compiler (fesql) has been removed from the Genero BDL product.

Starting with version 2.32, the fesqlc compiler and linker is no longer part of the Genero BDL package.
Contact your support channel for more details.

**Related concepts**
- Command reference on page 2065
- Command line tools provided for BDL.

### BDL 2.30 upgrade guide

These topics describe product changes you must be aware of when upgrading to version 2.30.

**Important:** This is an incremental upgrade guide that covers only topics related to the Genero BDL version specified in the page title. Check prior upgrade guides if you migrate from an earlier version. Make sure to also read about the new features for this Genero version.

Corresponding new features page: BDL 2.30 new features on page 78.

#### GUI server auto start

FGLSERVER defaults the server defined by wsmap settings, when starting GUI server

Before version 2.30, the runtime system was trying to connect to `localhost:0` when FGLSERVER was not set, even if `gui.server.autostart` FGLPROFILE entries are defined.

This behavior has been identified as a bug (FGL-1583) and has been fixed, changing the way fglrun proceeds with the GUI connection when autostart settings are defined; with 2.30, the `wsmap` workstation mappings are now taken into account, so that FGLSERVER defaults to `localhost:n`, where `n` is the GUI server number found from the wsmap settings.

**Related concepts**
- Automatic front-end startup on page 1121

#### Form compiler is more strict

The .per grammar parser has been reviewed to deny invalid code.

In version 2.30, the internals of fglform have been reviewed to simplify the extension of the form syntax with new item types and attributes. This code review has removed some inconsistencies in the grammar parser; as a result, the form compiler is more strict regarding invalid syntaxes. Thus, you may experience compilation errors with forms that compiled with prior versions. Simply fix the invalid syntax in your forms and recompile.

**Related concepts**
- fglform on page 2068
  The fglform tool compiles form specification files into XML formatted files used by programs.

#### ORACLE and INTERVAL columns

INTERVAL storage bug fix needs a review of existing databases in production.

Before 2.30.00 (build 1566), negative (and only negative) INTERVAL values were inserted incorrectly. This a critical bug.

For example, it was not possible to compare an INTERVAL value inserted by a program with an INTERVAL literal:

```sql
SELECT ... FROM table
WHERE interval_col = INTERVAL '-55555-11' YEAR(9) TO MONTH
```

The problem concerns database columns with the following interval types:

- INTERVAL YEAR(p) TO MONTH
- INTERVAL DAY(p) TO FRACTION(n)

(Other INTERVAL types are stored in a CHAR(50))
A simple INTERVAL to CHAR to INTERVAL conversion will fix the values:

```
UPDATE table SET interval_col = TO_CHAR(interval_col)
```

**Related concepts**

**Interval expressions** on page 336
This section covers interval expression evaluation rules.

**Interval literals** on page 329
Interval literals define an interval value in an expression.

**DIALOG.setCurrentRow() changes row selection flags**
Row selection flags are reset by a call to setCurrentRow().

Before version 2.30, the `DIALOG.setCurrentRow()` method did not modify the row selection flags.

Starting with version 2.30, the method resets row selection flags to false and marks the new current row as selected.

**Related concepts**

**Multiple row selection** on page 1851
Multiple row selection allows the end user to select several rows within a list of records.

**Schema extractor needs table owner**
The `fgldbsch` schema extractor requires a `-ow` option to distinguish different database users/schemas.

Starting with version 2.30, the `fgldbsch` schema extractor will always use a table owner / schema to select tables from databases where several schemas can hold tables with the same name.

The table owner can be specified with the `-ow` option, and defaults to the user name passed with the `-un` option, or to the current database user if no `-up` option was given. The last case can occur when the database connection information is taken from the FGLPROFILE configuration file, or when the OS user authentication is used.

**Related concepts**

`fgldbsch` on page 2079
The `fgldbsch` tool generates the database schema files from an existing database.

**Windows™ installation for all users only**
Installation on Windows platforms is for all users.

Starting with version 2.30, the Windows® installer forces you to install the product for all users.

**Related concepts**

`Installation` on page 37
This chapter contains installation and setup instructions.

**MenuAction close no longer created by default**
The close action is no longer created by default in MENU dialog.

Before version 2.30, a close MenuAction was created by default for MENU dialogs. This action node is no longer created, except if you have a COMMAND KEY(INTERRUPT) in the MENU, or if you have your own user action handler ON ACTION close, of course. You must take this change into account if you are manipulating the AUI tree with om classes in MENUs.

**Related concepts**

`Ring menus (MENU)` on page 1467
The MENU instruction implements a list of options the end user can choose from.

**The DomNode class** on page 2584
The `on.DOMNode` class provides methods to manipulate a DOM node of a data tree.

**Emulated scrollable cursor temp files in DBTEMP**
Directory of scrollable cursor data storage can be defined with DBTEMP.

On UNIX™ platforms, starting with 2.30, the temporary files for emulated scrollable cursors will be created in the directory defined by the `DBTEMP` on page 273 environment variable when defined, otherwise `TMPDIR`, `TEMP` or `TMP` will be used. Using `DBTEMP` for database files conforms to `DBTEMP` usage for temporary files of TEXT and BYTE data storage.

**Related concepts**
- `TMPDIR`, `TMP`, `TEMP` on page 266
  Defines the directory for temporary files.
- TEXT on page 307
  The TEXT data type stores large text data.
- BYTE on page 290
  The BYTE data type stores any type of binary data, such as images or sounds.

**Modifying tree view data during dialog execution**
Use `ui.Dialog` methods to insert/append/delete treeview nodes.

Before version 2.30, it was possible to use the `insertRow() / appendRow() / deleteRow() / deleteAllRows()` dialog class methods to modify the tree array during the dialog execution. But these methods were not designed to handle tree data properly. An alternative was to use program array methods, but when modifying the program array directly, multi-range selection flags or cell attributes were not synchronized.

Starting with 2.30.02, you can now use the `insertNode(), appendNode() and deleteNode()` methods of the `ui.Dialog` class. You can still directly fill the program array before the dialog execution, but it is recommended to use dialog methods during the dialog execution.

**Related concepts**
- The `Dialog class` on page 2512
  The `ui.Dialog` class provides a set of methods to configure, query and control the current interactive instruction.
- Tree views on page 1889
  Describes how to implement tree views.

**Presentation styles changes**
Modifications to consider when using presentation styles.

**Form.resetFormSize style attribute**
Starting with Genero BDL 2.30, the Form presentation style `resetFormSize` can be used to control the window resizing behavior when doing successive `OPEN FORM / DISPLAY FORM`.

For more details, see Form style attributes on page 1196.

**Related concepts**
- Windows and forms on page 1131
  The section describes the concept of windows and forms in the language.

**BDL 2.21 upgrade guide**
These topics describe product changes you must be aware of when upgrading to version 2.21.

**Important:** This is an incremental upgrade guide that covers only topics related to the Genero BDL version specified in the page title. Check prior upgrade guides if you migrate from an earlier version. Make sure to also read about the new features for this Genero version.

Corresponding new features page: BDL 2.21 new features on page 80.
Web Services changes
There are changes in support of web services in Genero 2.21.

Operation publication restrictions on server side
If you use a variable as the name of the function to publish, you will get an error message at compile time. For example:

```java
com.WebOperation.CreateRPCStyle(test,"Add",add_in,add_out)
```
Where `test` is a string variable, `add_in` and `add_out` are input and output records. At compile time, you get the error message:

```
error:(-9054) Web service function must be a string
```
The function name in the parameter can only be a string literal not a string variable.

Since version 2.21, FGL has introduced the concept of PUBLIC/PRIVATE functions, there is a risk for a user publishing private functions. Private functions are not always available at runtime.

As a workaround you can add a switch based on the function name value in order to call the appropriate publication API with the name in a string literal as shown in the following sample:

```java
CASE function_name
  WHEN "Operation1"
    LET op = com.WebOperation.CreateDocStyle("Operation1","Operation1",op1_in,op1_out)
  WHEN "Operation2"
    LET op = com.WebOperation.CreateDocStyle("Operation2","Operation2",op2_in,op2_out)
  OTHERWISE
    DISPLAY "ERROR"
END CASE
```
In Java or in .NET you cannot publish a different number of operations for the same service, everything is done at compile time. For instance, when you publish a Web service in Java, only the public methods are published as operations of the service. There is no way to add or remove some methods at runtime. The only way you have is to create another Java class.

Be aware that if you dynamically change the service operations names, you are creating a different service, which might be confusing for the Web service client.

PostgreSQL 8.4 and INTERVAL type
The `dbmpgs84x` database driver requires your database schema use the INTERVAL type, rather than a CHAR(50) type.

Version 2.21 introduced support for PostgreSQL 8.4 with the new database driver `dbmpgs84x`. This version of PostgreSQL implements a native INTERVAL data type that is similar to the Genero Business Development Language INTERVAL type.

When using the `dbmpgs84x` (and higher) driver, Informix-style INTERVAL types will be mapped / translated to native PostgreSQL INTERVALS. Prior drivers will keep using the CHAR(50) replacement. If your application is storing INTERVAL in a PostgreSQL database, you will have to modify you database schema to replace the existing CHAR(50) column with the native INTERVAL data type of PostgreSQL 8.4. If you cannot migrate the database, you can still use the older `dbmpgs83x` driver using CHAR(50) for INTERVALS, but that driver requires a PostgreSQL client version 8.3.

Related concepts
PostgreSQL on page 964
**fglcomp --build-rdd compiles the module**

*fglcomp --build-rdd* now creates both the .42m and .rdd files.

Before version 2.21, *fglcomp --build-rdd* only produced the .rdd data definition file.

This option is now a compilation option. Both .42m and .rdd files are created at the same time.

**Related concepts**

- *fglcomp* on page 2071
  - The *fglcomp* tool compiles .4gl source files into .42m p-code modules, and does various other tasks.
- *Command reference* on page 2065
  - Command line tools provided for BDL.

**Unique and primary key constraint violation**

Unique and primary key constraint violations mostly return error -268. However, error -269 may be checked too.

When a unique or primary key constraint is violated, the IBM® Informix® driver returns the error -268 in SQLCA.SQLCODE if the database uses transaction logging, and error -239 if the database uses no logging.

Regarding non-Informix drivers, all 2.21 drivers now return -268 when a unique constraint or primary key constraint is violated. Before 2.21, the Oracle and SQL Server / SAP ASE drivers returned error -239, which is only returned by IBM® Informix® databases without transaction logging. Returning error -268 for all drivers is the best choice in a context of transactional databases.

Check your code for -239 error code usage and replace by -268. If you still need to test error -239 (for example because you have IBM® Informix® databases without transactions), we recommend that you write a function testing different error codes to check unique constraint violation:

```plaintext
FUNCTION isUniqueConstraintError()
   IF (SQLCA.SQLCODE==-239 OR SQLCA.SQLCODE==-268)
   OR (SQLCA.SQLCODE==-346 AND SQLCA.SQLERRD[2]==-100)
   THEN
      RETURN TRUE
   ELSE
      RETURN FALSE
   END IF
END FUNCTION
```

**IMPORT with list of C-Extensions**

The IMPORT instruction for C extensions denies a comma-separated syntax.

Before version 2.21.00, the IMPORT instruction for C extensions was documented as allowing a comma-separated list of libraries:

```plaintext
IMPORT lib1, lib2
```

This compiled, but at runtime only the first library was found. Using elements of the other libraries raised a runtime error.

With 2.21.00 and the new .42m module importation support, the compiler is now more strict and denies the comma-separated syntax. You must specify every library, Java class or .4gl module in separate lines:

```plaintext
IMPORT lib1
IMPORT JAVA myclass
IMPORT FGL mymodule
```

**Related concepts**

- *Importing modules* on page 480
  - Use the IMPORT ... instruction to import BDL, C or Java external modules in the current module.
- *Structure of a module* on page 462
A module defines a set of program elements such as functions, report routines, types, constants and variables.

**Initializing dynamic arrays to null**
The INITIALIZE TO NULL instruction clears the dynamic array.

Starting with version 2.21.00, the INITIALIZE TO NULL instruction clears the dynamic arrays (i.e. array.getLength() returns 0). Before this version, all elements of the dynamic array were kept, and set to null. Since the old behavior was documented, this behavior change required a migration note. The new behavior is expected by most programmers.

**Related concepts**
INITIALIZE on page 399
The INITIALIZE instruction initializes program variables with NULL or default values.

**Strict screen record definition for tables**
The fgform compiler of version 2.21.00 now makes a strict checking of the fields used in the screen record definition for table containers.

It generates error -6819 if the screen record do not use all columns used in the table. The order can be different, however.

**Related concepts**
TABLE container on page 1326
Defines a re-sizable table designed to display a list of records.

**BDL 2.20 upgrade guide**
These topics describe product changes you must be aware of when upgrading to version 2.20.

**Important:** This is an incremental upgrade guide that covers only topics related to the Genero BDL version specified in the page title. Check prior upgrade guides if you migrate from an earlier version. Make sure to also read about the new features for this Genero version.

Corresponding new features page: BDL 2.20 new features on page 85.

**Web Services changes**
There are changes in support of web services in Genero 2.20.

**Migration to 2.20 on client side**
If migrating a GWS client application to version 2.20, you need to regenerate all client stubs in your application using the fglwsdl tool.

**Important:**
This is mandatory. The regenerated code is based on low-level COM and XML APIs and is completely different from versions prior to 2.1x. If you do not regenerate your client stubs, you will not be able to execute the code.

**Backward compatibility option -compatibility**
When using a Genero 2.2x runner for the GWS client application, you must:

1. Regenerate the GWS client stubs using the -compatibility option of the fglwsdl tool, so the function prototypes will be compatible:
   ```
   ```

2. Compile the GWS client stubs and re-link the client application (.42r).
Renamed options

The `http_invoketimeout` and `tcp_connectiontimeout` options have been respectively renamed as `readwritetimeout` and `connectiontimeout`, as they are now available for either HTTP or TCP protocol. While the old option names remain for backward compatibility, using the new option names is strongly recommended.

Moved options

`xml_ignoretimezone` and `xml_usetypedefinition` options were part of the `com.WebServiceEngine` class. They have been moved to the `xml.Serializer` class, which groups functions on serialization.

Sort is now possible during INPUT ARRAY

Built-in sort is available in INPUT ARRAY. Starting with version 2.20, the built-in sort is now available during INPUT ARRAY. If you want to avoid sorts in a table, use the `UNSORTABLECOLUMNS` attribute.

Related concepts

`UNSORTABLECOLUMNS attribute` on page 1402
The `UNSORTABLECOLUMNS` attribute indicates that the columns of the table cannot be selected by the user for sorting.

`Editable record list (INPUT ARRAY)` on page 1531
The INPUT ARRAY instruction provides always-editable record list handling in an application form.

Cell attributes and buffered mode

Must use the UNBUFFERED mode when setting cell attributes.

Before version 2.20, array cell attributes were synchronized quite often by the runtime system, and this was not very efficient. As a result, there was not much difference between using buffered or unbuffered mode; when changing cell attributes, the result was immediate even in buffered mode.

Starting with version 2.20, it is recommended that you use the UNBUFFERED mode when setting cell attributes; otherwise, the colors will not be synchronized on the front-end.

Related concepts

`Cell color attributes` on page 1850
List controllers can display every cell in a specific color.

`The buffered and unbuffered modes` on page 1720
The buffered and unbuffered mode control the synchronization of program variables and form fields.

Field methods are more strict

Dialog class methods are more strict regarding form field names.

Starting with Genero 2.20 (or when using multiple dialogs in 2.11.08 and higher), DIALOG class methods such as `setFieldActive()` need the correct field specification with the screen-record name prefix, if the field was explicitly bound with the `FROM` clause of INPUT or INPUT ARRAY.

In prior versions, the field was found by these methods even if the prefix was invalid. (Actually, the prefix was just ignored and only the fieldname was used.)

Related concepts

`Identifying fields in ui.Dialog methods` on page 2556

Strict variable identification in SQL statements

Program variable identification in static SQL statements is more strict in version 2.20 than older versions.

If you define a variable with the same name as a SQL object (i.e. table name, table alias), the fglcomp compiler will raise an error because it will consider the program variable first. For example, if the variable name matches the table or alias identifier, using `table.column` in the SQL statement will be resolved as `variable.member`, which does not exist.
The next code example will not compile because the program defines a variable using the same name as the table alias `c`:

```main
DEFINE c INTEGER
DATABASE stores
SELECT COUNT(*) INTO c
  FROM customer c
  WHERE c.fname IS NULL
END MAIN
```

The code also fails to compile with IBM® Informix® 4GL 7.32, but it did compile with version of Genero Business Development Language.

To work around this, you must either rename the program variable, or explicitly identify SQL objects with the @ prefix in the SQL statement:

```main
DEFINE c INTEGER
DATABASE stores
SELECT COUNT(*) INTO c
  FROM customer c
  WHERE @c.fname IS NULL
END MAIN
```

Recompile all your programs to find the conflicts.

**SQL Warnings with non-Informix databases**

SQL Warnings are now propagated for all database drivers, and can set the SQLCA.SQLAWARN, SQLSTATE and SQLERRMESSAGE registers.

Before version 2.20, is was impossible for a non-Informix driver to return SQL Warning information in SQLCA, SQLSTATE and SQLERRMESSAGE.

This new behavior will have no impact if you test SQL Errors with STATUS or SQLCA.SQLCODE, as these registers remain zero if an SQL Warning is raised. However, if you are using SQLSTATE to check for SQL Errors, you must now distinguish SQLSTATE of class 01: These are SQL Warnings, not SQL errors.

In this example, when connected to IBM® DB2®, the SQLSTATE register will get the value 01504 indicating that all rows of the table have been deleted. As a result, testing SQLSTATE against 00000 will evaluate to false, and run into the error handling block, which is unexpected:

```main
DATABASE stores
WHENEVER ERROR CONTINUE
DELETE FROM customer
IF SQLSTATE <> "00000" THEN
  -- handle error
END IF
END MAIN
```

To check for successful SQL execution with or without warning, you can, for example, code:

```main
DATABASE stores
WHENEVER ERROR CONTINUE
DELETE FROM customer
IF NOT (SQLSTATE="00000" OR SQLSTATE MATCHES "01*") THEN
  -- handle error
END IF
END MAIN
```
SERIALREG table for 64-bit serial emulation
You must alter the SERIALREG table to do serial emulation on a BIGINT column.

The SERIALREG based serial emulation is defined by the following FGLPROFILE entry:

```plaintext
dbi.database.<dbname>.ifxemul.datatype.serial.emulation = "regtable"
```

Version 2.20 introduces the BIGINT data type, which is a 64-bit signed integer. You can use BIGSERIAL or SERIAL8 columns with IBM® Informix®, and ODI drivers can emulate 64-bit serials in other database servers. However, if you are using serial emulation based on the SERIALREG table, you must redefine this table to change the LASTSERIAL column data type to a BIGINT. If the BIGINT data type is not supported by the database server, you can use a DECIMAL(20,0) instead:

```sql
CREATE TABLE serialreg (  
tablename VARCHAR2(50) NOT NULL,  
lastserial BIGINT NOT NULL,  
PRIMARY KEY ( tablename )  
)
```

**Important:** If you need to migrate an installed database using SERIALREG-based triggers, you will have to keep the current registered serials and use ALTER TABLE instead of CREATE TABLE. This example shows the ALTER TABLE syntax for SQL Server. Check the database server manuals for the exact syntax of the ALTER TABLE statement.

```sql
ALTER TABLE serialreg ALTER COLUMN lastserial BIGINT NOT NULL
```

Additionally, all existing SERIALREG-based triggers must be modified, in order to use BIGINT instead of INTEGER variables, otherwise you will get BIGINT to INTEGER overflow errors. For example, to modify existing triggers with SQL Server, you can use the ALTER TRIGGER statement, which can be easily generated from the database browser tool (there is a modify option in the context menu of triggers). After the existing trigger code was generated, you must edit the code to replace the INTEGER data type by BIGINT in the variable declarations, and execute the ALTER TRIGGER statement.

**Extracting the database schema with fgldbsch**
The fgldbsch database schema extraction tool has been updated to map native database types to newly-added types.

Version 2.20 implements new data types such as BIGINT and BOOLEAN. The fgldbsch database schema extraction tool has been reviewed to map native database types to these new types when possible. Pay attention to these changes, when extracting a schema from your database.

For example, before version 2.20, fgldbsch converted an Oracle NUMBER(20,0) to a DECIMAL(20,0) by default. Now, since 2.20 provides the BIGINT native FGL type, it can be used to store a NUMBER(20,0) from Oracle.

You can get the previous behavior by using a conversion directive with the -cv option of fgldbsch.

To see the new conversion rules, run the fgldbsch tool with the -ct option.

**Related concepts**

- **BIGINT** on page 290
  The BIGINT data type is used for storing very large whole numbers.

- **BOOLEAN** on page 293
  The BOOLEAN data type stores a logical value, TRUE or FALSE.

- **fgldbsch** on page 2079
  The fgldbsch tool generates the database schema files from an existing database.

**Database driver internal error changed from -768 to -6319**
The internal error raised was changed to avoid conflicts with an IBM® Informix® SQL error code.

Prior to version 2.20, if an unexpected error occurred in a database driver, the driver could return error -768, which is a real IBM® Informix® SQL error that instructs the user to call the IBM® support center.
To avoid any mistake, 2.20 database drivers return now the error -6319 if an internal error occurs, which is a Genero Business Development Language specific error message that suggests you to set the FGLSQLDEBUG environment variable to get detailed debug messages.

**Related concepts**

FGLSQLDEBUG on page 281
Defines the debug level for tracing SQL instructions.

**Searching for image files on the application server**

For security reasons, the image file transfer mechanism has been slightly modified in version 2.20.

(This modification has also been back-ported in 2.11.14):

If FGLIMAGEPATH is set, the current working directory is no longer searched as in previous versions. You must explicitly add "," to the list of directories. By default, if FGLIMAGEPATH is not defined, the runtime system still searches the current directory.

If FGLIMAGEPATH is defined, the image files used in IMAGE form fields or in the IMAGE attribute must be located below one of the directories listed in the environment variable. This constraint does not exist if FGLIMAGEPATH is not set and has been relaxed in 2.21.00 for image fields displayed by program.

Starting with 2.21.00, images displayed by program to IMAGE fields are considered as valid files to be transferred to the clients without risk and do not follow the FGLIMAGEPATH security restrictions. Images are however searched for in the path list defined in FGLIMAGEPATH.

**Related concepts**

FGLIMAGEPATH on page 276
Defines the search paths for VM server image files.

IMAGE attribute on page 1370
The IMAGE attribute defines the image resource to be displayed for the form item.

IMAGE item type on page 1279
Defines an area that can display an image resource.

**Strict action identification in dialog methods**

Actions referenced in methods of the dialog class must exist in the current dialog, or an error is raised.

Starting with version 2.20.00, dialog class methods like `ui.Dialog.setActionActive()` can now raise a runtime error -8089 if the action name is invalid. Before version 2.20, the method ignored the invalid action name, which made it difficult for the programmer to debug.

**Related concepts**

`ui.Dialog.setActionActive` on page 2541
Enabling and disabling dialog actions.

**Strict field identification in dialog methods**

Fields referenced in methods of the dialog class must exist in the current dialog, or an error is raised.

Starting with version 2.20.05, dialog class methods like `ui.Dialog.setFieldTouched()` can now raise a runtime error -1373 if the field specified does not match a field in the current dialog. Before version 2.20.05, these methods previously ignored the invalid field specification, making it difficult for the programmer to quickly find the mistake.

**Related concepts**

`ui.Dialog.setFieldTouched` on page 2550
Sets the modification flag of the specified field.

**Form compiler checking invalid layout definition**

It is better to identify form layout mistakes when the form is compiled, rather than at runtime.

Starting with version 2.20.05, the fglform compiler performs more layout checking than before. Thus, existing (invalid) forms that compiled with prior versions of Genero may no longer compile with 2.20.05. This strict checking...
is done to detect layout mistakes during form design, instead of having the front-ends render invalid forms in a unknown manner at run time.

For example, the following form definitions are invalid and will raise a compilation error with fglform:

```plaintext
SCHEMA FORMONLY
LAYOUT
GRID
{
  [f01:  |f02    ]     -- HBox layout tags in lists are denied
  [f01:  |f02    ]
  [f01:  |f02    ]
  [f01:  |f02    ]
}
END
END

SCHEMA FORMONLY
LAYOUT
GRID
{
  [f01 ]  [f02    ]
  [f01 ]       [f02    ]   -- Misaligned field tags (vertical)
  [f01 ]  [f02    ]
  [f01 ]       [f02    ]
}
END
END

SCHEMA FORMONLY
LAYOUT
GRID
{
  [f01][f01][f01]  [f01]   -- Misaligned field tags (horizontal)
}
END
END
```

**Database schema compatibility**

fgldbsch extracts specific type for BOOLEAN.

Version 2.20.06 database schema extraction now generates a different type code for BOOLEAN, that introduces a compatibility issue with older versions of fglcomp and fglform.

If database tables use data types that are equivalent to the BOOLEAN Informix® type, such as the BIT type in SQL Server, you must regenerate the .sch database schema file with the fgldbsch tool. If you keep using the schema generated by an older version such as 2.20.04, fglcomp or fglform will raise the error -6634.

This problem will only occur if your database tables use the BOOLEAN (or native equivalent type). See SQL database guides on page 742 for more details about database specific boolean types.

**Related concepts**

Database schema on page 467
Defines database table structures with column type information to be reused in program variable definitions.

**Predefined actions get automatically disabled depending on the context**

Dialogs will automatically disable some predefined actions, if it makes no sense to trigger the action in the current context.

Starting with version 2.20, (or with version 2.10 when FGL_USENDIALOG=1), the dialogs will automatically disable some predefined actions if it makes no sense to trigger the action in the current context. For example, during
an INPUT ARRAY, if there are no rows to remove, the predefined delete action will be disabled automatically. Similarly, the insert and append actions get disabled when the array is full (this can happen with static arrays or when using the MAXCOUNT attribute). The predefined actions will also be disabled if you overwrite them with your own ON ACTION handler.

**Related concepts**

Predefined actions on page 1740
Genero predefines some action names for common operations of interactive instructions.

**BEFORE ROW no longer executed when array is empty**
In order to trigger the BEFORE ROW block when entering an array, the array must not be empty.

Before version 2.20, the BEFORE ROW block was always executed when entering a DISPLAY ARRAY or INPUT ARRAY dialog, even if the number of real data rows was zero. Starting with 2.20, when using an empty dynamic array or when using a static array and specifying zero data rows with a SET_COUNT(0) call or with the COUNT=0 attribute, the BEFORE ROW control block is no longer executed when the dialog starts.

The BEFORE ROW block will be executed when a new row is created in INPUT ARRAY. When entering an INPUT ARRAY with an empty array, a new temporary row is created by default, except if you use the AUTO_APPEND = FALSE attribute.

**Related concepts**

BEFORE ROW block on page 1510

**Controlling INPUT ARRAY temporary row creation**
Down move after last row in INPUT ARRAY creates a new temporary row.

The INPUT ARRAY dialog and sub-dialog provides the APPEND ROW and AUTO APPEND attributes to control row creation at the end of a list (known as temporary row creation).

APPEND ROW controls explicit temporary row creation, while AUTO APPEND controls automatic temporary row creation.

Starting with version 2.20, moving down after the last row (with the mouse or keyboard) or leaving the last column of the last row with a TAB key are considered events that trigger automatic temporary row creation.

Before version 2.20, these cases were considered as events for an explicit temporary row creation. In other words, if you want to deny temporary row creation in such case, it is now done with AUTO APPEND = FALSE while in older versions it was controlled by APPEND ROW = FALSE.

**Related concepts**

Appending rows in INPUT ARRAY on page 1848
Rows appended at the end of an editable list are temporary until they are edited.

**BDL 2.11 upgrade guide**
These topics describe product changes you must be aware of when upgrading to version 2.11.

**Important:** This is an incremental upgrade guide that covers only topics related to the Genero BDL version specified in the page title. Check prior upgrade guides if you migrate from an earlier version. Make sure to also read about the new features for this Genero version.

Corresponding new features page: BDL 2.11 new features on page 88.

**Writing timestamp information in p-code modules**
A compilation timestamp is no longer automatically written to p-code files, when the source code is not modified.

Before release 2.10, the 42m p-code files were stamped with a compilation timestamp. This information changed after every compilation, even if the source code was not modified.
Since 2.10, the timestamp information is no longer written to p-code files by default, allowing 42m file comparison, checksum creation, or storage of 42m file in versioning tools. Version 2.11.05, provides the fglcomp on page 2071 --timestamp option to force a timestamp in p-code modules:

```
$ fglcomp --timestamp mymodule.4gl
$ fglrun -b mymodule.42m
2008-12-24 11:22:33 2.11.05-1169.84 /home/devel/stores/mymodule.4gl 15
```

**Related concepts**

**Compiling source files** on page 2106
Describes how to build the runtime files from source files.

**BDL 2.10 upgrade guide**

These topics describe product changes you must be aware of when upgrading to version 2.10.

**Important:** This is an incremental upgrade guide that covers only topics related to the Genero BDL version specified in the page title. Check prior upgrade guides if you migrate from an earlier version. Make sure to also read about the new features for this Genero version.

Corresponding new features page: BDL 2.10 new features on page 91.

**XML declaration added automatically**

The XML declaration is added automatically when writing XML files.

An XML file must start with a "Prolog" or "XML Declaration" defining the XML version and character set used by the file:

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<root ...>
  ...
</root>
```

Starting with Genero version 2.10.05, the XML declaration is now added automatically when writing XML files.

Before 2.10.05, a workaround allowed you to write this header as a processing instruction, but this solution was subject to mistakes: the non-ASCII characters written to the XML file must match the encoding specification in the XML Declaration.

To avoid invalid character set definitions, the Genero BDL built-in classes now add the XML Declaration with the encoding attribute defined depending on the current locale used by the runtime system. The value written in the encoding attribute is defined by the charmap.alias file.

**Related concepts**

**C-Extensions** on page 2230
With **C-Extensions**, you can bind your own C libraries in the runtime system, to call C function from the application code.

**The om package** on page 2577
These topics cover the built-in classes of the om package

**Using SQL Server 2008 date/time types**

SQL Server 2008 introduces new SQL type to store date/time information.

Starting with Genero 2.10, SQL Server 2008 is now supported with the dbmsncA0 driver.

SQL Server 2008 introduces new data types to store date/time information, namely TIME(n), DATE, DATETIME2(n) and DATETIMEOFFSET(n). These data types offer better precision for data storage, as well as a one-to-one mapping for equivalent Genero BDL types (BDL DATE = SQL Server DATE).

When connected to SQL Server 2008 and higher, Genero database drivers for SQL Server use these new data type, when converting a CREATE TABLE / ALTER TABLE statement, and to store Genero BDL DATE and DATETIME information.
Therefore, upgrading to SQL Server 2008 requires planning the SQL Server database migration, in order to use the new date/time types provided in SQL Server 2008.

Note:

Up to Genero version 2.50, ODI drivers are built for a given database client and database server version. For example, `dbmfmtm90` (using FreeTDS) is designed for SQL Server 2005, and will use old date/time types. Pay attention to the fact that drivers designed for an old SQL Server version (2005), can also work with a more recent SQL Server version (2008), but will act as if they were connected to SQL Server 2005 (using old date/time types).

Starting with Genero 3.00 (2.51 for Genero Mobile), ODI drivers can detect the database server version at runtime, and adapt the SQL conversions to the target server: Assuming that the FreeTDS version supports the required TDS protocol version to connect to the target server version, the `dbmfmtm_0` ODI driver can connect to SQL Server 2005 or 2008, and adapt the date/time type usage to the targeted SQL Server version.

However, since Genero 3.10, SQL Server 2005 is no longer supported and only new SQL Server 2008 date/time types can be used.

For more details about BDL to SQL Server type mappings, see DATE and DATETIME data types on page 844 and Data type conversion table: Informix to SQL Server on page 837.

**BDL 2.02 upgrade guide**

These topics describe product changes you must be aware of when upgrading to version 2.02.

**Important:** This is an incremental upgrade guide that covers only topics related to the Genero BDL version specified in the page title. Check prior upgrade guides if you migrate from an earlier version. Make sure to also read about the new features for this Genero version.

Corresponding new features page: BDL 2.02 new features on page 94.

**Automatic HBox/VBox**

fglform adds automatically HBox/VBox elements when needed.

Starting with version 2.02.01, the form compiler automatically adds HBox and VBox containers with splitter around stretchable form elements that are placed side-by-side.

When recompiling your forms with this new version of fglform on page 2068, the generated .42f can get additional HBox/VBox nodes even if you did not touch the .per source file.

**Related concepts**

Form rendering on page 1415

The section explains the layout rules to render forms on graphical front-ends.

Compiling form specification files (.per) on page 2106

The .per form definition files must be compiled to .42f XML files, in order to be loaded by the runtime system.

**BDL 2.01 upgrade guide**

These topics describe product changes you must be aware of when upgrading to version 2.01.

**Important:** This is an incremental upgrade guide that covers only topics related to the Genero BDL version specified in the page title. Check prior upgrade guides if you migrate from an earlier version. Make sure to also read about the new features for this Genero version.

There is no upgrade note with this version.

Corresponding new features page: BDL 2.01 new features on page 95.

**BDL 2.00 upgrade guide**

These topics describe product changes you must be aware of when upgrading to version 2.00.

**Important:** This is an incremental upgrade guide that covers only topics related to the Genero BDL version specified in the page title. Check prior upgrade guides if you migrate from an earlier version. Make sure to also read about the new features for this Genero version.
Web Services changes
There are changes in support of Web services in Genero 2.00.

Using the 1.3x Web Services API
There is no need to create a special runner for Genero Web Services 2.x. Instead, the GWS 2.x library is imported into your applications. If you want to migrate your existing 1.x GWS Server application to 2.x, to avoid the need for a special runner, as well as to take advantage of any bug fixes, take the following steps:

1. Add the following statement at the top of any .4gl module where before you used GWS 1.3x functions:

   ```
   import com
   ```

2. Compile and re-link your GWS Server application (.42r).

   This imports the new GWS com library, and ensures that any GWS 1.3x functions that you have used will be compatible. Your existing Genero 1.3x Client applications, as well as third-party Client applications, will continue to work.

Using the new 2.00 Web Services API
If you want to take advantage of the new features and simplify future migrations, you can migrate your Genero Web Services (GWS) Server runner and also use the new GWS 2.x APIs. All the 1.3x publishing functions for all the operations in your application must be replaced with 2.x publishing functions. Since this does not change the interface, all existing Genero 1.3x client applications, as well as third-party client applications, will continue to work.

As 1.3x only supports RPC-Encoded style services, you must use the RPC style functions of the new 2.x APIs as the replacement functions, with `setInputEncoded` and `setOutputEncoded` set to true. And, you cannot add XML attributes to the records used as Web Service function parameters.

To replace the `fgl_ws_server_publishfunction()` statement in an existing GWS Server application; for example:

```
CALL fgl_ws_server_publishfunction(
   "EchoInteger",
   "http://tempuri.org/webservices/types/in", "echoInteger_in",
   "http://tempuri.org/webservices/types/out", "echoInteger_out",
   "echoInteger")
```

1. Add this statement at the top of each module:

   ```
   import com
   ```

2. Define variables for the WebService and WebOperation objects:

   ```
   DEFINE serv  com.WebService
   DEFINE op    com.WebOperation -- Operation of a WebService
   ```

3. Create the GWS Server object:

   ```
   LET serv = com.WebService.CreateWebService(
       "EchoInteger",
       "http://tempuri.org/webservices")
   ```

4. Use the 2.x publishing functions for each operation:

   ```
   LET op = com.WebOperation.CreateRPCStyle(
       "echoInteger",
       "EchoInteger",
       echoInteger_in,
       echoInteger_out,
       echoInteger_out)
   ```
5. Compile and re-link your GWS Server application (.42r)

GWS 2.x also allows your Server application (.42r) to contain multiple services. If you would like 2.x and 1.3x GWS to coexist in the same .42r executable, replace the existing publishing 1.3x functions.

**Enhance the GWS server application to be WS-I compliant (recommended)**

**Important**: You must be able to change all the client applications that access your migrated Genero Web Services (GWS) Server.

If you use the Literal styles now available in GWS 2.x for your Web service, your application will be WS-I compliant. However, the migration techniques still use the RPC/Encoded style (Only RPC/Encoded was supported in GWS 1.3x.). If you can change all the client applications that access your migrated GWS Server, we recommend that you enhance the GWS Server application to be WS-I compliant.

1. **Replace the publishing functions in the GWS Server application** but omit the `setInputEncoded` and `setOutputEncoded` lines. The resulting style will be Literal.

2. The enhanced GWS Server will have a new RPC/Literal WSDL that must be used to regenerate the client stub with the `fglwsdl` tool:

   ```
   fglwsdl -o NewClientstub http://localhost:8090/MyCalculator?WSDL
   ```

3. Compile that new client stub, and re-link it with the GWS client application. This operation must be repeated for each client application accessing that service.

4. Third party client applications must also be changed to use the new WSDL.

**Runner creation is no longer needed**

Starting with version 2.00, you no longer need to recompile/build a runner.

The runtime system architecture is now based on shared libraries (or DLLs on Windows®), and the database drivers are automatically loaded according to FGLPROFILE configuration parameters.

If you have C extensions, you must rebuild them as shared libraries.

**Important**: Database vendor client libraries (libclntsh, libcli, libpq, libaodbc) must be provided as shared objects (or DLL on Windows®).

**Related concepts**

[C-Extensions](#) on page 2230

With C-Extensions, you can bind your own C libraries in the runtime system, to call C function from the application code.

**Desupported Informix® client environments**

Upgrade IBM® Informix® Client Software Development Kit (CSDK) to the most recent version.

Always upgrade the IBM® Informix® Client Software Development Kit (CSDK) to the most recent version supported by Genero BDL.

The database interface of Genero Business Development Language (BDL) version 2.00 was redesigned to allow dynamic loading of database drivers. The following IBM® Informix® drivers and environments have been desupported with this redesign:

- ix210: Informix® ESQL/C 2.10
- ix410: Informix® ESQL/C 4.10
- ix501: Informix® ESQL/C 5.01
- ix711: Informix® ESQL/C 7.11
- ix720: Informix® ESQL/C 7.20
If required, old IBM® Informix® drivers can be re-enabled in a next Genero BDL version. However, we strongly recommend you to upgrade the IBM® Informix® Client Software Development Kit (CSDK) to the most recent version supported by Genero BDL.

Related concepts
Installation on page 37
This chapter contains installation and setup instructions.

Database drivers changes
Desupported database drivers.

Databases / ODI drivers desupported in versions 2.00:
- Adabas D 12 (dbmabd12)
- PostgreSQL 7 (dbmpgs7x)
- SQL Server 7 (dbmmsv7x)

Related concepts
Database driver specification (driver) on page 662

fglmkrutm tool removed
The fglmkrutm tool has been removed, as database drivers are loaded dynamically.

Starting with version 2.00, database drivers are now always loaded dynamically. Thus the fglmkrutm tool has been removed from the distribution. This tool was provided in previous versions to create a fgldr run runner with the correct database driver.

Refer to Database connections on page 656 for more details about database driver configuration.

Related concepts
Database driver specification (driver) on page 662
Command reference on page 2065
Command line tools provided for BDL.

fglinstall tool removed
The fglinstall tool has been removed from the distribution.

This tool was provided in previous versions to compile product message files, form files, and program modules provided in the distribution. The compiled versions of all these files are now included in the package.

Related concepts
Command reference on page 2065
Command line tools provided for BDL.

Linking the utility functions library
All utility functions are in the libfgl4js.42x library, up until 2.21.

Prior to version 2.00, some utility functions (canvas draw* and database db_* functions) were linked automatically to the 42r program when using fgldr run -l or fgllink. These functions are implemented in the fgldraw.4gl and fgldbutl.4gl modules, which were linked in the libfgl.42x library and loaded automatically at runtime by fgldr run.

Starting with version 2.00, all utility functions are now in the libfgl4js.42x library. So, if you use the draw* or db_* utility functions, you must now add the libfgl4js.42x library explicitly when using fgldr run -l or fgllink, or you can use the fgl2p on page 2070 tool to link .42x programs. The fgl2p tool links the program with the libfgl4js.42x library by default.

Starting with version 2.21, the libfgl4js.42x library is no longer provided.

Related concepts
Utility modules on page 2301
A utility function is a function provided in a separate library; it is not built in the runtime system.

Command reference on page 2065
Command line tools provided for BDL.

**Dynamic C extensions**
Dynamic C extensions are automatically loaded with **IMPORT** instructions.

Prior to version 2.00, you had to use **FGLPROFILE** entries to specify Dynamic C extensions to be loaded at runtime.

Starting with version 2.00, Dynamic C extensions are automatically loaded with **IMPORT** instructions. The FGLPROFILE entries are no longer used.

**Important:** Global variables (userData) can no longer be shared between the runtime system and the C extensions. You must use functions to pass global variable values.

There is no longer any need to define the FGL_API_MAIN macro in the extension interface file.

All C data type definitions are now centralized in the `fglExt.h` header file; header files such as `Date.h`, `MyDecimal.h` have been removed from the distribution.

**Related concepts**
- **C-Extensions** on page 2230
  With **C-Extensions**, you can bind your own C libraries in the runtime system, to call C function from the application code.

**Example 4: Global variables** on page 404

**WANTCOLUMNSANCHORED is desupported**
Use **UNMOVABLECOLUMNs** to specify that table columns cannot be moved around by the user.

Before version 2.00, the WANTCOLUMNSANCHORED attribute was undocumented but still supported by the language, to simplify migration from 1.20.

Starting with version 2.00, the WANTCOLUMNSANCHORED attribute is desupported; you must use **UNMOVABLECOLUMNs** to specify that table columns cannot be moved around by the user.

**Related concepts**
- **UNMOVABLECOLUMNs attribute** on page 1400
  The UNMOVABLECOLUMNs attribute prevents the user from moving columns of a table.

**PIXELWIDTH / PIXELHEIGHT are deprecated**
Use the **WIDTH** and **HEIGHT** attributes to specify the size of an image.

Before version 2.00, the PIXELWIDTH and PIXELHEIGHT attributes were used to specify the real size of an IMAGE form item.

Starting with version 2.00, you must use the **WIDTH attribute** on page 1411 and **HEIGHT attribute** on page 1368 to specify the size of an image:

In the .per form file:

```plaintext
IMAGE img1 = FORMONLY.image1,
    HEIGHT = 100 PIXELS,
    WIDTH = 100 PIXELS;
```

The PIXELWIDTH and PIXELHEIGHT attributes are still supported by the form compiler, but are deprecated and will be removed in a future version.

**Related concepts**
- **IMAGE item type** on page 1279
Defines an area that can display an image resource.

**Prefetch parameters with Oracle**

Prefetch parameters allow an application to automatically fetch rows from the Oracle® database when opening a cursor.

Before version 2.00, the default prefetch parameters are 50 rows and 65535 bytes for the prefetch buffer. Some customers experienced a huge memory usage with those default values, when using a lot of cursors: It appears that the Oracle® client is allocating a buffer of prefetch.memory (64 Kbytes) for each cursor.

Starting with version 2.00, the default is 10 rows and 0 (zero) bytes for the prefetch buffer (memory), meaning that memory is not included in computing the number of rows to prefetch.

**Related concepts**

[Database connections](#) on page 656
Explains how to manage database connections in a program.

**Preprocessor directive syntax changed**

The preprocessor directives use an ampersand character (&) instead of a hash (#) character.

Before version 2.00, the preprocessor directives start with a (#) hash character, to be compliant with standard preprocessors (like cpp). This caused too many conflicts with standard language comments that use the same character:

```plaintext
#include "myheader.4gl"
# This is a comment
```

Starting with version 2.00, the preprocessor directives use an ampersand character (&):

```plaintext
&include "myheader.4gl"
FUNCTION debug( msg )
  DEFINE msg STRING
&ifdef DEBUG
    DISPLAY msg
&endif
END FUNCTION
```

The preprocessor is now integrated in the compiler, to achieve faster compilation.

**Important:** To simplify the migration, the # hash character is still supported when using the -p fglpp option of the compiler. However, it is recommended that you review your source code and use the & character instead; # hash will be desupported in a future version.

**Related concepts**

[Source preprocessor](#) on page 2132
A typical preprocessor like in the C language.

**Static SQL cache is removed**

The Static SQL Cache has been removed.

Before version 2.00, the size of the static SQL cache is defined by an FGLPROFILE entry:

```plaintext
dbi.sql.static.optimization.cache.size = max
```

This entry was provided to optimize SQL execution without touching code using a lot of static SQL statements, especially when using non-Informix® database servers where the execution of static SQL statements is slower than with Informix®. This is useful for fast migrations, but there were a lot of side effects and unexpected errors.

Starting with version 2.00, the Static SQL Cache has been removed for the reasons described. Programs continue to run without changing the code, but if you want to optimize program execution, you must use dynamic SQL (PREPARE + EXECUTE).
Related concepts
Dynamic SQL management on page 709
Explains how to execute and manage SQL statements at runtime.

SQL directive set removed
The SQL directive set specification has been removed.

Before version 2.00, it was possible to define SQL directive sets in FGLPROFILE:

```sql
  dbi.sqldirset.set-name.directive.directive-name.{deflist|substrg} = "value"
```

With this feature, one could write SQL statement with database specific syntax, to be adapted at runtime depending on the target database type:

```sql
-- substrg directive:
  dbi.sqldirset.ora.directive.trunc.substrg = "TRUNCATE('$(1)'")"
  "SELECT * FROM customer WHERE {% trunc custname} = ?"

-- ifdef directive:
  dbi.sqldirset.ifx.directiveifdef.deflist = "INFORMIX"
  dbi.sqldirset.ora.directiveifdef.deflist = "ORACLE,ANSI"

  "SELECT * FROM"
  " {% ifdef INFORMIX "\"t1, OUTER(t2)\""}"
  " {% ifdef ORACLE \"t1, t2\"}"
  " WHERE t1.key = t2.col1 {% ifdef ORACLE \"(+)\"}"
```

Starting with version 2.00, SQL directive sets are no longer supported. Consider writing SQL statements with a syntax that is supported by all target database servers, or use dynamic SQL if you need to adapt the SQL syntax to the database type.

Related concepts
Dynamic SQL management on page 709
Explains how to execute and manage SQL statements at runtime.

Static SQL cache is removed on page 196
The Static SQL Cache has been removed.

Connection database schema specification
Oracle- and DB2-specific FGLPROFILE entries can be specified to define the database schema at runtime.

Before version 2.00, an FGLPROFILE entry was specified to define the database schema at runtime:

```sql
  dbi.database.dbname.schema = "schema-name"
```

This entry was used to select the native database schema after connecting to the server, for Oracle® and DB2® only.

Starting with version 2.00, this entry is now specific to the Oracle® and DB2 database driver configuration parameters:

```sql
  dbi.database.dbname.ora.schema = "schema-name"
  dbi.database.dbname.db2.schema = "schema-name"
```

For other database servers, this configuration parameter is not defined.

**Important**: It is no longer possible to specify the "schema" parameter in the connection string (dbname +schema=’name’).

Related concepts
Database connections on page 656
Explains how to manage database connections in a program.

**Schema extraction tool changes**
The fgldbsch schema extractor is recommended, and has been enhanced.

**Unique tool**
Version prior to 2.00 two schema extractors were provided: fglschema and fgldbsch on page 2079. The first one can only extract schemas from Informix® databases, while the second one can extract schemas from all supported databases.

Starting with version 2.00, the fgldbsch tool has been extended to support the old fglschema options, and fglschema has been replaced by a simple script calling fgldbsch. When you call fglschema, you actually call fgldbsch. We recommend that you use fgldbsch with its specific command line options.

**System tables**
In 2.0x, fgldbsch does not extract system tables by default. You must specify the -st option to get the system tables description in the schema files.

**Remote synonyms**
The original fglschema tool searched for remote synonyms with Informix® databases. The fgldbsch tool of version 2.00 does not search for remote synonyms.

**Public and private synonyms**
Since version 1.32.1b (build 620.313), fgldbsch does not extract private synonyms anymore. Only public synonyms are extracted. The .sch schema files do not contain table owners, and if two private synonyms have the same names, there is no way to distinguish them in the schema files. Therefore, to avoid any mistakes, private synonyms are not extracted anymore.

**Related concepts**
[Database schema on page 467](#)
Defines database table structures with column type information to be reused in program variable definitions.

[Database schema extractor options on page 476](#)
The fgldbsch tool extracts the schema description for an existing database.

**Global and module variables using the same name**
Program variable names must be different in global and module scope.

Since version 2.00, when you declare a module variable with the same name as a global variable, a compilation error must be thrown.

This is critical to avoid confusion with the variable usage:

```plaintext
GLOBALS
  DEFINE level INTEGER
END GLOBALS

GLOBALS "globals.4gl"
DEFINE level INTEGER
FUNCTION func1()
  LET level = 123  -- is this the global or the module variable?
END FUNCTION
```

Before version 2.00, the compiler did not detect this and the module variable was used, but one might want to use the global variable instead!
If you have module variables defined with the same name as global variables, the compiler now raises the following error:

```
-4319: The symbol 'variable-name' has been defined more than once.
```

You can easily fix this by renaming the module variable. There is no risk in doing this modification, because in versions before 2.00, the module variable was used, not the global variable.

Remark: The compiler now also detects duplicate global variable declaration. Just remove the duplicated lines in your source.

**Related concepts**

- **Globals** on page 551
  Global variables can be shared among all modules of a program.
- **Variables** on page 390
  Explains how to define program variables.
- **Importing modules** on page 480
  Use the `IMPORT ...` instruction to import BDL, C or Java external modules in the current module.

**Connection parameters in FGLPROFILE when using Informix®**

The `dbi.database.*` connection parameters defined in FGLPROFILE are used by the Informix® driver.

Before version 2.00, the `dbi.database.*` connection parameters defined in FGLPROFILE are ignored by the Informix® drivers.

Starting with version 2.00, the `dbi.database.*` connection parameters defined in FGLPROFILE are used by the Informix® driver, as well as other database vendor drivers. For example, if you connect to the database "stores", and you have the following entries defined, the driver tries to connect as "user1" with password "alpha":

```
dbi.database.stores.username = "user1"
dbi.database.stores.password = "alpha"
```

You typically get SQL errors -387 or -329 when the wrong database login or the wrong database name is used.

**Related concepts**

- **Database connections** on page 656
  Explains how to manage database connections in a program.

**Inconsistent USING clauses**

Having data types changing at each execute is no longer supported.

**Important:** This issue applies to non-Informix databases only.

Before version 2.00, it was possible to execute a prepared statement with the variable list changing at each `EXECUTE` statement:

```
DEFINE var1 DECIMAL(6,2)
DEFINE var2 CHAR(10)
DEFINE var3 DATE
PREPARE st1 FROM "INSERT INTO tab1 VALUES ( ?, ?, ? )"
EXECUTE st1 USING var1, var2, var3
EXECUTE st1 USING var2, var3, var1 -- different order = different data types
```

The database interface of version 2.00 has been rewritten for better performance. Having data types changing at each execute is no longer supported.

Error -254 will be raised if different data types are used in subsequent `EXECUTE` statements (with the same statement name).
Related concepts

**Database transactions** on page 600
Database transactions define a set of SQL instructions to be executed as a whole, or rolled back as a whole.

**Usage of RUN IN FORM MODE**

RUN ... IN LINE MODE is recommended to run interactive applications.

Before version 2.00, RUN ... IN FORM MODE was recommended to run interactive applications.

Starting with version 2.00, RUN ... IN FORM MODE is recommended to run interactive applications. The RUN command is used as follows (in both GUI and TUI mode):

1. When starting an interactive program, either use RUN ... IN FORM MODE or, if the default mode is LINE MODE, use the RUN instruction without any option.
2. When starting a batch program that does not display any message, it is recommended that you use RUN ... IN FORM MODE.

For more details about the RUN options, see the **RUN** on page 490 instruction.

Related concepts

**Program execution** on page 488
This section describes program execution and language instructions related to program execution.

**TTY and COLOR WHERE attribute**

All types of fields now allow TTY attributes and the conditional COLOR WHERE attribute.

Before version 2.00, only some field types like EDIT or TEXTEDIT provided support for TTY attributes (COLOR, REVERSE), and the conditional COLOR WHERE attribute.

Starting with version 2.00, all field types allow TTY attributes and the conditional COLOR WHERE attribute. This means that when using ATTRIBUTES(tty-attribute) in programs, all fields will now be affected.

For example, CHECKBOX and RADIOGROUP fields will now get a colored background, this was not the case in prior versions.

Related concepts

**COLOR attribute** on page 1357
The COLOR attribute defines the foreground color of the text displayed by a form element.

**BDL 1.33 upgrade guide**

These topics describe product changes you must be aware of when upgrading to version 1.33.

**Important:** This is an incremental upgrade guide that covers only topics related to the Genero BDL version specified in the page title. Check prior upgrade guides if you migrate from an earlier version. Make sure to also read about the new features for this Genero version.

Corresponding new features page: **BDL 1.33 new features** on page 99.

**Database drivers changes**

Desupported database drivers.

**Databases / ODI drivers desupported in versions 1.33:**

- MySQL 3.23.x (dbmmys32x)

Related concepts

**Database driver specification (driver)** on page 662
BDL 1.32 upgrade guide
These topics describe product changes you must be aware of when upgrading to version 1.32.

Important: This is an incremental upgrade guide that covers only topics related to the Genero BDL version specified in the page title. Check prior upgrade guides if you migrate from an earlier version. Make sure to also read about the new features for this Genero version.

There is no upgrade note with this version.

Corresponding new features page: BDL 1.32 new features on page 100.

BDL 1.31 upgrade guide
These topics describe product changes you must be aware of when upgrading to version 1.31.

Important: This is an incremental upgrade guide that covers only topics related to the Genero BDL version specified in the page title. Check prior upgrade guides if you migrate from an earlier version. Make sure to also read about the new features for this Genero version.

There is no upgrade note with this version.

Corresponding new features page: BDL 1.31 new features on page 100.

BDL 1.30 upgrade guide
These topics describe product changes you must be aware of when upgrading to version 1.30.

Important: This is an incremental upgrade guide that covers only topics related to the Genero BDL version specified in the page title. Check prior upgrade guides if you migrate from an earlier version. Make sure to also read about the new features for this Genero version.

Corresponding new features page: BDL 1.30 new features on page 101.

Action and field activation
Dialog methods can be used to control action and field activation.

Version 1.30 provides dialog methods to control action and field activation:

• ui.Dialog.setActionActive( action-name, TRUE/FALSE )
• ui.Dialog.setFieldActive( field-name, TRUE/FALSE )

Previous versions allowed you to modify directly the 'active' attribute of the underlying DOM node in the AUI tree. This is now forbidden: it is mandatory to use the methods to enable/disable action or fields. The dialog will synchronize the 'active' attribute in the AUI tree based on the value passed to the methods and depending on the context (some actions or fields can be automatically disabled).

Using HBox tags in forms
HBox tags can be used to stack form items horizontally.

Version 1.30 supports now HBox tags to stack form items horizontally without being influenced by elements above or below.

In an HBox there is a free mix of Form Fields, labels, and Spacer Items possible.

A typical usage of an HBox tag is to have zipcode/city form fields side by side with predictable spacing in-between.

The "classic" layout would look like the following form definition:

```xml
<G "User Data(version 1.20)"
Last Name [l_name]
First Name [f_name]
Street [street]
City [city] Zip Code [zip]
Phone (private) [phone] At work []
Code [aa]-[ab]-[ac]
```
In Figure 7: HBox tag example screenshot on page 202 you will notice that the distance between "l_name" and "First Name" is smaller than between "First Name" and "f_name". How can this be? Two lines under, there is the "zip" field which affects this distance.

If we put HBox tags around the fields we want to group horizontally together, we get the predictable spacing between "l_name","First Name" and "f_name".

```xml
<G "User Data in HBoxes stacked" >
Last Name  [l_nameh  :"First Name":f_nameh   ]
Street     [streeth      :            ]
City       [cityh       :"Zip Code":ziph    ]
Phone(private) [phoneh    :"At work":phonewh  :]
Code       [ba:   "-":bb:"-":bc:              ]
</G>
```

Here "l_nameh","First Name" and "f_nameh" are together in one HBox; the ":" colon acts as a separator between the 3 elements.

The width of an element is calculated from the space between "[" and ":" (width of cityh is 14), or from the space between ":" and ":" (width of "bb" is 2), or from the space between ":" and "]" (width of "f_nameh" is 16). The "zip" field in the version 1.20 example has a width of five and the "ziph" field has also a width of five.

In the second Groupbox in Figure 7: HBox tag example screenshot on page 202 you will notice that the HBox is smaller than the first one, even though it uses two characters more in the screen definition. The reason is that each HBox occupies only ONE cell in the parent grid, and the content in one HBox is independent of the content in another HBox. This relaxes the parent grid; it has to align only the edges of the HBoxes and the labels left of the HBoxes. The two extra characters in the Form file for the second Group come from the fact that the labels need quoting to distinguish them from field definitions. Of course, you could use a Label field if the two extra characters are unwanted (which is done in the third Groupbox).

The third Groupbox shows how the alignment in an HBox can be affected by putting empty elements ( : : ) inside the HBox Tag:

```xml
<G "User Data in HBoxes right part right aligned" >
Last Name  [l_nameh2  : :lfirsth2:f_nameh2   ]
Street     [streeth2     :            ]
City       [cityh2       : :lzip:ziph2    ]
Phone(private) [phoneh2    : :latw:phonewh2  :]
Code       [ca:   "-":cb:        ":" :cc        ]
</G>
```

Between "l_nameh2" and "lfirsth2" there are two ":" signs with a whitespace between them. This means: put a Spacer Item between l_nameh2 and lfirsth2, which gets all the additional space if the HBox is bigger than the sum of l_nameh2, lfirsth2 and f_nameh2. The number of spaces, however, has no effect. The spacer item between cityh2 and lzip has the same force as the spacer between l_nameh2 and lfirsth2.

You can treat a spacer item like a spring. The spacer item between cityh2 and lzip presses cityh2 to the left-hand side, and the rest of the fields to the right-hand side. In the "Code" line there is more than one spacer item; they share the additional space among them. (The "Code" HBox sample in the third line is only to show how spacer items work; we always advise using "Code" as in the second Groupbox, or to use a picture)

In general we advise using the approach shown in the second Groupbox: stack the items horizontally by replacing field ends with ":". This is the easy way to remove unwanted horizontal spacing.

**Figure 7: HBox tag example screenshot**
A big advantage in using elements in an HBox tag is that the fields get their real sizes depending on the definition.

```
LAYOUT
GRID
{
<G g1 >
[a    ] single Edit Field

<G g2 >
MMMMMM
[b    ] The large label expands the Edit Field

<G g3 >
MMMMMM
[c  :] The large label has no influence on the Edit width
}
END
END
ATTRIBUTES
EDIT a = formonly.a, sample="0", default="12345";
EDIT b = formonly.b, sample="0", default="12345";
```
EDIT c = formonly.c, sample="0", default="12345";
END

In the second Groupbox, the edit field is expanded to be as large as the label above; using an HBox prevents this.

![Figure 8: Use of HBox](image)

**Note:** in this example, we use a sample of "0" to display **exactly** five numbers.

**HBox Tags limitations**

HBox tags don't work for fields of Screen Arrays or Tables; you will get a form compiler error. The reason is that the current AUI structure does not allow this. The front-end needs a **Matrix** element directly in a **Grid** or a **ScrollGrid** to perform the necessary positioning calculations for the individual fields.

**Width of ButtonEdit/DateEdit/ComboBox**

When using **BUTTONEDIT/COMBOBOX/DATEEDIT** fields, it is recommended that you account for the width of the widget button in addition to the input area.

The problem with **BUTTONEDIT**, **DATEEDIT** and **COMBOBOX** in versions prior to 1.30 is that a field [b] got the width 3, the same width as an edit field with the same layout.

For example:

```
LAYOUT
GRID
{
  [e ]
  [b ]
}
END
END
ATTRIBUTES
EDIT e=formonly.e;
BUTTONEDIT b=formonly.b;
END
```

In this example, the outer (visual) width of both elements was the same, but the edit portion of "b" was much smaller, because the button did not count at all. (In practice this meant that on average only one and a half characters of "b" was visible). However, you could input 3 characters! This resulted in a **BUTTONEDIT** where only one character was visible and inputting more than one character was possible.
Starting with version 1.30, for the Button, the Form Compiler subtracts two character positions from the width of \texttt{BUTTONEDIT/COMBOBOX/DATEEDIT}. This is possible because now the form compiler differentiates the width of the widget from the width of the entry part.

In fact, there is no visual difference between version 1.20 and 1.30 regarding this example, but in version 1.30 you can only enter one character, which is visually more correct.

In the example the \texttt{BUTTONEDIT} aligns with the Edit; that's why the Edit part of the \texttt{BUTTONEDIT} is usually still a bit bigger than one character (this depends on the button size, but if a button edit is contained by an HBox, it will get the exact size of "width" multiplied by the average character pixel width.

To express the \texttt{BUTTONEDIT/COMBOBOX/DATEEDIT} layout more visually, it is possible to specify:

\begin{verbatim}
[ e  ]
[b- ]
\end{verbatim}

the "]-" sign marks the end of the edit portion and the beginning of the button portion ( edit width ="1", widget width ="3" ).

The two characters are also subtracted for a \texttt{BUTTONEDIT} which is child of an HBox.

\begin{verbatim}
[ b :]
\end{verbatim}

gets also width="1", but no widget width, because the HBox stacks the elements horizontally without needing widget width definition.

The two extra characters are only used to show the real size relations more WYSIWYG, and to have the same calculation as in a field without an HBox parent.

\begin{verbatim}
[e1:e2:e3: ]
[b1  :b2  :b3 ]
\end{verbatim}

shows that three \texttt{BUTTONEDIT} fields are much larger than three \texttt{EDIT} fields with the same width.

You can even write:

\begin{verbatim}
[e1:e2:e3: ]
[b1- :b2- :b3- ]
\end{verbatim}

or:

\begin{verbatim}
[e1:e2:e3: ]
[b1-:b2-:b3-]
\end{verbatim}

to use slim buttons and

\begin{verbatim}
[e1:e2:e3: ]
[b1- :b2- :b3- ]
\end{verbatim}

if one uses large buttons to get the maximum WYSIWYG effect.

Please note that buttons do not grow if two characters "]-" are expanded to three characters "- "; the button always computes its size from the image used, it's just reserves more space in the form to match the real size.

\textbf{Related concepts}

\texttt{Form fields default sample} on page 206
An algorithm is used to compute the field width when no SAMPLE attribute is specified.

**Form fields default sample**

An algorithm is used to compute the field width when no SAMPLE attribute is specified.

Starting with version 1.30, if no SAMPLE attribute is specified in the form files, the client uses an algorithm to compute the field width. In this case, a very pessimistic algorithm is used to compute the field widths: The client assumes a default SAMPLE of "M" for the first six characters and then "0" for the subsequent characters and applies this algorithm to all fields, with some exceptions like DATEEDIT fields.

The default algorithm tends to produce larger forms compared to forms used in BDS V3 and very first versions of Genero. Do not hesitate to modify the SAMPLE attribute in the form file, to make your fields shorter.

If you do not want to touch all your forms, a more tailored automatic solution would be to specify a `ui.form.setDefaultInitializer()` function, to set the SAMPLE depending on the AUI tag. In this example small UPSHIFT fields get a sample of "M"; all other fields get a sample of "0". This will preserve the original width for UPSHIFT fields. However, numeric and normal String fields will get the sample of "0" and make the overall width of the form smaller.

Program:

```plaintext
# this demo program shows how to affect the "sample" attribute in a
# ui.form.setDefaultInitializer function
# the main concern is to set a default sample of "0" and to
# correct the sample attribute for small UPSHIFT fields to "M"
# to be able to display full uppercase letter for fields with a small width

MAIN
    DEFINE three_char_upshift CHAR(3)
    DEFINE three_digit_number Integer
    DEFINE longstring CHAR(100)
    CALL ui.form.setDefaultInitializer("myinit")
    OPEN form f from "sampletest2"
    DISPLAY form f
    INPUT BY NAME three_char_upshift,three_digit_number,longstring
END MAIN

FUNCTION myInit(f)
    DEFINE f ui.Form
    CALL checkSampleRecursive(f.getNode())
END FUNCTION

FUNCTION checkSampleRecursive(node)
    DEFINE node,child om.DomNode
    LET child= node.getFirstChild()
    WHILE child IS NOT NULL
        CALL checkSampleRecursive(child)
        CALL setSample(child)
        LET child=child.getNext()
    END WHILE
END FUNCTION

FUNCTION setSample(node)
    DEFINE node,parent om.DomNode
    LET parent=node.getParent()
    -- only set the "sample" for FormFields in this example
    IF parent.getTagName()<>"FormField" THEN
        RETURN
    END IF
    IF node.getAttribute("shift")="up"
        AND node.getAttribute("width")<=6 THEN
            CALL node.setAttribute("sample","M")
        ELSE
```
CALL node.setAttribute("sample","0")
END IF
DISPLAY "set sample attribute of ",node.getId()," to ",
node.getAttribute("sample"),"
"
END FUNCTION

Form File:

LAYOUT(text="sampletest2")
GRID
{
  <G sampletest
      >
      3 Letter Code: [a ] 3 digit code:[b ] Description:[longstring ]
  <G "What can be seen"
      >
        There is no default sample set in this form, but due to a
        ui.form.setDefaultInitializer function, small UPSHIFT fields
        are adjusted to a sample of "M", all other fields get the sample "0"
  }
END
END
ATTRIBUTES
EDIT a=formonly.three_char_upshift,UPSHIFT,default="MMM";
EDIT b=formonly.three_digit_number,default="123";
EDIT longstring=formonly.longstring,UPSHIFT,
  default="DESCRIPTION OF THE ITEM",SCROLL;
END

Figure 9: Sample usage in form

Related concepts
SAMPLE attribute on page 1389
The SAMPLE attribute defines the text to be used to compute the width of a form field widget.

**Size policy for ComboBoxes**

You can use the SIZEPOLICY attribute for a COMBOBOX.

Starting with version 1.30 you can use the SIZEPOLICY attribute for COMBOBOXes.

COMBOBOX form items had a special behavior in versions prior to 1.30, because they adapted their size to the maximum item of the value list. On one hand, this is very convenient because the programmer doesn't have to find the biggest string in the value list, and to estimate how large it will be on the screen (with proportional fonts the string with the highest number of characters is not automatically the largest string). On the other hand, this behavior often led to an unpredictable layout if the programmer didn't reserve enough space for the COMBOBOX.

The SIZEPOLICY attribute gives better control of the result.

```
<code>
< G "Combo makes edit2 too big" >
[edit1]
[combo  ]
[edit2  ]
...
ATTRIBUTES
EDIT edit1=formonly.edit1;
COMBOBOX combo=formonly.combo,
   ITEMS=((0,"Veeeeeeery Loooooooooooooooong Item"),(1,"hallo")),
   DEFAULT=0;
EDIT edit2=formonly.edit2;
END
</code>
```

**Figure 10: Use of SIZEPOLICY**

In this case, the "combo" field gets very large as does "edit2", because it ends in the same grid column. It will confuse the end user if he can input only eight characters and the field is apparently much bigger. Two possibilities exist to surround this:

Use an HBox to prevent the edit2 from growing, and use HBoxes for all fields which start together with combo and are as large or bigger than combo:

```
<code>
< G "Edit2 in HBox doesn't grow" >
[edit1]
[combo :]
[edit2 :]
...
</code>
```
Use the new `SIZEPOLICY` attribute, and set it to `fixed` to prevent combo from getting bigger than the initial six characters (6+Button):

```
<G "Combo has a fixed size">
...
[combo  ]
[edit2  ]
...
ATTRIBUTES
...
COMBOBOX combo=formonly.combo,
    ITEMS = ((0,"Veeeeeeeery Looooooooooooooooong Item"),(1,"hallo")),
    DEFAULT=0, SIZEPOLICY=FIXED ;
....
```

In this example the `edit2` dictates the maximum size of combo, because even if the `SIZEPOLICY` is `fixed`, the elements are aligned by the Grid.

To prevent this and have exactly six characters (numbers) in the ComboBox, you need to de-couple `combo` from `edit2` by using an `HBox`.

```
<G "Combo has a fixed size,sample 0,in HBox">
...
Combo [combo  ]
Edit2 [edit2  ]
...
COMBOBOX combo=formonly.combo,
    ITEMS = ((0,12345678 Looooooooooooooooong Item),(1,"hallo")),
    DEFAULT=0, SIZEPOLICY=FIXED, SAMPLE="0";
Figure 13: Use of HBox

Now the wanted six numbers are displayed and `combo` does not grow to the size of `edit2`.

**Related concepts**

**SIZEPOLICY attribute** on page 1391
The `SIZEPOLICY` attribute is a sizing directive based on the content of a form item.

**Action defaults at form level**
You can define action defaults in forms.

Starting with version 1.30 it is now possible to define action defaults in forms. In previous versions you had to define a global action default file; this works for defining common global action attributes, but there is a need to define specific action attributes in some forms. A typical zoom window may have search and navigation actions, while data input windows need to define add/delete/update actions instead.

It is now possible to define an action default section in the form file, and you can also load action defaults with `ui.Form.loadActionDefaults` on page 2504.

**Related concepts**

**ACTION DEFAULTS section** on page 1296
The `ACTION DEFAULTS` section defines local action view default attributes for the form elements.

**Compiled string files (.42s)**
The file extension of compiled string files is new `.42s`.

Starting with version 1.30, compiled localized string files use now the file extension `.42s`.

Before version 1.30, the file extension was `.41s`.

See **Localized strings** on page 538.

**BDL 1.20 upgrade guide**

These topics describe product changes you must be aware of when upgrading to version 1.20.

**Important:** This is an incremental upgrade guide that covers only topics related to the Genero BDL version specified in the page title. Check prior upgrade guides if you migrate from an earlier version. Make sure to also read about the new features for this Genero version.

There is no upgrade note with this version.

Corresponding new features page: **BDL 1.20 new features** on page 105.

**BDL 1.10 upgrade guide**

These topics describe product changes you must be aware of when upgrading to version 1.10.

**Important:** This is an incremental upgrade guide that covers only topics related to the Genero BDL version specified in the page title. Check prior upgrade guides if you migrate from an earlier version. Make sure to also read about the new features for this Genero version.

There is no upgrade note with this version.
Planned desupport

Features described in this topic might be desupported in a future release of the product.

**Important:** If you are using one of these features, consider reviewing your code now.

**Deprecated versus desupported**

A deprecated feature is still available in the product, while a desupported feature may no longer exist. For more details see deprecated and desupported terms definitions.

**Deprecated features no longer documented**

- The PIXELWIDTH and PIXELHEIGHT form file attributes (use WIDTH = n PIXELS / HEIGHT = n PIXELS instead).
- The FIELD form item type that specified abstract form fields with attributes defined in the .val schema file.
- The Window.commentPosition style attribute (only supported by the GDC front-end).

**Deprecated features still documented**

- (GDC only) Local Actions.
- The GDC WinCOM, WinDDE and WinMail front call modules.
- The standard.setWebComponentPath front call.
- The fglrun.mmapDisable=true FGLPROFILE entry is deprecated.

**Migrating from IBM® Informix® 4GL to Genero BDL**

Product differences you must be aware of (and plan for) when migrating from IBM® Informix® 4GL to Genero Business Development Language.

**IBM® Informix® 4GL and Genero BDL products**

IBM® Informix® 4GL (I4GL) and Genero Business Development Language (BDL) are distinct development tools. The goal of Genero BDL is to be as compatible as possible with I4GL, and it is very close. The success of Genero BDL depends on the ability to compile and run legacy code with minimum code changes. For text-mode applications, the migration steps are often reduced to recompile-and-run.

Genero BDL extends the I4GL language with advanced features, such as a Graphical User Interface and SQL access to non-Informix® databases. This leads to some differences that you have to handle, but these incompatibilities are minor compared to the added value.

In some rare cases, the Genero BDL team decided to take a different path to implement an I4GL feature, because we considered that the IBM® Informix® 4GL solution was not adaptable. For example, dynamic arrays in I4GL and Genero BDL have different semantics.

This guide will help you identify the differences and find solutions to make the migration from IBM® Informix® 4GL easier.

**IBM® Informix® 4GL reference version**

Several versions of the IBM® Informix® 4GL language have been released. It started in the mid-80s with I4GL version 4.x; then came version 6.x in 1996. I4GL version 7.2 was released in 1998; then versions 7.31, 7.32, and finally the version: 7.50 came out.
There have been several bug fixes and enhancements over the life of I4GL, resulting in releases that slightly differ. Supporting strict compatibility with all versions of I4GL is not possible for Genero BDL.

The Genero BDL compatibility level with IBM® Informix® 4GL is achieved by comparing with the latest version of I4GL, which is version 7.50 at the time of this writing.

**Installation and setup topics**

When migrating from I4GL to Genero BDL, review the differences between the installation and setup of the two products. Reviewing the differences allows you to plan and prepare for a smooth migration.

**Using C extensions**

When migrating to Genero BDL, you must provide your C-Extensions as shared libraries.

With IBM® Informix® 4GL, you can extend the fglgo runtime executable or link your binary programs with c4gl by adding your own C functions.

When migrating to Genero Business Development Language, the C-Extensions must be reviewed in order to provide them as shared libraries. Normally, C extensions modules must be specified in .4gl modules with the IMPORT instruction. To simplify migration, the runtime system loads the userextension shared library (or DLL) automatically, so you can group all your existing C functions in a unique shared library and use it without changing the source code of your programs.

**Related concepts**

C-Extensions on page 2230

With C-Extensions, you can bind your own C libraries in the runtime system, to call C function from the application code.

**Localization support in Genero**

I4GL and Genero BDL use different libraries and environment variables in support of localization.

To support language-specific and country-specific locales, as well as multibyte character sets like BIG5, IBM® Informix® 4GL uses the Informix® GLS library.

For locale support, Genero Business Development Language (BDL) does not use the Informix® GLS library, to be independent from Informix® GLS libraries. Genero uses the standard C library functions for character data handling, based on the POSIX setlocale() function.

I4GL uses the CLIENT_LOCALE environment variable to define the locale for the application. With Genero BDL, you must use the LANG/LC_ALL environment variables to specify the locale of the application. However, CLIENT_LOCALE is still needed to define the locale for the IBM® Informix® database client.

**Related concepts**

Localization on page 512

Localization support allows you to implement programs that follow specific language and cultural rules.

**Database schema extractor**

Before compiling .4gl or .per files, you must extract the database schema with the fgl dbsch tool.

The fgl dbsch tool will produce an .sch file, and optionally .val and .att files. The fgl dbsch tool can extract database schemas from Informix®, as well as from other databases such as Oracle® and SQL Server, but you must be aware of data type conversion rules.

**Related concepts**

Database schema on page 467

Defines database table structures with column type information to be reused in program variable definitions.

**Compiling 4GL to C**

Genero's use of a p-code architecture removes restrictions on which platforms you use to develop your application.

The IBM® Informix® 4GL compilers include a p-code based runtime system called RDS as well as a C-compiled solution, the c4gl compiler. The RDS solution is typically used in a development environment, supporting a debugger,
while the Informix® 4GL C compiler is traditionally used to maximize performance on production sites. However, the C compiled binaries need to be built on the same target platform as the production system.

Genero Business Development Language supports a p-code architecture, which is as fast as the C-compiled version of IBM® Informix® 4GL. Since p-code files are portable, you can develop your application on a platform that is different from the production platform, saving porting procedures and simplifying deployment tasks.

**User interface topics**

When migrating from I4GL to Genero BDL, review the differences between how windows and form content is rendered between the two products. Reviewing the differences allows you to plan and prepare for a smooth migration.

**Smooth migration with traditional UI mode**

IBM® Informix® 4GL (I4GL) and Genero Business Development Language (BDL) handle windows and form content rendering differently.

I4GL is a TUI-style product, designed to write applications for dumb terminals; Genero BDL uses real GUI rendering, with re-sizable windows and proportional fonts.

To simplify migration from TUI-style products, Genero BDL supports the traditional GUI mode.

**Related concepts**

*Graphical mode with Traditional Display* on page 1114

**Refreshing the user interface**

Genero BDL only refreshes the user interface when the runtime waits for user interaction. In certain scenarios, this can result in information displayed in an I4GL application not being displayed when running as a Genero BDL application.

To optimize the display, the Genero runtime system only refreshes the screen when the user gets the control back. In other words, when an interactive instruction (a dialog) waits for a user interaction.

*Important:* In most cases, the user interface refresh behavior of Genero BDL (where the output is delayed until the runtime waits for a user interaction) does not introduce any display problem in legacy programs. However, attention is required when information must be immediately visible while the runtime continues processing code. A typical example is a *"Please wait..."* message, displayed just before a lengthy processing.

**IBM® Informix® 4GL behavior**

IBM® Informix® 4GL (I4GL) only supports a TUI mode, displaying screens on dumb terminal or terminal emulators.

When a program executes an instruction displaying information to the user, IBM® Informix® 4GL refreshes the screen immediately.

For example, when doing successive `DISPLAY ... AT` instructions in a loop, I4GL will show screen changes for each `DISPLAY` instruction:

```
MAIN
  DEFINE i INTEGER
  MENU "Test"
  COMMAND "Start"
    FOR i=1 TO 50000
      DISPLAY i AT 5,5
    END FOR
  COMMAND "Quit"
  EXIT MENU
END MENU
END MAIN
```

**Genero BDL in TUI mode**

The application runs in text (TUI) mode when FGLGUI=0. This section refers to an application running in TUI mode.
With the above code running in TUI mode, the screen will not show the numbers. It will only show the final number, when the MENU gives the control back to the user.

To display all of the numbers to the end user, force the screen refresh with an `ui.Interface.refresh()` API call.

```plaintext
MAIN
  DEFINE i INTEGER
  MENU "Test"
    COMMAND "Start"
      FOR i=1 TO 50000
        DISPLAY i AT 5,5
        CALL ui.Interface.refresh()
      END FOR
    COMMAND "Quit"
    EXIT MENU
  END MENU
END MAIN

**Important:** Forcing the user interface refresh as shown in this example must only be done when using the TUI mode. See GUI mode case for details.

**Genero BDL in GUI mode**

When using the GUI mode of Genero, instead of displaying directly to the terminal screen, the runtime system communicates with a front-end. See The dynamic user interface on page 1105 for more details.

With GUI mode, if the program refreshes the user interface too often, it will produce a lot of unnecessary network traffic. The user interface should be refreshed periodically rather than continuously, to avoid network clogging.

In the following code example, we have added the `(i MOD 100)==0` test to reduce the number of refresh calls.

```plaintext
MAIN
  DEFINE i INTEGER
  MENU "Test"
    COMMAND "Start"
      FOR i=1 TO 5000000
        IF (i MOD 100) == 0 THEN
          DISPLAY i AT 5,5
          CALL ui.Interface.refresh()
        END IF
      END FOR
    COMMAND "Quit"
    EXIT MENU
  END MENU
END MAIN

**Important:** This code example is provided to explain the Genero behavior. Do not use this as a programming pattern! Let the runtime system refresh the user interface automatically when needed, and ONLY use `ui.Interface.refresh()` to update the display of messages like "Please wait..." before a lengthy processing.

**Related concepts**

Refreshing the display when processing on page 1710
This topic explains when to use the `ui.Interface.refresh()` method.

**SCREEN versus LAYOUT section**

When writing new programs for GUI applications, it is recommended that you use a LAYOUT section instead of SCREEN. However, the SCREEN section is still supported to be used to design TUI mode forms.

To design a form with IBM® Informix® 4GL (I4GL), you organize labels and fields in the SCREEN section of a .per form file. Genero Business Development Language introduced a new LAYOUT section to hold form elements. The new LAYOUT section allows for a more sophisticated form design than the SCREEN section.

![Figure 14: Form using a SCREEN section in TUI mode](image)
Figure 15: Form using a LAYOUT section in GUI mode

Related concepts
LAYOUT section on page 1301
The LAYOUT section defines the graphical alignment of the form by using a tree of layout containers.

MENU rendering and behavior
This topic describes differences between I4GL and FGL MENU instructions in TUI mode.

Menu disappears when displaying to its lines
With IBM® Informix® 4GL, it is possible to overwrite the menu options and comment line with DISPLAY text AT line, column: The menu elements remain visible and will be refreshed when the control goes back to the MENU dialog:

```plaintext
MAIN
    MENU "Menu1"
        COMMAND "Option1" "This is option #1"
        COMMAND "Option2" "This is option #2"
        COMMAND "Menu2" "Executes second MENU" CALL menu2()
        COMMAND "Quit" EXIT MENU
    END MENU
END MAIN
FUNCTION menu2()
```
When in TUI mode with Genero BDL, the first MENU will disappear as soon as some text is displayed in one of the menu lines. Genero understands that you want to reuse space used by the first menu, and clears the menu lines to avoid corrupted screens.

**Menu disappears when leaving the MENU instruction**

With IBM® Informix® 4GL, the menu options and comment text remain visible even when the MENU instruction ends. In the next code example, after the first MENU instruction terminates, the menu display line is modified (OPTIONS MENU LINE) and a second MENU instruction is executed. The first MENU is still visible while executing the second MENU:

```
MAIN
  MENU "Menu1"
  COMMAND "Option1"
  COMMAND "Quit" EXIT MENU
END MENU
OPTIONS MENU LINE 10
MENU "Menu2"
  COMMAND "Option1"
  COMMAND "Quit" EXIT MENU
END MENU
END MENU
```

With Genero BDL, the first MENU disappears when the second MENU starts.

**Related concepts**

- [Ring menus (MENU)](page 1467)

  The MENU instruction implements a list of options the end user can choose from.

- **Migrating screen arrays to tables**

  Tables in Genero BDL display using a real table widget, providing a more robust display and interaction than the I4GL screen array.

  With IBM® Informix® 4GL, a list of records can be displayed on the screen by using a static screen array in the SCREEN section of the form specification file, with a finite number of lines:

```
DATABASE stores
SCREEN
{
  Id   First name   Last name
  [f001  f002       f003  ]
  [f001  f002       f003  ]
  [f001  f002       f003  ]
  [f001  f002       f003  ]
  [f001  f002       f003  ]
  [f001  f002       f003  ]
}
END
TABLES
  customer
END
ATTRIBUTES
  f001 = customer.customer_num ;
```
The display of the form specification file in GUI mode:

![Legacy static screen array (Linux/Gnome GDC)](image)

**Figure 16: Legacy static screen array (Linux/Gnome GDC)**

With Genero Business Development Language, use a static screen array for applications displayed in dumb terminals. For GUI applications, use the `TABLE` container:

```4gl
DATABASE stores
LAYOUT (TEXT="Customers")
TABLE
{  
  [Id] [First name] [Last name]
  [f001 f002 f003 ]
  [f001 f002 f003 ]
  [f001 f002 f003 ]
  [f001 f002 f003 ]
  [f001 f002 f003 ]
  [f001 f002 f003 ]
}
END
END
TABLES
  customer
END
ATTRIBUTES
  f001 = customer.customer_num ;
  f002 = customer.fname ;
  f003 = customer.lname ;
END
INSTRUCTIONS
  SCREEN RECORD sr_cust( customer.* );
END
```

The display of the form specification file is a real table widget, which is resizable. The .4gl source is untouched.
Figure 17: Table widget (Linux/Gnome GDC)

Related concepts

**TABLE container** on page 1326
Defines a re-sizable table designed to display a list of records.

**Review TUI-specific features for GUI display**

IBM® Informix® 4GL programs use the TUI mode and often exploit all the display possibilities of the language for dumb terminals. Some instructions are specific to TUI mode and should be reviewed when redesigning the application for GUI mode.

For example, data records can be displayed in a screen array with a `DISPLAY array[array-index].* TO screen-array[screen-line]` instruction, optionally with the `ATTRIBUTES()` clause to use some TTY attributes like colors, reverse and bold effects. When scrolling a list, I4GL actually uses the terminal scrolling capabilities to preserve the TTY attributes in each row. This applies only to the current rows visible on the screen, but it was a commonly used feature.

In order to display application screens on different types of front-ends, Genero Business Development Language (BDL) handles user interface elements in a more abstract way. Therefore, dumb terminal specifics as described above cannot be supported. A good replacement for `DISPLAY ... TO ... ATTRIBUTES()` in `DISPLAY ARRAY` or `INPUT ARRAY` is to use the `DIALOG.setArrayAttributes()` method.

Genero BDL supports TUI-specific instructions such as `DISPLAY AT`, `CLEAR SCREEN`, `CLEAR WINDOW`, as well as TTY attributes such as `BLUE`, `RED`, `REVERSE`, but it is recommended that you use those instructions for TUI programs only. The recommendation for new GUI programs is to use graphical user interface possibilities. For example, a good replacement for TTY attributes is to use presentation styles.

**Related concepts**

**Presentation styles** on page 1165
Use presentation styles to specify decoration attributes for window and form elements.

**The default SCREEN window**

When the first interactive instruction is reached in a Genero BDL program, a default window named `SCREEN` is created.

The default `SCREEN` window can be used to open one or more successive forms; it can also be closed, with the `CLOSE WINDOW SCREEN` instruction. If the default `SCREEN` window is not closed, and a new window is created with the `OPEN WINDOW` command, an empty default `SCREEN` window will be displayed.

When writing a GUI application, you typically open the main form in the `SCREEN` window, and display other forms with the `OPEN WINDOW name WITH FORM` instruction:
The SCREEN window is not visible in TUI mode because program windows are rendered as simple boxes and SCREEN is created without borders. The size of the SCREEN window is 80x25 in TUI mode.

**Related concepts**

OPEN WINDOW on page 1131

Creates and displays a new window.

**Specifying WINDOW position and size**

With Genero BDL in GUI mode, window position and sizes are ignored; in TUI mode, window position and sizes are respected.

When writing a program for TUI mode, the windows can be created with the OPEN WINDOW name AT x, y instruction, specifying an position on the screen; sometimes even the width and height of the window is specified, for example when you don’t use a form to create the window. Window position and size is allowed by Genero Business Development Language for TUI mode applications.

However, the window position and sizes are ignored in GUI mode. In GUI mode, the window position is defined by the window manager, and the size adapts to the form displayed. In this mode, the preferred way to display application forms is to use the OPEN WINDOW name WITH FORM instruction.

**Related concepts**

OPEN WINDOW on page 1131

Creates and displays a new window.

**Right justified field labels**

I4GL forms that specify right-justified labels should be reviewed for update to LABEL form items.

If the application forms define right-justified labels and use a proportional font in GUI mode, the text will no longer be aligned as on a dumb terminal. Form layout must be reviewed to replace any right-justified text with LABEL form items.

Migration to GUI mode can be made easier by using the GUI mode with traditional display, which allows leaving TUI-style I4GL forms untouched.

**Example of right-justified static form labels**

```i4gl
DATABASE FORMONLY
SCREEN
{
  Customer id: [f01       ]
  Name: [f02                          ]
  Zipcode: [f03     ]
  Address: [f04                                   ]
}
END
ATTRIBUTES
EDIT f01 = FORMONLY.cust_id;
EDIT f02 = FORMONLY.cust_name;
EDIT f03 = FORMONLY.cust_zipcode;
EDIT f04 = FORMONLY.cust_address;
END
```
Example of form label items with localized text

LAYOUT
GRID
{
  [101 | f01     ]
  [102 | f02     ]
  [103 | f03 ]
  [104 | f04 ]
}
END
END

ATTRIBUTES
LABEL l01: TEXT=%"customer.id", JUSTIFY=RIGHT;
LABEL l02: TEXT=%"customer.name", JUSTIFY=RIGHT;
LABEL l03: TEXT=%"customer.zipcode", JUSTIFY=RIGHT;
LABEL l04: TEXT=%"customer.address", JUSTIFY=RIGHT;
EDIT f01 = FORMONLY.cust_id;
EDIT f02 = FORMONLY.cust_name;
EDIT f03 = FORMONLY.cust_zipcode;
EDIT f04 = FORMONLY.cust_address;
END

Related concepts
LABEL item definition on page 1341
Defines a simple text area to display a read-only value, in a grid-based layout.

Graphical mode with Traditional Display on page 1114

Reduce multiple text screens
Moving beyond the 80x25 dimensions of a display may require a review of your dumb-terminal-oriented programs.

Applications designed for dumb terminals (TUI mode) use various techniques to ensure that all display fits in an 80x25 screen. This may mean iterating through a number of dialogs using different forms, only displaying certain columns in an record list, using abbreviations for labels, and so on.

With a graphical user interface (GUI mode), windows are wider, re-sizable, and can contain different layout elements and widgets, displaying much more information than in a simple dumb terminal. For example, TABLE containers display record lists and have the ability to scroll horizontally so that you can show more than 78 characters of data.

It is recommended that you review dumb-terminal oriented programs to see how to take advantage of the GUI possibilities. However, avoid ending up with over-crowded screens that may be unreadable to the end user.

Related concepts
TABLE container on page 1326
Defines a re-sizable table designed to display a list of records.

Review application ergonomics
Genero BDL no longer limits programs to executing a single interactive instruction, and provides additional GUI concepts such as drag-and-drop and tree views.

With IBM® Informix® 4GL, programs can only execute a single MENU, INPUT, CONSTRUCT, DISPLAY ARRAY or INPUT ARRAY instruction at a time. This may be sufficient for dumb-terminal applications, but is not adapted for a graphical user interface.

Genero Business Development Language (BDL) introduces the concept of multi-dialog, where multiple interactive instructions control several form areas at the same time. Typical GUI concepts such as Drag and Drop and Tree Views are supported as well. You may wish to review your code to take advantage of these features.
**Subscripted form fields are not supported**

Subscripted form fields must be located and redefined when moving to Genero BDL.

IBM® Informix® 4GL forms can define subscripted fields with multiple field definition entries in the `ATTRIBUTES` section, each defining a piece of the data displayed by the field, as in this example:

```plaintext
DATABASE stores
SCREEN
{  
  1234567890  
  [f01       ]  
  [f02       ]  
}
END

ATTRIBUTES
f01 = customer.cust_name[1,10];
f02 = customer.cust_name[11,20];
END
```

In the `ATTRIBUTES` section, the name of the field is immediately followed by a subscript specification defining the piece of sub-data the screen tag displays and allows to input.

This feature is not supported by Genero BDL, all fields must be defined as a whole.

**Related concepts**

[LAYOUT section on page 1301](#)

The `LAYOUT` section defines the graphical alignment of the form by using a tree of layout containers.

**WORDWRAP field attribute**

Use a `TEXTEDIT` field to replace repeated multi-line input fields.

IBM® Informix® 4GL forms allow to define multi-line input fields by repeating the field tag in the `SCREEN` section on several lines, and by specifying the `WORDWRAP` attribute in the field definition (with `COMPRESS` or `NONCOMPRESS` options):

```plaintext
DATABASE FORMONLY
SCREEN
{  
  Multi-line input field:
  [f01                           ]  
  [f01                           ]  
  [f01                           ]  
  [f01                           ]  
}
END

ATTRIBUTES
f01 = FORMONLY.comment, WORDWRAP COMPRESS;
END
```

With the graphical mode of Genero BDS, consider using a `TEXTEDIT` field instead:

```plaintext
LAYOUT
GRID
{  
  Multi-line input field:
  [f01                           ]  
  [                              ]  
  [                              ]  
  [                              ]  
}
END

ATTRIBUTES
TEXTEDIT f01 = FORMONLY.comment, STRETCH=BOTH;
END
```
Notice that the TEXTEDIT field tags in the layout section do not repeat the item tag name (f01).

**Related concepts**

- **WORDWRAP Attribute** on page 1413
  - The WORDWRAP attribute enables a multiple-line editor in TUI mode.
- **TEXTEDIT item type** on page 1289
  - Defines a multi-line edit field.

**Ignored form definition attributes**

Field attributes inherited from the Informix® SQL PERFORM syntax should be reviewed for necessity and handling.

IBM® Informix® 4GL form specification file allows field attributes that are inherited from the Informix® SQL PERFORM syntax:

```plaintext
DATABASE FORMONLY
SCREEN
{
[f01]
}
END
ATTRIBUTES
f01 = FORMONLY.comment,
    QUERYCLEAR,
    NOUPDATE,
    ZEROFILL,
    RIGHT
    ;
END
```

With Genero BDL, use the `-Wall` option of `fglform` to detect such attributes:

```
$ fglform -Wall form.per
form.per:9:9:9:18:warning:(-8005) Deprecated feature: The QUERYCLEAR attribute is ignored
form.per:10:9:10:16:warning:(-8005) Deprecated feature: The NOUPDATE attribute is ignored
form.per:11:9:11:16:warning:(-8005) Deprecated feature: The ZEROFILL attribute is ignored
form.per:12:9:12:13:warning:(-8005) Deprecated feature: The RIGHT attribute is ignored
```

**Tip:** Replace the RIGHT attribute by `JUSTIFY=RIGHT` Genero BDL attribute.

**Related concepts**

- **Form item attributes** on page 1352
  - The form item attributes reference.
- **fglform** on page 2068
The fglform tool compiles form specification files into XML formatted files used by programs.

**4GL programming topics**

When migrating from I4GL to Genero BDL, review the programming differences between the two products. Reviewing the differences allows you to plan and prepare for a smooth migration.

**Dynamic arrays**

Support for dynamic arrays differs between I4GL and Genero BDL.

Both IBM® Informix® 4GL (I4GL) and Genero Business Development Language (BDL) implement static arrays with a fixed size. Static arrays cannot be extended:

```plaintext
DEFINE arr ARRAY[100] OF RECORD LIKE customer.*
```

I4GL introduced dynamic arrays in version 7.32. Unlike Genero BDL, I4GL requires explicitly to associate memory storage with a dynamic array by using the `ALLOCATE ARRAY` statement, and memory must be freed with `DEALLOCATE ARRAY`. I4GL dynamic arrays can be resized with the `RESIZE ARRAY` statement. I4GL dynamic arrays cannot be used in a interactive instructions such as `DISPLAY ARRAY`.

```plaintext
DEFINE arr DYNAMIC ARRAY OF RECORD LIKE customer.*
ALLOCATE ARRAY arr[10]
RESIZE ARRAY arr[100]
LET arr[50].cust_name = "Smith"
DEALLOCATE ARRAY arr
```

Genero BDL supports dynamic arrays in a slightly different way than I4GL. There are no allocation, resizing, or deallocation instructions, because the memory for element storage is automatically allocated when needed. Furthermore, you can use dynamic arrays with interactive instructions, making a `DISPLAY ARRAY` or `INPUT ARRAY` unlimited.

```plaintext
DEFINE arr DYNAMIC ARRAY OF RECORD LIKE customer.*
LET arr[50].cust_name = "Smith"
DISPLAY ARRAY arr TO sr.*
```

In Genero BDL, the main difference between static arrays and dynamic arrays is the memory usage; when you use dynamic arrays, elements are allocated on demand. With static arrays, memory is allocated for the complete array when the variable is created.

**Important:** The semantics of dynamic arrays is very similar to static arrays, but there are some small differences. Keep in mind that the runtime system automatically allocates a new element for a dynamic array when needed. For example, when a `DISPLAY arr[100].*` is executed with a dynamic array, the element at index 100 is automatically created if does not exist.

**Related concepts**

Arrays on page 418

Arrays (static or dynamic) allow you to handle an ordered collection of elements.

**Debugger command syntax**

While I4GL and Genero BDL both provide a program debugger, the commands used and how it is used can differ. IBM® Informix® 4GL (I4GL) provides a program debugger. Genero Business Development Language provides a program debugger with a different set of commands as I4GL, compatible with the well-known gdb tool. This debugger can be used alone in command line mode, or with a graphical shell compatible with gdb, such as `ddd`:

```plaintext
ddd --debugger "fglrun -d myprog"
```

**Related concepts**

Starting fglrun in debug mode on page 2143
The runtime system can be started in debug mode with the `-d` option.

**Mismatching global variable definitions**

Review your code and use the same data type for all global variables having the same name.

The `c4gl` C-code compiler of IBM® Informix® 4GL has a weakness that allows global variable declarations of the same variable with different data types. Each different declaration found by the `c4gl` compiler defines a distinct global variable, which can be used separately. This can actually be very confusing (the same global variable name can, for example, reference a `DATE` value in module A and an `INTEGER` value in module B).

IBM® Informix® 4GL RDS (fglpc / fglgo) does not allow multiple global variable declaration with different types. The fglgo runner raises error -1337 if this happens.

The next code example shows two .4gl modules defining the same global variable with different data types:

**Main.4gl:**

```plaintext
GLOBALS
    DEFINE v INTEGER
END GLOBALS
...
MAIN
    LET v = 123
    ...
END MAIN
```

**Module.4gl:**

```plaintext
GLOBALS
    DEFINE v DATE
END GLOBALS
...
FUNCTION test ()
    ...
    LET v = TODAY
    ...
END FUNCTION
```

The fglcomp tool compiles both modules separately without problem, but when linking with fglink, the linker raises error -1337.

You must review your code and use the same data type for all global variables having the same name.

**Related concepts**

- **Primitive Data types** on page 289
  Selecting the correct data type assists you in the input, storage, and display of your data.

- **Strict function signature checking**
  With Genero BDL, a function's signature is detected at link time.

IBM® Informix® 4GL (I4GL) is not very strict regarding function signature. With I4GL, you can, for example, define a function in module A that returns three values, and call that function in module B with a returning clause specifying two variables:

**Module A:**

```plaintext
FUNCTION func ()
    RETURN "abc", "def", "ghi"
END FUNCTION
```
Module B (main):

```4gl
MAIN
    DEFINE v1, v2 VARCHAR(100)
    CALL func() RETURNING v1, v2
END MAIN
```

The c4gl compiler (7.32) compiles and links these modules without error, but at execution time you get the following runtime error:

```
Program stopped at "main.4gl", line number 3.
FORMS statement error number -1320.
A function has not returned the correct number of values expected by the calling function.
```

With Genero Business Development Language (BDL), the mistake will be detected at link time:

```
$ fgllink -o prog.42x main.42m module_a.42m
ERROR(-6200): Module 'main': The function module_a.func(0,3) will be called as func(0,2).
```

Similarly, I4GL does not detect an invalid number of parameters passed to a function defined in a different module:

Module A:

```4gl
FUNCTION func(p)
    DEFINE p INTEGER
    DISPLAY p
END FUNCTION
```

Module B (main):

```4gl
MAIN
    CALL func(1,2)
END MAIN
```

The c4gl compiler (7.32) compiles and links these modules without error, but at execution time, you get the following runtime error:

```
Program stopped at "main.4gl", line number 2.
FORMS statement error number -1318.
A parameter count mismatch has occurred between the calling function and the called function.
```

When using Genero BDL, the error will be detected at link time:

```
$ fgllink -o prog.42x main.42m module_a.42m
ERROR(-6200): Module 'main': The function module_a.func(1,0) will be called as func(2,0).
```

However, Genero BDL does not check function signatures when several RETURN instructions are found by the compiler. This is necessary in order to be compatible with I4GL. The next code example compiles and runs with both I4GL and BDL:

```4gl
MAIN
    DEFINE v1, v2 VARCHAR(100)
    CALL func(1) RETURNING v1
    DISPLAY v1
    CALL func(2) RETURNING v1, v2
    DISPLAY v1, v2
```
FUNCTION func( n )  
DEFINE n INTEGER  
IF n == 1 THEN  
   RETURN "abc"  
ELSE  
   RETURN "abc", "def"  
END IF  
END FUNCTION

However, this type of programming is not recommended.

**Related concepts**

Returning values on page 446  
A function can return values with the RETURN instruction.

**STRING versus CHAR/VARCHAR**

Genero BDL supports the STRING data type in addition to CHAR and VARCHAR. While the STRING data type is useful in certain situations, there are times when you should use CHAR or VARCHAR instead.

Genero Business Development Language (BDL) introduces a new data type named STRING, which is similar to VARCHAR but without a size limit. The STRING data type does not exist in IBM® Informix® 4GL. The STRING data type implementation is optimized for memory usage; unlike CHAR/VARCHAR, BDL will only allocate the memory needed to hold the actual character string value in a STRING variable.

A STRING variable is typically used within utility functions (for example, to hold the path to a file). Another typical usage is with CONSTRUCT, to hold the SQL condition. The STRING variable can then be completed to build the SQL text and passed to the PREPARE or DECLARE instruction.

STRING variables can be used in SQL statements. However, it is recommended to use the CHAR/VARCHAR variables that match the size of the corresponding CHAR/VARCHAR database columns.

The STRING data type has a number of built-in methods that are very useful and will reduce source code, such as getLength().

**Related concepts**

STRING on page 305  
The STRING data type is a variable-length, dynamically allocated character string data type, without limitation.

**Review user-made C routines**

Genero BDL provides libraries which may replace some of the C routines required by I4GL applications.

IBM® Informix® 4GL (I4GL) applications often need additional utility C routines implemented in C-Extensions. For example, to access the file system and read the content of a directory with the os.Path class. Writing C-Extensions is an important cost in cross-platform portability and maintenance.

Genero Business Development Language (BDL) provides a set of libraries that include functions and classes which can probably replace some of the routines written for I4GL application. For example, BDL implements typical file management functions to search directories and files.

If portability is a concern (for example if you want to move from a UNIX™ platform to a Microsoft™ Windows™ or Mac OS-X™ platform), review your C routines and check whether there is a replacement built into the language or in one of the libraries provided.

Genero BDL even allows use of the huge Java class library with the Java Interface.

**Web Services support**

Both I4GL and Genero BDL support Web services, albeit with different implementations.

Starting with IBM® Informix® 4GL version 7.50, I4GL functions can be deployed as Web Services. The published functions can be subscribed from programs that run on a Web client in another programming language.
Web Services support was introduced in Genero Business Development Language before I4GL 7.50 was released. Each implementation is quite different, but the basic principles are the same: publishing 4gl functions as Web Services, by handling WS requests and supporting easy input and output parameter conversions between WS data formats and 4gl program variables.

**Related concepts**

*Web services* on page 3270
Create a Web service client or server with Genero BDL.

**File I/O statements and APIs**
Both I4GL and Genero BDL support accessing files on operating systems. The `base.Channel` built-in class supported by Genero BDL can also open streams to subprocesses and sockets.

IBM® Informix® 4GL version 7.50.xC4 introduced file manipulation instructions to access files on the operating system running the application. These instructions can be used to open, read, write, seek and close files:

```
MAIN
    DECLARE fd1, fd2 INTEGER, v1,v2 VARCHAR(10)
    OPEN FILE fd1 FROM "/tmp/file1" OPTIONS (READ, FORMAT="CSV")
    OPEN FILE fd2 FROM "/tmp/file2" OPTIONS (WRITE, APPEND, CREATE, FORMAT="CSV")
    READ FROM fd1 INTO v1, v2
    SEEK ON fd2 TO 0 FROM LAST INTO v1
    WRITE TO fd2 USING v1, v2
    CLOSE FILE fd1
    CLOSE FILE fd2
END MAIN
```

Genero Business Development Language (BDL) implements file I/O support with the `base.Channel` built-in class. This class implements file access, but it can also open streams to subprocesses (i.e. pipes) and sockets.

**Related concepts**

*The Channel class* on page 2412
The `base.Channel` class is a built-in class providing basic input/output functions.

**OPEN USING followed by FOREACH**
With Genero BDL, avoid having an `OPEN ... USING` instruction.

In earlier versions of IBM® Informix® 4GL (I4GL), the FOREACH instruction had no USING clause to pass SQL parameters to the prepared statement. SQL Parameters could be specified in a OPEN USING instruction, and were re-used by the next FOREACH instruction:

```
PREPARE st1 FROM "SELECT * FROM tab WHERE col>?'"
DECLARE cu1 CURSOR FOR st1
OPEN cu1 USING var
FOREACH cu1 INTO rec.*
    DISPLAY rec.*
END FOREACH
```

This feature is supported by Genero Business Development Language, but can lead to defects with some versions of the Informix® database client. Review your code to avoid the OPEN statement by moving the USING clause to the FOREACH instruction.

**Related concepts**

*FOREACH (result set cursor)* on page 724
Processes a series of data rows returned from a database cursor.

**ARG_VAL() returns NULL if no argument**

If the index passed to the function references an argument that does not exist, the ARG_VAL() function must be handled differently between I4GL and Genero BDL.

With IBM® Informix® 4GL (I4GL), if the index passed to the function references an argument that does not exist, the ARG_VAL() function returns a blank character. With Genero BDL, if the index passed to the function references an argument that does not exist, the ARG_VAL() function returns NULL:

```plaintext
$ cat arg_val.4gl
MAIN
  DEFINE v VARCHAR(10)
  LET v = ARG_VAL(1)
  IF v IS NULL THEN
    DISPLAY "IS NULL"
  ELSE
    DISPLAY "IS NOT NULL: [", v, "]"
  END IF
END MAIN
```

With IBM® Informix® 4GL (I4GL), if the index passed to the function references an argument that does not exist, the ARG_VAL() function returns a blank character. With Genero BDL, if the index passed to the function references an argument that does not exist, the ARG_VAL() function returns NULL:

```plaintext
$ fglcomp arg_val.4gl && fglrun arg_val.42m
IS NULL
$ c4gl -o arg_val.bin arg_val.4gl && ./arg_val.bin
IS NOT NULL: [ ]
```

You must review ARG_VAL() conditions using the != operator, and add NVL() to handle NULL arguments:

```plaintext
-- Default is "USER" mode when no argument is passed or when argument
-- is different from "ADMIN"
MAIN
  --IF ARG_VAL(1)!="ADMIN" THEN
  IF NVL(ARG_VAL(1),"?")!="ADMIN" THEN
    DISPLAY "User mode..."
  ELSE
    DISPLAY "Admin mode..."
  END IF
END MAIN
```

Normally the code should use the NUM_ARGS(), to check the number of parameters passed to the program.

**Related concepts**

arg_val() on page 2271

Returns a command line argument by position.

num_args() on page 2273

Returns the number of program arguments.

**Checking function parameters/returns**

With Genero BDL, a parameter count mismatch is detected at link time.

IBM® Informix® 4GL makes function parameter checking at runtime. It is possible to compile a source that calls a function of another module with more (or less) parameters (or return values) as specified in the function definition.

Unlike I4GL, the Genero BDL linker can detect such programming errors when building the program. This is much better as detecting the programming error a runtime.

See the following example:

```plaintext
-- main.4gl
MAIN
  CALL func(1,2)
```
With I4GL, the programming error can only be detected at runtime:

```
$ c4gl -c mod1.4gl
$ c4gl -c main.4gl
$ c4gl -o main.bin main.o mod1.o
$ ./main.bin
Program stopped at "mod1.4gl", line number 2.
FORMS statement error number -1318.
A parameter count mismatch has occurred between the calling
function and the called function.
```

With Genero BDL, the programming error is detected at link time:

```
$ fglcomp mod1.4gl
$ fglcomp main.4gl
$ fgllink -o main.42r main.42m mod1.42m
ERROR(-6200):Module 'main': The function mod1.func(3,0) will be called as
func(2,0).
```

**Tip:** Consider using IMPORT FGL to get rid of program linking with Genero!

**Related concepts**

IMPORT FGL module on page 481
The IMPORT FGL instruction imports module symbols.

Linking programs on page 2115
Describes how to link .42m modules together to build a .42r program file.

**Using variable suscripts in SQL**

Use of subscript operations on host variables in static SQL statements is not supported by Genero BDL.

IBM® Informix® 4GL allows to use subscripts operators on host variables in static SQL statements. For example:

```
DEFINE var VARCHAR(40)
SELECT * FROM tab WHERE col = var[2,3]
```

When doing this, I4GL will silently ignore the [2, 3] operator, to use the full variable value!

Unlike I4GL, the Genero BDL compiler denies this syntax, and is considered as bad practice.

**Related concepts**

Using program variables in static SQL on page 695
Static SQL syntax supports the usage of program variables as SQL parameters.

**Migrating from Four Js BDS to Genero BDL**

Product changes you must be aware of (and plan for) when migrating from Four Js BDS 3.xx to the most recent Genero Business Development Language version.

**Installation and setup topics**

When migrating from Four Js BDS to Genero BDL, review the differences between the installation and setup of the two products. Reviewing the differences allows you to plan and prepare for a smooth migration.

**License controller**

The licensing tool differs between Four Js BDS and Genero BDL.

With Four Js Business Development Suite (BDS), you must license the product with the licencef4gl command line tool. Starting with Genero Business Development Language, the command line tool to license the product is fglWrt. Run fglWrt -h to view the command line options.

**Runner linking is no longer needed**

With Four Js Business Development Suite (BDS), you need to create the fglrun binary with the fgmkrun tool, by specifying the type of the database driver and C extensions libraries to be linked with the runtime system. Since Genero Business Development Language version 2.00, you no longer need to link the runtime system.

The database drivers are provided as shared libraries ready to use; you just need to specify the driver to be loaded.

However, C extensions must be provided shared libraries for Genero BDL. To easy migration, the runtime system loads automatically the userextension share library (or DLL).

**Related concepts**

- Database connections on page 656
Explain how to manage database connections in a program.

- C-Extensions on page 2230
With C-Extensions, you can bind your own C libraries in the runtime system, to call C function from the application code.

**Localization support**

Four Js BDS and Genero BDL use different libraries and environment variables in support of localization.

IBM® Informix® 4GL (I4GL) and Four Js Business Development Suite (BDS) use the Informix® GLS library for localization support (i.e. to support non-ASCII character sets such as BIG5). This implies a strong dependency to the proprietary GLS library.

Genero Business Development Language (BDL) does not use the GLS library; Genero BDL uses the standard C library functions for character set handling, based on the setlocale() POSIX conformant function.

While I4GL/BDS need the CLIENT_LOCALE environment variable to define the locale for the application, you must now use the LANG/LC_ALL environment variables to specify the locale of the Genero application. Note, however, that CLIENT_LOCALE is still needed when connecting to an IBM® Informix® database.

In Four Js BDS, you could select the locale library with the fglmode tool, to select either GLS or ASCII mode. This tool is no longer needed in Genero.

**Related concepts**

- Localization on page 512
Localization support allows you to implement programs that follow specific language and cultural rules.

**Database schema extractor**

Before compiling .4gl or .per files with Four Js Business Development Suite (BDS) or with Genero Business Development Language (BDS), you need to extract the database schema as a .sch file. However, the extraction tools differ.

BDS provides the fglschema tool, while Genero BDL provides the fgldbsch tool. The fglschema tool could only extract schemas from Informix® databases; the fgldbsch can extract database schemas from Informix®, and from other databases like Oracle®, DB, SQL Server, DB2®, PostgreSQL and Oracle® MySQL.

The fglschema tool is still supported in Genero BDL for backward compatibility, but fglschema actually calls fgldbsch.

**Note:** Genero BDL allows you to centralize new widget types and attributes in the .val file.

**Related concepts**

Database schema [on page 467](#)

Defines database table structures with column type information to be reused in program variable definitions.

**C-Code compilation is desupported**

The compiler of Genero Business Development Language does not support C-Code generation.

Four Js Business Development Suite (BDS) fgcomp compiled to both P-Code or C-Code. The compiler of Genero Business Development Language does not support C-Code generation. Only P-Code generation is supported by Genero BDL.

If you experience performance problems when comparing Genero BDL to Four Js BDS, please contact your local support center.

**Desupported environment variables**

The Four Js Business Development Suite (BDS) environment variables no longer supported (or replaced) in Genero Business Development Language.

**Table 107: Desupported environment variables**

<table>
<thead>
<tr>
<th>Entry</th>
<th>Description of the BDS environment variable</th>
<th>Genero equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>FGLDBS</td>
<td>FGLDBS defines the type and version of the database driver, used when linking fglrn with fglmkrun.</td>
<td>Database drivers are loaded dynamically by fglrn.</td>
</tr>
<tr>
<td>FGLCC</td>
<td>FGLCC defines the name of the C compiler.</td>
<td>The fglrn tool does not need to be created, it's fully dynamic.</td>
</tr>
<tr>
<td>FGLLIBSQL</td>
<td>FGLLIBSQL defines the list of database client software libraries to be used to link fglrn with fglmkrun.</td>
<td>Database drivers are loaded dynamically by fglrn.</td>
</tr>
<tr>
<td>FGLLIBSYS</td>
<td>FGLLIBSYS defines the list of system libraries to be used to link fglrn with fglmkrun.</td>
<td>The fglrn tool does not need to be created, it's fully dynamic.</td>
</tr>
<tr>
<td>FGLSHELL</td>
<td>FGLSHELL defined the name of the fglrn program, for example when using tools like fglschema.</td>
<td>The name of the runtime system tool is fglrn and does not need to be changed.</td>
</tr>
</tbody>
</table>
**Desupported FGLPROFILE entries**
Identify the Four Js BDS FGLPROFILE configuration entries that are no longer supported, and the Genero BDL equivalent (where relevant).

Genero Business Development Language comes with redesigned software components and features. Some FGLPROFILE entries have been desupported. This table describes what configurations settings are no longer supported, and point to Genero equivalent features if they exist.

**Table 108: Desupported FOURJS BDS FGLPROFILE entries**

<table>
<thead>
<tr>
<th>Entry</th>
<th>Description of the BDS FGLPROFILE entry</th>
<th>Genero equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>fglrun.cursor.global</td>
<td>Boolean to defines if SQL cursors have global scope or not.</td>
<td>No equivalent option.</td>
</tr>
<tr>
<td>fglrun.ix6</td>
<td>Boolean to control the Informix 4GL version 6 and higher behavior. Default is false (Informix 4GL 4.10 behavior)</td>
<td>No equivalent option: Genero implements Informix 4GL 4.10 behavior.</td>
</tr>
<tr>
<td>fglrun.signalOOB</td>
<td>Defines the Out Of Band signal. When pressing the INTERRUPT key in GUI mode, OOB data is sent to the application over the socket. Some UNIX systems use a specific identifier for OOB signals that you can define with this entry. Default is 0.</td>
<td>No equivalent option.</td>
</tr>
<tr>
<td>fglrun.cmd.winnt</td>
<td>On Windows NT and 2000, this entry defines the program to execute 4GL applications started by RUN WITHOUT WAITING. Default is &quot;cmd /c&quot;.</td>
<td>N/A: Obsolete platforms.</td>
</tr>
<tr>
<td>fglrun.cmd.win95</td>
<td>On Windows 95 and XP, this entry defines the program to execute 4GL applications started by RUN WITHOUT WAITING. Default is &quot;start /m&quot;.</td>
<td>N/A: Obsolete platforms.</td>
</tr>
<tr>
<td>fglrun.interface</td>
<td>Defines the TCL/TK configuration file to be used by the graphical server. The file is read from FGLDIR/etc directory. Default is &quot;fgl2c.res&quot;.</td>
<td>N/A: No more TCL/TK client.</td>
</tr>
<tr>
<td>fglrun.scriptname</td>
<td>Defines the TCL/TK client script file to be used by the graphical server. The file is read from FGLDIR/etc directory. Default is &quot;fgl2c.tcl&quot;.</td>
<td>N/A: No more TCL/TK client.</td>
</tr>
<tr>
<td>fglrun.server.*</td>
<td>Parameters to automatically start the graphical server.</td>
<td>gui.server.autostart</td>
</tr>
<tr>
<td>Menu.style</td>
<td>Defines if the options of MENU instructions must be displayed in a separate control frame or in the MENU LINE.</td>
<td>No equivalent option.</td>
</tr>
</tbody>
</table>
**Related concepts**

**FGLPROFILE: VM configuration** on page 247
Identify the Four Js BDS FGLPROFILE virtual machine configuration entries that are no longer supported, and the Genero BDL equivalent (where relevant).

**FGLPROFILE: GUI configuration** on page 235
Identify the Four Js BDS FGLPROFILE GUI configuration entries that are no longer supported, and the Genero BDL equivalent (where relevant).

**User interface topics**

When migrating from Four Js BDS to Genero BDL, review the differences between how windows and form content is rendered between the two products. Reviewing the differences allows you to plan and prepare for a smooth migration.

**Smooth migration with traditional UI mode**

Four Js BDS and Genero Business Development Language (BDL) handle windows and form content rendering differently.

This topic also concerns IBM® Informix® 4GL migration, see the I4GL Migration page for mode details.

**Front-end compatibility**

With Genero BDL, you must use one of the Genero front-ends.

The WTK, WebFE and JavaFE front-ends are not compatible with the Genero fgldr runtime system. When migrating to Genero Business Development Language (BDL), you must use one of the Genero front-ends.

The UNIX™ version of Genero no longer includes the fg1x11d front-end. You must use the GDC front-end on UNIX™.

**FGLGUI is 1 by default**

The default mode differs between Four Js BDS and Genero BDL.

With Four Js Business Development Suite (BDS), when the FGLGUI environment variable is not set, the application starts in TUI mode (FGLGUI=0).

With Genero Business Development Language (BDL), when the FGLGUI environment variable is not set, the application starts in GUI mode (FGLGUI=1).

Therefore, when migrating from Four Js BDS, it is recommended that you set FGLGUI=0 to run the application in text mode as a first step.

<table>
<thead>
<tr>
<th>Entry</th>
<th>Description of the BDS FGLPROFILE entry</th>
<th>Genero equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>gui.screen.*</td>
<td>Define the layout of the application container window.</td>
<td>No equivalent option.</td>
</tr>
<tr>
<td>gui.workspaceframe.*</td>
<td>Parameters for the work frame (the area where 4GL windows and forms are displayed)</td>
<td>No equivalent option.</td>
</tr>
<tr>
<td>gui.controlframe.*</td>
<td>Parameters for the control frame (right menu)</td>
<td>No equivalent option.</td>
</tr>
<tr>
<td>gui.display.source.*</td>
<td>Display redirections for instructions like ERROR and MESSAGE.</td>
<td>Presentation styles for MESSAGE and ERROR.</td>
</tr>
<tr>
<td>gui.statusbar.*</td>
<td>Status bar configuration and rendering.</td>
<td>Status bar types with Windows style attributes</td>
</tr>
<tr>
<td>gui.toolbar.*</td>
<td>Toolbar configuration and rendering.</td>
<td>Toolbar definitions.</td>
</tr>
</tbody>
</table>

![Table](image-url)
**Related concepts**

FGLGUI on page 275

Defines the user interface mode to be used by the program.

**FGLPROFILE: GUI configuration**

Identify the Four Js BDS FGLPROFILE GUI configuration entries that are no longer supported, and the Genero BDL equivalent (where relevant).

This table shows Four Js Business Development Suite (BDS) FGLPROFILE entries related to GUI configuration which are desupported in Genero Business Development Language.

**Table 109: BDS/WTK FGLPROFILE entries related to GUI configuration which are desupported in Genero**

<table>
<thead>
<tr>
<th>Entry</th>
<th>Description of the BDS feature</th>
<th>Genero equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>fglrun.interface, fglrun.scriptName</td>
<td>These entries defined the TCL configuration and script to be send to the WTK front-end.</td>
<td>There is no equivalent in Genero.</td>
</tr>
<tr>
<td>fglrun.guiProtocol.*</td>
<td>These entries were used to configure the communication protocol with WTK front-end.</td>
<td>In Genero you can control this with gui.protocol.* entries.</td>
</tr>
<tr>
<td>fglrun.error.line.number</td>
<td>This entry was used to define the number of lines to be displayed in the error message line.</td>
<td>You can control the aspect of the error line with the statusBarType Window style attribute.</td>
</tr>
<tr>
<td>gui.useOOB.interrupt</td>
<td>These entries were used to configure or disable Out Of Band signal on the GUI protocol socket to avoid problems on platforms not supporting that feature.</td>
<td>Genero supports interruption event handling with a predefined action name called <strong>interrupt</strong>. You can bind any sort of action view (button in form, toolbar or topmenu item) with this name.</td>
</tr>
<tr>
<td>fglrun.signalOOB</td>
<td>OOB signal was used to send interruption events the program executed is processing.</td>
<td>Interrupt events are sent asynchronously with the new Genero GUI protocol and don't use OOB signals any longer.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See <a href="#">User interruption handling</a> on page 1711 for more details.</td>
</tr>
<tr>
<td>Sleep.minTime</td>
<td>This entry was used to define the number of seconds before the interrupt key button appeared on the screen window when the program is processing.</td>
<td>Genero supports interruption event handling with a predefined action name called <strong>interrupt</strong>. You can bind any sort of action view (button in form, toolbar or topmenu item) with this name.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Interrupt events are sent asynchronously with the new Genero GUI protocol and don't use OOB signals any longer.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See <a href="#">User interruption handling</a> on page 1711 for more details.</td>
</tr>
<tr>
<td>Entry</td>
<td>Description of the BDS feature</td>
<td>Genero equivalent</td>
</tr>
<tr>
<td>-------</td>
<td>---------------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>gui.watch.delay</td>
<td>This entry was used to define the number of seconds before the mouse cursor displays as a wait cursor, when the program is processing.</td>
<td>Genero supports interruption event handling with a predefined action name called <strong>interrupt</strong>. You can bind any sort of action (button in form, toolbar or topmenu item) with this name. Interrupt events are sent asynchronously with the new Genero GUI protocol and don't use OOB signals any longer. See <strong>User interruption handling</strong> on page 1711 for more details.</td>
</tr>
<tr>
<td>gui.bubbleHelp.*</td>
<td>These entries were used to enable and configure tooltips displaying field COMMENT text.</td>
<td>Genero front-ends display bubble-help with field COMMENT text by default.</td>
</tr>
<tr>
<td>gui.controlFrame.scroll.*</td>
<td>These entries were used to show and configure a scrollbar in the action panel displaying ON KEY or COMMAND buttons.</td>
<td>Genero front-ends display control frame scrolling buttons by default when needed. See also <strong>ringMenuScroll</strong> Window style attribute.</td>
</tr>
<tr>
<td>screen.scroll</td>
<td>This entry was used to get scrollbars in the main window when the form was too big for the screen resolution of the workstation.</td>
<td>With Genero, by default, each program window is rendered as a distinct GUI window by the front-end. Window aspect can be controlled with <strong>Window style attributes</strong>.</td>
</tr>
<tr>
<td>gui.screen.size.x</td>
<td>These entries were used to configure the size and position of the main screen window with the WTK front-end.</td>
<td>In Genero, each program window is rendered as a distinct GUI window by the front-end. There is no equivalent for these options. However, you can use the <strong>traditional mode</strong> to render program windows in a single parent screen window and with BDS/WTK.</td>
</tr>
<tr>
<td>gui.screen.size.y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gui.screen.x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gui.screen.y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gui.screen.incrx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gui.screen.incry</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gui.screen.withvm</td>
<td>This entry was used to integrate with the X11 window manager (allowing move and resize actions).</td>
<td>There is no equivalent in Genero.</td>
</tr>
<tr>
<td>gui.preventClose.message</td>
<td>This entry was used to display an error message to the user attempting to close the main GUI window with CTRL-F4 or the cross-button on the right of the GUI window title bar.</td>
<td>In Genero, each program window is rendered as a distinct GUI window by the front-end. You can use the <strong>close</strong> action to control window close events. See <strong>Implementing the close action</strong> on page 1772 for more details. See also <strong>ON CLOSE APPLICATION</strong> program option.</td>
</tr>
<tr>
<td>Entry</td>
<td>Description of the BDS feature</td>
<td>Genero equivalent</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-----------------------------------------------------------------------------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>gui.key.doubleClick.left</td>
<td>This entry could be used to define the key to be returned to the program when the user double-clicks on the left button of the mouse.</td>
<td>You can use the DOUBLECLICK attribute to define the action to be invoked when the user double-clicks on a Table container.</td>
</tr>
<tr>
<td>gui.key.click.right</td>
<td>This entry could be used to define the key to be returned to the program when the user clicks on the right button of the mouse.</td>
<td>You can configure context menus with the CONTEXTMENU attribute in action attributes.</td>
</tr>
<tr>
<td>gui.key.add_function</td>
<td>This was used to define the offset to identify SHIFT+Fx keys.</td>
<td>There is no equivalent in Genero.</td>
</tr>
<tr>
<td>gui.key.x.translate</td>
<td>These entries were used to map keys. For example, when the user pressed Control-U, it was mapped to F5 for the program.</td>
<td>There is no equivalent in Genero.</td>
</tr>
<tr>
<td>gui.key.radiocheck.invokeexit</td>
<td>This entry defined the key to select the RADIO or CHECK field and move to the next field.</td>
<td>There is no equivalent in Genero.</td>
</tr>
<tr>
<td>gui.mswindow.button</td>
<td>This entry defined the aspect of buttons on Windows® platforms.</td>
<td>There is no equivalent in Genero: Front-ends will use the current platform theme when possible.</td>
</tr>
<tr>
<td>gui.mswindow.scrollbar</td>
<td>This entry got MS Windows® scrollbar style.</td>
<td>There is no equivalent in Genero: Front-ends will use the current platform theme when possible.</td>
</tr>
<tr>
<td>gui.scrollbar.expandwindow</td>
<td>When set to true, the WTK front-end expanded the window automatically if scrollbars are needed in a screen array.</td>
<td>There is no equivalent in Genero.</td>
</tr>
<tr>
<td>gui.fieldButton.style</td>
<td>This defined the style of BMP field buttons.</td>
<td>There is no equivalent in Genero.</td>
</tr>
<tr>
<td>gui.BMPbutton.style</td>
<td>This defined the style of FIELD_BMP field buttons.</td>
<td>There is no equivalent in Genero.</td>
</tr>
<tr>
<td>gui.entry.style</td>
<td>This entry defined the underlying widgets to be used to manage form fields.</td>
<td>There is no equivalent in Genero.</td>
</tr>
<tr>
<td>gui.user.font.choice</td>
<td>This entry is set to true to allow the end user change the font of the application screen window.</td>
<td>Genero front-ends allow the user to change the font. See front-end specific documentation for option configuration.</td>
</tr>
<tr>
<td>gui.interaction.</td>
<td>This entry was used to highlight the current row during an INPUT ARRAY.</td>
<td>The current row highlighting can be controlled in Genero with the Table style attribute highlightCurrentRow.</td>
</tr>
<tr>
<td>gui.form.foldertab.multiline</td>
<td>These entries configure folder tabs and define the keys to be sent when a page is selected by the user.</td>
<td>Genero supports folder tabs with the FOLDER container in LAYOUT. An action can be defined for each folder PAGE.</td>
</tr>
<tr>
<td>Entry</td>
<td>Description of the BDS feature</td>
<td>Genero equivalent</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>gui.keyButton.position</td>
<td>These entries define the aspect of action panel buttons associated to ON KEY actions in dialogs like INPUT.</td>
<td>Default action views aspect and position can be controlled with Action Defaults attributes and with Window style attributes.</td>
</tr>
<tr>
<td>gui.keyButton.style</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gui.button.width</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Menu.style</td>
<td>These entries define the aspect of action panel buttons associated to COMMAND [KEY] actions in MENU.</td>
<td>Default action views aspect and position can be controlled with action attributes with Window style attributes.</td>
</tr>
<tr>
<td>gui.menu.timer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gui.menu.horizontal.*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gui.menu.showPagerArrows</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gui.menuButton.position</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gui.menuButton.style</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gui.empty.button.visible</td>
<td>This entry is used to hide action panel buttons without text. By default, the empty buttons are visible but disabled.</td>
<td>Default action views aspect can be controlled with action attributes. Use for example the defaultView attribute to display a default button for an action.</td>
</tr>
<tr>
<td>gui.containerType</td>
<td>These entries configure the WCI windows in BDS.</td>
<td>To define WCI containers and children in Genero, use the ui.Interface methods.</td>
</tr>
<tr>
<td>gui.containerName</td>
<td></td>
<td>See Window containers (WCI) on page 2023 for more details.</td>
</tr>
<tr>
<td>gui.mdi.*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gui.toolBar.*</td>
<td>These entries define the toolbar aspect in BDS.</td>
<td>Toolbar definition has been extended in Genero.</td>
</tr>
<tr>
<td>gui.statusBar.*</td>
<td>These entries define the status aspect in BDS.</td>
<td>The StatusBars are defined with Window presentation style attributes.</td>
</tr>
<tr>
<td>gui.directory.images</td>
<td>This entry defines the path to the directories where images (toolbar icons) are located, on the front-end workstation.</td>
<td>See Presentation Styles for more details.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See ToolBars for more details.</td>
</tr>
</tbody>
</table>
### Entry | Description of the BDS feature | Genero equivalent
---|---|---
`gui.display.<source>` | These entries redirect the ERROR / MESSAGE / COMMENT text to a specific place on the GUI screen. | The rendering of ERROR, MESSAGE or COMMENT can be configured with Window style attributes in Genero. However, it is not possible to customize keyboard NumLock / CapsLock status in Genero.  
See [Presentation Styles](#) for more details.  

`gui.local.edit` | These entry enables and configures cut/copy/paste local keys in WTK. | Cut/Copy/Paste are defined as front-end local actions in Genero. You can bind action views with `editcut`, `editcopy`, `editpaste` predefined action names.  
See [Dialog actions](#) on page 1739 for more details.  

`gui.local.edit.error` | |  

`gui.key.cut` | |  

`gui.key.copy` | |  

`gui.key.paste` | |  

`gui.key.*` | These entries were used to map physical key to a virtual key used in programs.  
For example:  
`gui.key.interrupt = "control-c"` | Cut/Copy/Paste are defined as front-end local actions in Genero. You can bind action views with `editcut`, `editcopy`, `editpaste` predefined action names.  
See [Dialog actions](#) on page 1739 for more details.  

`gui.workSpaceFrame.nolist` | This entry defines the aspect of fixed size screen arrays in forms, to render each array cell as an individual edit field. | There is no equivalent in Genero.  

### Related concepts
- [The FGLPROFILE file(s)](##) on page 255  
FGLPROFILE environment variable defines Genero BDL configuration files  

### Key labels versus action defaults
Genero BDL introduces the `ON ACTION` block to define actions. While the `ON KEY` block is still supported, it comes with limitations.

In Four Js Business Development Suite (BDS), labels can be defined for keys such as `accept`, `F10` or `Control-Z`. With this feature, it is possible to easily decorate `ON KEY` or `COMMAND KEY` blocks with a button in the action panel.

Key labels can be specified at different levels:
- at a global level with FGLPROFILE settings,  
- at the program level with the `fgl_setkeylabel()` function,  
- at the form level with the `KEYS` section,  
- at dialog level with the `fgl_dialog_setkeylabel()` function, and  
- at the field level with the `KEY="label"` attribute.  

For more details, see [Setting action key labels](#) on page 1775.
With Genero Business Development Language (BDL), interaction statements can define actions with the ON ACTION blocks. These action handlers are more abstract than ON KEY: You identify an action by a name, while decoration is defined in form files (ACTION DEFAULTS section) or in global configuration files (.4ad files).

When adapting your code for Genero, you are free to use the traditional ON KEY blocks or the new ON ACTION blocks. Genero still supports the key label settings as in Four Js BDS. However, key label settings will overwrite action defaults settings. Additionally, if the name of the key specified in the ON KEY clause does not only contain alphanumeric characters (such as Control-Z), it will not be possible to define action defaults attributes for these action handlers, as action names must be simple identifiers. This is also true for Menu COMMAND labels, for example with COMMAND "Exit program".

**Related concepts**

- **Binding action views to action handlers** on page 1762
  How are action views of the forms bound to action handlers in the program code?

- **ACTION DEFAULTS section** on page 1296
  The ACTION DEFAULTS section defines local action view default attributes for the form elements.

- **User interface basics** on page 1105
  This section introduces to the foundation of the Genero user interface.

### Migrating form field WIDGET="type"

Genero BDL supports the now-deprecated BDS fields using the WIDGET attribute, but recommends the use of Genero BDL form item types instead.

To get combo-boxes or check-boxes in Four Js Business Development Suite (BDS), .per forms defined fields with the WIDGET attribute. To ease migration, the WIDGET attribute and the corresponding form field widgets are still supported in Genero Business Development Language (BDL), but are now deprecated. It is recommended to use the new BDL form item types instead.

<table>
<thead>
<tr>
<th>Checkbox Y N</th>
<th>Confirm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checkbox class key F11 F12</td>
<td>Confirm</td>
</tr>
<tr>
<td>Combo box</td>
<td>Paris</td>
</tr>
<tr>
<td>BMP Field smiley F11</td>
<td>👍</td>
</tr>
<tr>
<td>Radio group A B C</td>
<td>Letter A</td>
</tr>
<tr>
<td></td>
<td>Letter B</td>
</tr>
<tr>
<td></td>
<td>Letter C</td>
</tr>
<tr>
<td>Radio group class key F11</td>
<td>F11 key</td>
</tr>
<tr>
<td></td>
<td>F12 key</td>
</tr>
<tr>
<td></td>
<td>F13 key</td>
</tr>
<tr>
<td>Label field</td>
<td>label 1</td>
</tr>
<tr>
<td>Button field</td>
<td>Print</td>
</tr>
<tr>
<td>Bitmap field</td>
<td>F11</td>
</tr>
</tbody>
</table>

**Figure 18: Four Js BDS-specific widgets**

This table lists the existing Four Js BDS WIDGET fields and the corresponding Genero BDL form item types.
### Table 110: Genero form item types corresponding to old BDS WIDGET fields

<table>
<thead>
<tr>
<th>WIDGET=</th>
<th>Description</th>
<th>Genero equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>WIDGET= &quot;Canvas&quot;</td>
<td>Drawing area for fgldraw functions</td>
<td>CANVAS item type</td>
</tr>
<tr>
<td>WIDGET= &quot;BUTTON&quot;</td>
<td>Text push button firing key event</td>
<td>BUTTON item type</td>
</tr>
<tr>
<td>WIDGET= &quot;BMP&quot;</td>
<td>Image push button firing key event</td>
<td>BUTTON item type</td>
</tr>
<tr>
<td>WIDGET= &quot;CHECK&quot;</td>
<td>Checkbox field</td>
<td>CHECKBOX item type</td>
</tr>
<tr>
<td>WIDGET= &quot;CHECK&quot; + CLASS= &quot;KEY&quot;</td>
<td>Checkbox field firing key event</td>
<td>CHECKBOX item type + ON CHANGE trigger in program</td>
</tr>
<tr>
<td>WIDGET= &quot;COMBO&quot;</td>
<td>Combobox field</td>
<td>COMBOBOX item type</td>
</tr>
<tr>
<td>WIDGET= &quot;FIELD_BMP&quot;</td>
<td>Edit field with push button</td>
<td>BUTTONEDIT item type</td>
</tr>
<tr>
<td>WIDGET= &quot;LABEL&quot;</td>
<td>Label field (no input)</td>
<td>LABEL item type</td>
</tr>
<tr>
<td>WIDGET= &quot;RADIO&quot;</td>
<td>Radio group field</td>
<td>RADIOGROUP item type</td>
</tr>
<tr>
<td>WIDGET= &quot;RADIO&quot; + CLASS= &quot;KEY&quot;</td>
<td>Radio group field firing key event</td>
<td>RADIOGROUP item type + ON CHANGE trigger in program</td>
</tr>
</tbody>
</table>

Genero introduced more form item types, such as **DATEEDIT** and **PROGRESSBAR**.

**Figure 19: New form types in Genero**

**Related concepts**

*Form specification files* on page 1237
Form specification files are the source files defining the layout and content of application forms.

**SCREEN versus LAYOUT section**
When writing new programs for GUI applications, it is recommended that you use a LAYOUT section instead of SCREEN. However, the SCREEN section is still supported to be used to design TUI mode forms.

This topic also concerns IBM® Informix® 4GL migration, see the I4GL Migration page for mode details.

**Migrating screen arrays to tables**
A TABLE container in Genero BDL displays using a real table widget, providing a more robust display and interaction than a screen array, while the SCROLLGRID container renders a list of records in separated field cells, providing a replacement for screen arrays using the OPTIONS="-nolist"

This topic also concerns IBM® Informix® 4GL migration, see the I4GL Migration page for mode details.

By default with Four Js Business Development Suite (BDS), fields of a screen array were rendered as listviews:

![Listview Rendering for Screen Arrays](image)

**Figure 20: Four Js BDS listview rendering for screen arrays**

To split such listview into individual field cells, Four Js BDS provided the OPTIONS="-nolist" attribute, to be specified for each field of the screen array:

```plaintext
f01 = FORMONLY.cust_fname, OPTIONS="-nolist";
f02 = FORMONLY.cust_lname, OPTIONS="-nolist";
```
Figure 21: Four Js BDS screen array with fields using OPTIONS="-nolist"

With Genero BDL, consider using a SCROLLGRID container, to render a list of records in separated field cells. The SCROLLGRID container allows to position field tags on several grid lines, by repeating the same layout for each row:
Review TUI specific features
Some programs use the TUI mode and often exploit all the display possibilities of the language for dumb terminals. These programs should be reviewed when redesigning the application for GUI mode.

This topic also concerns IBM® Informix® 4GL migration, see the I4GL Migration page for mode details.

The default SCREEN window
When the first interactive instruction is reached in a Genero BDL program, a default window named SCREEN is created.

This topic also concerns IBM® Informix® 4GL migration, see the I4GL Migration page for mode details.

Specifying WINDOW position and size
With Genero BDL in GUI mode, window position and sizes are ignored; in TUI mode, window position and sizes are respected.

This topic also concerns IBM® Informix® 4GL migration, see the I4GL Migration page for mode details.

Front-end configuration tools
With Genero BDL, use presentation styles instead of front-end specific configuration tools to define widget aspects such as color, borders, fonts, and more.

Four Js Business Development Suite (BDS) provided WTK front-end and X11 front-end specific configuration tools called "Configuration Manager" / confdesi. These tools define widget aspect (color, borders, fonts).
In Genero Business Development Language, the form items can be decorated with presentation styles for all sorts of front-ends.

**Related concepts**

- Presentation styles on page 1165

Use presentation styles to specify decoration attributes for window and form elements.

**Function key mapping**

When migrating to Genero BDL, special care must be taken for programs that uses function keys greater than F12.

With Four Js Business Development Suite (BDS), when the user pressed a key modifier plus a function key (like Shift-F4 or Ctrl-F6), the key combination was mapped to a regular function key F(nn+offset), because Shift and Control key modifiers are not handled in the 4GL language.

The number of function keys of the keyboard was defined by the gui.key.add_function FGLPROFILE entry. For example, when this entry is set to 12 (the default), a Shift-F4 was received as F16 (4 + 12) in the program, to be handled with the **ON KEY (F16)** clause.

This feature and FGLPROFILE entry is still supported when using the traditional mode.

When using the standard GUI mode, special consideration needs to be taken regarding function keys above F12.

Function keys from F1 to F12 are common keys found on the keyboard and do not need any particular configuration regarding accelerators: The **ON KEY** clause defines an action object identified by the key name (in lowercase) and the first accelerator attribute defined with the same name. For example, **ON KEY (F10)** creates an action f10, with accelerator F10.

However, if the program uses **ON KEY (Fnn)** clauses where nn is above 12, in order to have Shift-F(nn-12) key combinations working, you need to define this accelerator in the corresponding action default entry.

**Important:** Since the **ON KEY (Fnn)** clause defines an automatic shortcut key with the first accelerator attribute, it overwrites the value of acceleratorName attribute defined in the action defaults. In order to associate the actual Shift-F(nn-12) key combination for the action fnn, you must use the second accelerator attribute. For example: **ON KEY (F14)** creates an action f14, with first accelerator F14. Specify acceleratorName2="Shift-F2" in the .4ad action defaults file, to define the Shift-F2 key combination for the f14 action.

**Related concepts**

- Configuring actions on page 1744

Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with action attributes.

**Activating form items with DISPLAY**

The methods for enabling or disabling fields and actions differs between Four Js BDS and Genero BDL.

With Four Js Business Development Suite (BDS), the program could enable and disable WIDGET="BUTTON", "CHECK" and "RADIO" fields of class "KEY" with the DISPLAY TO instruction:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISPLAY &quot;!&quot; TO order.ord_status</td>
<td>Enables the field</td>
</tr>
<tr>
<td>DISPLAY &quot;*&quot; TO order.ord_status</td>
<td>Disables the field</td>
</tr>
</tbody>
</table>

With Genero BDL, use the DIALOG.setFieldActive() method to enable/disable fields, and the DIALOG.setActionActive() method to enable/disable actions (controlling form buttons). For more details, see Enabling and disabling actions on page 1763

**Related concepts**

- Migrating form field WIDGET="type" on page 240
Genero BDL supports the now-deprecated BDS fields using the WIDGET attribute, but recommends the use of Genero BDL form item types instead.

**Defining keys for WIDGET fields**
The method for binding a keyboard function or control key differ between Four Js BDS and Genero BDL.

With Four Js Business Development Suite (BDS), a form field using the WIDGET attribute as "BUTTON" or as "CHECK" or "RADIO" with CLASS="KEY", could specify the key to be sent to the program with the CONFIG attribute:

```plaintext
DATABASE FORMONLY
SCREEN
{
 [f01] [f02] [f03]
}
END
ATTRIBUTES
f01 = FORMONLY.b_print, WIDGET="BUTTON", CONFIG="F10";
f02 = FORMONLY.b_ok, WIDGET="BUTTON", CONFIG="KEY_accept";
f03 = FORMONLY.b_cancel, WIDGET="BUTTON", CONFIG="KEY_interrupt";
END
```

The key specification could be a function key like F10, a control key like Control-P, or a virtual key like KEY_accept (by default, ESC) or KEY_interrupt (by default, Control-C).

With Genero BDL, use the ACCELERATOR action attribute to bind a keyboard function or control key to a named action that is controlled in the program with an ON ACTION action-name handler.

**Related concepts**
Defining keyboard accelerators for actions on page 1749

**BEFORE DISPLAY / BEFORE ROW execution order**
Genero BDL scenarios define whether the BEFORE DISPLAY or BEFORE ROW block is executed first.

When starting a DISPLAY ARRAY dialog with Four Js Business Development Suite (BDS), the BEFORE ROW control block is executed first, then BEFORE DISPLAY is executed.

This is not logical: The BEFORE DISPLAY block is typically used to do some initializations. Therefore, it should be executed before any other dialog trigger.

With Genero BDL, when a singular DISPLAY ARRAY dialog starts, the BEFORE DISPLAY block is executed first, then the BEFORE ROW block is executed. In a DISPLAY ARRAY subdialog of a DIALOG instruction, the BEFORE DISPLAY is executed when the list gets the focus, then BEFORE ROW is executed.

**Related concepts**
Record list (DISPLAY ARRAY) on page 1501
The DISPLAY ARRAY instruction provides record list navigation in an application form, with optional record modification actions.

Multiple dialogs (DIALOG - inside functions) on page 1585
The procedural DIALOG instruction allows for the combination of record list, record input, and query criteria input in the same application form.

### 4GL Programming topics

When migrating from Four Js BDS to Genero BDL, review the programming differences between the two products. Reviewing the differences allows you to plan and prepare for a smooth migration.

### FGLPROFILE: VM configuration

Identify the Four Js BDS FGLPROFILE virtual machine configuration entries that are no longer supported, and the Genero BDL equivalent (where relevant).

Genero Business Development Language (BDL) comes with redesigned software components and features. Some Four Js Business Development Suite (BDS) specific FGLPROFILE entries have been desupported. This section describes what configurations settings are no longer supported, and point to Genero equivalent features if they exist.

This table shows BDS FGLPROFILE entries related to runtime system configuration which are desupported in Genero. See the FGLPROFILE description page for supported entries:

#### Table 111: BDS FGLPROFILE entries related to runtime system configuration which are desupported in Genero

<table>
<thead>
<tr>
<th>Entry</th>
<th>Description of the BDS feature</th>
<th>Genero BDL equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>fglrun.checkDecimalPrecision</td>
<td>Controls decimal variable assignment when overflow occurs. For example, a value of 1000.0 does not fit in a DECIMAL(2,0). Is false by default = no overflow error, value assigned.</td>
<td>There is no equivalent in Genero. By default Genero assigns NULL to a decimal when overflow occurs. Can be trapped by WHENEVER ANY ERROR.</td>
</tr>
<tr>
<td>fglrun.ix6</td>
<td>Controls Informix® version 6.x compatibility. By default BDS is compatible with I4GL 4.x</td>
<td>There is no equivalent in Genero. By default Genero is compatible to Informix® 4gl 7.32.</td>
</tr>
<tr>
<td>fglrun.cmd.winnt, fglrun.cmd.win95</td>
<td>Defines the command line to be executed for a RUN WITHOUT WAITING on Windows® platforms.</td>
<td>With Genero the command program can be defined with the COMSPEC environment variable.</td>
</tr>
<tr>
<td>fglrun.database.listvar, fglrun.remote.envvar</td>
<td>These entries were used by Informix® driver to set environment variables with the ifx_putenv() function on Windows® platforms.</td>
<td>There is no equivalent in Genero.</td>
</tr>
<tr>
<td>fglrun.setenv.<em>, fglrun.defaultenv.</em></td>
<td>These entries define environment variables for all programs.</td>
<td>There is no equivalent in Genero.</td>
</tr>
<tr>
<td>fgllic.*</td>
<td>License controller related entries</td>
<td>With Genero you configure license settings with the flm.* entries. See license manager documentation for more details.</td>
</tr>
<tr>
<td>Entry</td>
<td>Description of the BDS feature</td>
<td>Genero BDL equivalent</td>
</tr>
<tr>
<td>----------------------------</td>
<td>------------------------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>fglrun.server.*</td>
<td>These entries define X11 front-end automatic startup.</td>
<td>In Genero this can be configured with gui.server.autostart.* entries.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See Automatic front-end startup on page 1121 for more details.</td>
</tr>
</tbody>
</table>

**Related concepts**
The FGLPROFILE file(s) on page 255
FGLPROFILE environment variable defines Genero BDL configuration files

**The fgl_init4js() function**
The fgl_init4js() function has no effect in Genero BDL.

Four Js Business Development Suite (BDS) provided a few utility functions in the libfgl4js.42x library. This library had to be initialized with a call to fgl_init4js():

```c
MAIN
    ...
    CALL fgl_init4js()
    ...
END MAIN
```

Genero Business Development Language still supports the fgl_init4js() function, but only for backward compatibility. Calling this function has no effect in Genero.

**Static versus Dynamic Arrays**
Support for dynamic arrays in Genero BDL may result in a need for some redesigns in your application.

This topic also concerns IBM® Informix® 4GL migration, see the I4GL Migration page for more details.

**Debugger syntax changed**
While Four Js BDS and Genero BDL both provide a program debugger, the commands used and how it is used can differ.

This topic also concerns IBM® Informix® 4GL migration, see the I4GL Migration page for more details.

**fgl_system() function**
With Genero BDL, the fgl_system() function no longer raises a terminal window by default, but some front-ends offer a workaround.

The fgl_system() function is still supported in Genero Business Development Language, but it does not raise a terminal window on the front-end as with Four Js Business Development Suite (BDS). However, some front-ends implement a workaround for this feature, based on the detection of special strings displayed to stdout by fglrun. See the front-end documentation for more details.

**Related concepts**
fgl_system() on page 2296
Runs a command on the application server.

**The Channel:: methods**
Review your code and replace Channel:: calls with the new base.Channel API.

Four Js Business Development Suite (BDS) provides the Channel:: functions to do file I/O.

Genero Business Development Language provides file, socket and process I/O with the base.Channel built-in class.
Review your code and replace `Channel::` calls with the new `base.Channel` API.

**Related concepts**
- The Channel class on page 2412
  The `base.Channel` class is a built-in class providing basic input/output functions.

**The Dialog:: methods**
Review your code and replace `Dialog::` calls with the new `ui.Dialog` API.

Four Js Business Development Suite (BDS) provides the `Dialog::` functions to control interactive instructions.
Genero Business Development Language provides the `ui.Dialog` built-in class.
Review your code and replace `Dialog::` calls with the new `ui.Dialog` API.

**Related concepts**
- The Dialog class on page 2512
  The `ui.Dialog` class provides a set of methods to configure, query and control the current interactive instruction.

**STRING versus CHAR/VARCHAR**
Genero BDL supports the `STRING` data type in addition to `CHAR` and `VARCHAR`. While the `STRING` data type is useful in certain situations, there are times when you should use `CHAR` or `VARCHAR` instead.

This topic also concerns IBM® ® Informix® 4GL migration, see the I4GL Migration page for more details.

**Review user-made C routines**
Genero BDL provides libraries which may replace some of the C routines required by I4GL applications.

This topic also concerns IBM® Informix® 4GL migration, see the I4GL Migration page for more details.

**Variable identification in SQL statements**
Program variable identification in static SQL statements is more strict in version 2.20 than older versions.

This topic applies also to older Genero Business Development versions, see the Genero 2.20 Migration page for more details.

**Default action of WHENEVER ANY ERROR**
By default, the WHENEVER ANY ERROR action is to CONTINUE the program flow.

With old Four Js Business Development Suite (BDS) versions like 2.10, expression evaluation errors such as a division by zero stop the program with an error message. Genero Business Development Language behaves like IBM® Informix® 4GL and recent BDS versions like 3.55: By default, the WHENEVER ANY ERROR action is to CONTINUE the program flow. You can change this behavior by setting the next FGLPROFILE entry to true:

```plaintext
fglrn.run.mapAnyErrorToError = true
```

**Related concepts**
- Exceptions on page 502
  Describes exception (error) handling in the programs.
- The FGLPROFILE file(s) on page 255
  FGLPROFILE environment variable defines Genero BDL configuration files

**Database driver features**
Some ODI driver features are no longer supported in Genero BDL.

The following ODI driver features are no longer supported in Genero:

- Static SQL cache
- SQL directive sets
Frequently Asked Questions

The FAQ lists those questions frequently asked when migrating an existing Informix® 4GL application to Genero.

FAQ001: Why do I have a different display with Genero than with BDS V3?

Explanation

Genero Business Development Language (BDL) introduces major graphical user interface enhancements that sometimes require code modification. With BDS V3, application windows created with the OPEN WINDOW instruction were displayed as static boxes in the main graphical window. In the GUI mode of Genero, application windows are displayed as independent, re-sizeable graphical windows.

Solution:

Review the program logic to reduce the number of windows created by the programs. Replace MENU created in specific windows by TOPMENU elements in your forms.

FAQ002: Why does an empty window always appear?

Description

An additional empty window appears when I explicitly create a window with the OPEN WINDOW instruction.

```
MAIN
  OPEN WINDOW w1 AT 1,1 WITH FORM "form1"
  MENU "Example"
    COMMAND "Exit"
    EXIT MENU
  END MENU
CLOSE WINDOW w1
END MAIN
```

Explanation

In the GUI mode of Genero, all windows are displayed as real front-end windows, including the default SCREEN window. When an application starts, the runtime system creates this default SCREEN window. This is required because some applications use the SCREEN window to display forms with OPEN FORM + DISPLAY FORM (they do not use the OPEN WINDOW instruction to create new windows). Therefore, the runtime system must keep the default SCREEN window creation; otherwise, existing applications would fail if their code was not modified.

Solution

Use the OPEN FORM + DISPLAY FORM instructions, to display the main form in the default SCREEN window:

```
MAIN
  OPEN FORM f FROM "form1"
  DISPLAY FORM f
  MENU "Example"
    COMMAND "Exit"
    EXIT MENU
  END MENU
END MAIN
```

Note: Another option is to delete the SCREEN window with the CLOSE WINDOW SCREEN instruction at the beginning of the program.
FAQ003: Why do some COMMAND KEY buttons no longer appear?

Description

When creating a MENU with COMMAND KEY(keyname) "option" clause, the button for keyname is no longer displayed:

```
MAIN
  MENU "Example"
    COMMAND "First"
    EXIT PROGRAM
    COMMAND KEY (F5) "Second"
    EXIT PROGRAM
    COMMAND KEY (F6) -- Third is a hidden option
    EXIT PROGRAM
  END MENU
END MAIN
```

Explanation

In BDS Version 3, when using the MENU instruction, several buttons are displayed for each clause of the type COMMANDKEY(keyname) "option": one for the menu option, and others for each associated key.

When using Genero, for a named MENU option defined with COMMAND KEY, the buttons of associated keys are no longer displayed (F5 in our example), because there is already a button created for the named menu option. The so-called "hidden menu options" created by a COMMAND KEY(keyname) clause (F6 in our example) are not displayed as long as you do not associate a label, for example with the FGL_SETKEYLABEL() function.

Related concepts

Setting action key labels on page 1775

Labels can be defined to decorate buttons controlled by ON KEY / COMMAND KEY action handlers.

FAQ004: Why aren't the elements of my forms aligned properly?

Description

In my forms, I used to align labels and fields by character, for typical terminal display. But now, when using the new LAYOUT section, some elements are not aligned as expected. In this example, the beginning of the field f001 is expected in the column near the end of the digit-based text of the first line, but the field is actually displayed just after the label "Name:"

```
LAYOUT
  GRID {
    01234567890123456789
    Name: [f001 ]
  }
END
END
ATTRIBUTES
  f001 = formonly.field1 TYPE CHAR;
END
```
Explanation

By default, Genero displays form elements with proportional fonts, using layout managers to align these elements inside the window. In some cases, this requires a review of the content of form screens when using the new layout management, because the layout is based on new alignment rules which are more abstract and automatic than the character-based grids in Version 3.

In most cases, the fglform compiler is able to analyze the layout section of .per form specification file to produce an acceptable presentation, but sometimes you will have to tweak the form files to give hints for the alignment of elements.

Solution

In this example, the field f001 alignment is based on the label appearing on the same line. By adding one space before the field position, the form compiler will understand that the field must be aligned to the text in the first line:

```
LAYOUT
  GRID {
    01234567890123456789
    Name:              [f001     ]
  }
END
END
ATTRIBUTES
  f001 = formonly.field1 TYPE CHAR;
END
```

In the next example, the fields are automatically aligned to the text in the first line:

```
LAYOUT
  GRID {
    01234567890123456789
    Name:        [f001     ]  [f002     ]
  }
END
END
ATTRIBUTES
  f001 = formonly.field1 TYPE CHAR;
  f002 = formonly.field2 TYPE CHAR;
END
```

Related concepts

Form rendering on page 1415
The section explains the layout rules to render forms on graphical front-ends.

FAQ005: Why doesn't the ESC key validate my input?

Description

The traditional ESC (escape) key does not validate an INPUT, it cancels the dialog instead.

Explanation

To follow common GUI standards, the ESC key is the standard key to cancel the current interactive statement.

Solution

You can change the accelerator keys for the 'accept' action with action defaults. However, is not recommended to change the defaults, because ESC is the standard key used to cancel a dialog in GUI applications.
Related concepts
Configuring actions on page 1744
Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with action attributes.

FAQ006: Why doesn't the Ctrl-C key cancel my input?

Description
The traditional Ctrl-C key does not cancel an INPUT statement.

Explanation
To follow platform GUI standards, the Ctrl-C key is used as the standard key to copy the current selected text to the clipboard, for cut and paste.

Solution
You can change the accelerator keys for the 'cancel' action with action defaults. However, is not recommended to change the defaults, because ESC is the standard key used to cancel a dialog in GUI applications.

Related concepts
Configuring actions on page 1744
Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with action attributes.

FAQ007: Why do the gui.* FGLPROFILE entries have no effect?

Description
The gui.* and some other FGLPROFILE entries related to graphics no longer have effect.

Explanation
These entries are related to the old user interface. They are no longer supported. In BDS version 3, the gui.* entries were interpreted by the front-end. As the user interface has completely been redesigned in Genero, some gui.* entries have been removed.

Solution:
Review all FGLPROFILE entries used in your current application and verify if there is a replacement.

Related concepts
FGLPROFILE: GUI configuration on page 235
Identify the Four Js BDS FGLPROFILE GUI configuration entries that are no longer supported, and the Genero BDL equivalent (where relevant).

FAQ008: Why do I get invalid characters in my form?

Description
The application starts, connects to the database and seems to work properly, but strange symbols (rectangles, question marks) are displayed in the forms for non-ASCII characters. The ASCII characters display properly.

Explanation
This is certainly a character set configuration error in on of the software components of you application (FGL runtime system, database client, GUI front-end, terminal emulator).
Solution
Use the correct runtime system locale and database client locale, as described in Application locale on page 512.

FAQ009: Why do large static arrays raise a stack overflow?

Description
When using very large static arrays (DEFINE a1 ARRAY[10000] OF ...), I get a stack overflow (typically, on Windows® platforms).

Explanation
The runtime system uses the default stack size defined by the C compiler. Because function static arrays are allocated on the C stack, using very large static arrays in functions can result in a stack overflow error.

Solution
Review the program and use dynamic array instead of static arrays.

FAQ010: Why do I get error -6366 "Could not load database driver drivername"?

Description
Error -6366 occurs when the runtime system fails to load the specified database driver.

Explanation
The database driver shared object (.so, .dylib or .DLL) or a dependent library could not be found.

Solution
Make sure that the specified driver name does not have a spelling mistake. If the driver name is correct, there is probably an environment problem. Make sure that the database client software is installed on the system (Genero does not communicate directly with the database server, you need the database client library). Check the UNIX™ LD_LIBRARY_PATH environment variable or the PATH variable on Windows®. These must point to the database client libraries. Another common error is the installation of a database client software of a different object type to the Genero runtime system. For example, if you install a 64 bit Genero version, you must install a 64 bit version of the database client software, the 32 bit version will not work.

Related concepts
Database client software on page 38
To connect to a database server, the database client software must be installed on the system where you run the Genero BDL programs.

**Configuration**

These topics cover configuration options of the Genero Business Development Language.

**The FGLPROFILE file(s)**

FGLPROFILE environment variable defines Genero BDL configuration files

**Understanding FGLPROFILE**

The runtime system uses one or more configuration files in which you can define options and parameters to change the behavior of the programs.

The FGLPROFILE files define standard BDL or user-defined entries with a name and value.

Standard entries can be used to control the runtime system behavior, and user-defined entries can be defined to configure your application.

**Tip:** Prefix user-defined entries with the name of your application to avoid conflicts with standard BDL FGLPROFILE entries.

Multiple profile files can be specified in the FGLPROFILE environment variable.

**Note:** On mobile devices, you must deploy a file with the name "fglprofile" in the appdir directory. See FGLPROFILE for mobile apps on page 261 for more details.

The list of standard core language FGLPROFILE entries can be found in FGLPROFILE entries for core language on page 256.

**Syntax of FGLPROFILE entries**

The syntax of FGLPROFILE entries is in the **key = value** form.

**Syntax**

```
# comment
entry-definition
```

where `entry-definition` is:

```
entry = value
```

where `entry` is:

```
ident \ident \ident \ident \ident \ident \ident \ident \ident \ident \ident \ident
```

and `value` is:

```
[-]digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \digit \di
3. *value* is a numeric value, a string literal, or a boolean value (true/false).

**Usage**

An FGLPROFILE entry is a line in the configuration file associating a parameter name to a value that can be specified as a numeric, string, or boolean.

**Important:** The encoding of FGLPROFILE files must match the application locale of the program. For more details, see FGLPROFILE encoding on page 261.

The entries are defined by a name composed of a list of identifiers separated by a dot character.

**Note:** FGLPROFILE entry names are case insensitive. In order to avoid any confusion, it is recommended to write FGLPROFILE entry names in lower case.

If an entry is defined several times in the same file, the last entry found in the file is used. No error is raised.

The value can be a numeric literal, a string literal, or a boolean (true/false).

Numeric values are composed by an optional sign, followed by digits, followed by an optional decimal point and digits:

```plaintext
my.numeric.entry = -1566.57
```

String values must be delimited by single or double quotes. The escape character is backslash, \t \n \r \f are interpreted as TAB, NL, CR, FF. Double the backslash to write a backslash character (\\):

```plaintext
my.string.entry = "C:\data\test1.dbf"
```

Boolean values must be either the `true` or `false` keyword:

```plaintext
my.boolean.entry = true
```

**Example**

```plaintext
# Last modification: 2013-03-12/mike
report.aggregatezero = true
gui.connection.timeout = 100
dbi.database.stores.source = "C:\data\test1.dbf"
dbi.database.stores.prefetch.rows = 200
```

**FGLPROFILE entries for core language**

This is a summary of FGLPROFILE entries supported by the core BDL.

Find more information for an FGLPROFILE entry by following the documentation link in the description of the entry.

This topic describes FGLPROFILE entries for the BDL core language. Web services specific FGLPROFILE entries description can be found in FGLPROFILE entries for Web Services on page 3489.

**Table 112: Partial list of supported FGLPROFILE entries**

<table>
<thead>
<tr>
<th>Entry</th>
<th>Values</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dialog.currentRowVisibleAfterSort</td>
<td>boolean</td>
<td>false</td>
<td>Forces current row to be shown after a sort in a table. See Dialog configuration with FGLPROFILE on page 1708.</td>
</tr>
<tr>
<td>Entry</td>
<td>Values</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>--------</td>
<td>---------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Dialog.fieldOrder</td>
<td>boolean</td>
<td>false</td>
<td>Defines if the intermediate field triggers must be executed when a new field gets the focus with a mouse click.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Dialog configuration with FGLPROFILE on page 1708.</td>
</tr>
<tr>
<td>dbi.default.driver</td>
<td>string</td>
<td>NULL</td>
<td>Defines the default database driver.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Default database driver on page 664.</td>
</tr>
<tr>
<td>dbi.database.dbname.driver</td>
<td>string</td>
<td>NULL</td>
<td>Defines the database driver for a database name.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Database driver specification (driver) on page 662.</td>
</tr>
<tr>
<td>dbi.database.dbname.source</td>
<td>string</td>
<td>NULL</td>
<td>Defines the data source for a database name.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Database source specification (source) on page 661.</td>
</tr>
<tr>
<td>dbi.*</td>
<td>N/A</td>
<td>N/A</td>
<td>Database interface configuration.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Connections.</td>
</tr>
<tr>
<td>fglrun.arrayIgnoreRangeError</td>
<td>boolean</td>
<td>false</td>
<td>Controls runtime behavior when array index is out of bounds.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Arrays on page 418 for more details.</td>
</tr>
<tr>
<td>fglrun.decToCharScale2</td>
<td>boolean</td>
<td>false</td>
<td>Formats DECIMAL(P) with 2 decimal digits globally.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Data type conversion reference on page 311.</td>
</tr>
<tr>
<td>fglrun.decToCharScale2.print</td>
<td>boolean</td>
<td>false</td>
<td>Formats DECIMAL(P) with 2 decimal digits in PRINT statements.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Data type conversion reference on page 311.</td>
</tr>
<tr>
<td>fglrun.floatToCharScale2</td>
<td>boolean</td>
<td>false</td>
<td>Formats FLOAT/SMALLFLOAT with 2 decimal digits globally.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Data type conversion reference on page 311.</td>
</tr>
<tr>
<td>fglrun.floatToCharScale2.print</td>
<td>boolean</td>
<td>false</td>
<td>Formats FLOAT/SMALLFLOAT with 2 decimal digits in PRINT statements.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Data type conversion reference on page 311.</td>
</tr>
<tr>
<td>fglrun.defaults</td>
<td>string</td>
<td>NULL</td>
<td>Defines the directory where program specific configuration files are located.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Understanding FGLPROFILE on page 255.</td>
</tr>
<tr>
<td>Entry</td>
<td>Values</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>--------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>fglrun.ignoreDebuggerEvent</code></td>
<td>boolean</td>
<td>false</td>
<td>Defines whether the runtime system can switch to debug mode.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See <a href="#">Integrated debugger</a> on page 2142.</td>
</tr>
<tr>
<td><code>fglrun.ignoreLogoffEvent</code></td>
<td>boolean</td>
<td>false</td>
<td>Defines whether the runtime system ignores a <code>CTRL_LOGOFF_EVENT</code> on Windows® platforms.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See <a href="#">Responding to <code>CTRL_LOGOFF_EVENT</code></a> on page 566.</td>
</tr>
<tr>
<td><code>fglrun.localization.*</code></td>
<td>N/A</td>
<td>N/A</td>
<td>Defines load parameters for localized string resource files.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See <a href="#">Localized strings</a> on page 538.</td>
</tr>
<tr>
<td><code>fglrun.mapAnyErrorToError</code></td>
<td>boolean</td>
<td>false</td>
<td>Controls default action of <code>WHENEVER ANY ERROR</code>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See <a href="#">Default exception handling</a> on page 508.</td>
</tr>
<tr>
<td><code>fglrun.mmapDisable</code></td>
<td>boolean</td>
<td>false</td>
<td>Turns program files memory mapping off on Windows® platforms.</td>
</tr>
<tr>
<td><strong>Important:</strong> This feature is deprecated, its use is discouraged although not prohibited.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Note: This entry is only provided to solve file overwrite issues when doing live program files updates on Windows® platforms.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See <a href="#">Dynamic module loading</a> on page 586.</td>
</tr>
<tr>
<td><code>flm.*</code></td>
<td>N/A</td>
<td>N/A</td>
<td>License management related entries.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See licensing documentation.</td>
</tr>
<tr>
<td><code>gui.connection.timeout</code></td>
<td>integer</td>
<td>30</td>
<td>Defines the timeout delay (in seconds) the runtime system waits when it establishes a connection to the front-end. After this delay the program stops with an error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See <a href="#">GUI connection timeout</a> on page 1118.</td>
</tr>
<tr>
<td><code>gui.key.add_function</code></td>
<td>integer</td>
<td>none</td>
<td>If set, this entry defines the offset for function key mapping when using Shift-Fx and Control-Fx key modifiers.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See <a href="#">Graphical mode with Traditional Display</a> on page 1114.</td>
</tr>
<tr>
<td>Entry</td>
<td>Values</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>--------</td>
<td>---------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>gui.programStoppedMessage</td>
<td>string</td>
<td>none</td>
<td>Generic message to be displayed to the end user when a program stops because of a runtime error. When not specified, fglrun displays the detailed error message. See Default exception handling on page 508.</td>
</tr>
<tr>
<td>gui.protocol.pingTimeout</td>
<td>integer</td>
<td>600</td>
<td>Defines the timeout delay (in seconds) the runtime system waits for a front-end ping when there is no user activity. After this delay the program stops with an error. See Wait for front-end ping timeout on page 1118.</td>
</tr>
<tr>
<td>gui.protocol.format</td>
<td>string</td>
<td>default</td>
<td>Controls Front-End protocol format. Possible values are: &quot;block&quot;, &quot;zlib&quot;. Default is &quot;block&quot; (encapsulation only). See GUI protocol compression on page 1119.</td>
</tr>
<tr>
<td>gui.rendering</td>
<td>string</td>
<td>native</td>
<td>Defines the GUI rendering mode. Possible values are: &quot;native&quot;, &quot;universal&quot;. Default is &quot;native&quot;. See Graphical mode with Universal Rendering on page 1112.</td>
</tr>
<tr>
<td>gui.server.autostart.*</td>
<td>N/A</td>
<td>N/A</td>
<td>Defines automatic front-end startup parameters. See Automatic front-end startup on page 1121.</td>
</tr>
<tr>
<td>gui.uiMode</td>
<td>string</td>
<td>NULL</td>
<td>Defines the user interface mode, to render windows in traditional 14GL mode. Possible values are: &quot;default&quot; or &quot;traditional&quot;. Default is the new Genero GUI mode with real resizable windows. See Graphical mode with Traditional Display on page 1114.</td>
</tr>
<tr>
<td>key.key-name.text</td>
<td>string</td>
<td>N/A</td>
<td>Defines a label for an action defined with an ON KEY clause. Note: This entry is provided for backward compatibility with BDS V3. See Setting action key labels on page 1775.</td>
</tr>
</tbody>
</table>

**Important**: This feature is deprecated, its use is discouraged although not prohibited.
<table>
<thead>
<tr>
<th>Entry</th>
<th>Values</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mobile.environment.name</td>
<td>N/A</td>
<td>N/A</td>
<td>Define environment variable values in FGLPROFILE for mobile applications.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Setting environment variables in FGLPROFILE (mobile) on page 263.</td>
</tr>
<tr>
<td>Report.aggregateZero</td>
<td>boolean</td>
<td>false</td>
<td>Defines if the report aggregate functions must return zero or NULL when all</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>values are NULL.</td>
</tr>
<tr>
<td>Important:</td>
<td></td>
<td></td>
<td>This feature is deprecated, its use is discouraged although not prohibited.</td>
</tr>
<tr>
<td>Note:</td>
<td></td>
<td></td>
<td>This entry is provided for backward compatibility with BDS V3.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Report engine configuration on page 2064.</td>
</tr>
</tbody>
</table>

**Related concepts**

FGLPROFILE entries for Web Services on page 3489

The FGLPROFILE entries relating to Genero Web Services are divided between five categories: security, basic or digest HTTP authentication, proxy configuration, server configuration, and XML cryptography.

**FGLPROFILE files selection**

When multiple FGLPROFILE files can be selected, the runtime system applies an order of precedence.

There are three levels of FGLPROFILE configuration files, and these files are loaded in the following order:

1. First, the runtime system reads the default configuration file provided in $FGLDIR/etc/fglprofile. This file contains all supported entries, identifies the possible values for an entry, and documents default values.

   **Important:** Do not modify the default FGLPROFILE configuration file, as it can be overwritten by a new installation.

2. Then, if the FGLPROFILE environment variable is set, the runtime system reads entries from the files specified by this environment variable. A list of files can be provided with FGLPROFILE. Files must be separated by the operating system specific path separator.

   **Note:** On mobile devices, you must deploy a file with the name "fglprofile" in the appdir directory. See FGLPROFILE for mobile apps on page 261 for more details.

3. After loading and merging the two previous levels, the runtime system checks whether the fglrun.defaults entry is set. This entry defines the program-specific profile directory. If this directory contains a file with the same name as the current program (without a .42r or .42m extension), the runtime system reads the entries from that file.

   **Note:** If the value of fglrun.defaults starts with $envvar, the specified environment variable is expanded to its value in order to build the path. For example, if the value is "$MYVAR/standard", and MYVAR contains /opt/app/config, and the program name is custinfo.42m (or custinfo.42r), the runtime system will try to read the file /opt/app/config/standard/custominfo.

The runtime system merges the different configuration files found at the three levels: If the same entry is defined in several files, the last loaded entry wins.

This means that the order of precedence is:

1. Program-specific configuration file (if fglrun.defaults is defined in one of the other levels).
2. Configuration files defined by the FGLPROFILE environment variable, or appdir/fglprofile, for mobile applications.
3. The default configuration file $FGLDIR/etc/fglprofile.
FGLPROFILE encoding

The encoding of the FGLPROFILE files must match the runtime system locale.

The application locale defines the character encoding used by the runtime system.

Since no character set conversion is done when reading entries from FGLPROFILE files, it is mandatory that the FGLPROFILE files use the same encoding as the runtime system.

If your application is designed to be executed with different encodings, consider defining ASCII-only values in FGLPROFILE files.

If character strings of FGLPROFILE entries need to be in a specific language selected at runtime, use string identifiers that point to localized strings. The language-specific messages can then be loaded dynamically with the LSTR(string-identifier) operator.

The default FGLPROFILE

The Genero BDL package ships a default FGLPROFILE file as $FGLDIR/etc/fglprofile.

Do NOT modify the $FGLDIR/etc/fglprofile default configuration file: This file will be overwritten by a new installation and your changes will be lost.

Make a copy of the default profile file, change the entries in your private configuration file, and specify this private file in the FGLPROFILE environment variable.

FGLPROFILE for mobile apps

The name of the FGLPROFILE file matters for mobile applications.

For non-mobile apps, there is no specific naming convention for FGLPROFILE configuration files. You can use a file name without an extension, or use the .txt or .prf extensions.

On mobile devices, it is not possible to define environment variables.

To specify a custom FGLPROFILE file for a mobile application, you must deploy a file with the name "fglprofile" in the appdir directory, along with the other application program files (.42m, .42f, and so on).

Only one custom FGLPROFILE file can be deployed for a given mobile application.

For more details, see Deploying mobile apps on page 3588.

Reading FGLPROFILE entries

Entries of FGLPROFILE files can be read by program.

Use the base.Application.getResourceEntry() method to read an entry of FGLPROFILE files.

Note: The FGL_GETRESOURCE() built-in function is equivalent to base.Application.getResourceEntry().

Consider the following facts when reading FGLPROFILE entries:

1. FGLPROFILE entries are case insensitive. However, you should use the exact name to read the entry. Consider using lowercase identifiers like myapp.myentry.
2. Multiple files can be defined in the FGLPROFILE environment variable, and multiple entries with the same name can be defined. In such case, the last found wins.
3. The encoding of the FGLPROFILE file must match the encoding of the runtime system.
4. Use a STRING variable to hold the value of the FGLPROFILE entry to be read.

Example:

```
MAIN
   DISPLAY base.Application.getResourceEntry("myapp.myentry")
END MAIN
```
With the `p1.prf` file defining:

```
myapp.myentry = "aaa"
```

And the `p2.prf` file defining:

```
myapp.myentry = "bbb"
myapp.myentry = "ccc"
```

Setting FGLPROFILE environment variable to this value:

```
export FGLPROFILE="p1.prf:p2.prf"
```

The last found entry is read:

```
$ fglcomp main.4gl && fglrun main.42m
ccc
```

---

### Environment variables

Genero BDL related environment variables.

#### Setting environment variables on UNIX™

On UNIX™ platforms, environment variables can be set through the following methods, depending on the command interpreter used:

- **Bourne shell:**
  
  ```
  VAR=value; export VAR
  ```

- **Korn shell:**
  
  ```
  export VAR=value
  ```

- **C shell:**
  
  ```
  setenv VAR=value
  ```

For more details, refer to the documentation for your UNIX™ system.

#### Setting environment variables on Windows™

On Windows™ platforms, environment variables can be set by one of the following methods:

- In a command window, with the `SET` command.
- In the registry, for the current user in `HKEY_CURRENT_USER` or a global setting in `HKEY_LOCAL_MACHINE`.

For more details, refer to the documentation of your Windows™ system.

On Windows™, double quotes do not have the same meaning as on UNIX™ systems. For example, if you set a variable with the command `SET VAR="abc"`, the value of the variable will be "abc" (with double quotes), and not abc. When using Informix®, some variables related to the database engine must be set using the SETNET32 utility.
Setting environment variables in FGLPROFILE (mobile)

When executing applications on mobile devices, you can configure environment settings with FGLPROFILE entries. Setting an environment variable with an FGLPROFILE entry is equivalent to setting the environment variable before running the fglrun VM process on a server.

**Note:** Environment variables set in an FGLPROFILE file are only read when the deployed application runs on the mobile device. They are not read during development mode (that is when the VM runs on the development machine and the mobile client displays on the device). The FGLPROFILE environment variable settings are only for the VM component and are ignored by the GMA/GMI front-end component.

FGLPROFILE environment variables settings can be used to define DBDATE and DBFORMAT, if the default regional settings on the mobile must be ignored for date and numeric value formatting. Note that defining DBMONEY will have no effect, because DBFORMAT is defined automatically by the GMI or GMA front-end component before starting the VM component. Since DBFORMAT takes precedence over DBMONEY, setting DBMONEY in FGLPROFILE is pointless.

**Important:** C-runtime library variables such as LANG/LC_ALL cannot be set with FGLPROFILE entries, because the C-runtime library is (and must be) initialized before reading FGLPROFILE files.

The syntax is:

```
mobile.environment.env_name = "env_value"
```

where:

1. `env_name` is the name of the environment variable to be set.
2. `env_value` is the value for the `env_name` environment variable.

For example:

```
mobile.environment.MY_ENV_VAR = "my value"
```

The value specified in a `mobile.environment` entry can contain `$NAME` placeholders, that will be replaced by the actual value of the `NAME` environment variable. The `NAME` environment variable will typically be set by the front-end component, before starting the runtime system component, for example to define FGLDIR and FGLAPPDIR values.

If the environment variable contains directory or file paths, use the UNIX® path notation with / slashes as directory name separator, and the : colon as path separator.

This example defines the FGLIMAGEPATH environment variable for the mobile app, using FGLAPPDPR and FGLDIR predefined environment variables:

```
mobile.environment.FGLIMAGEPATH = "$FGLAPPDIR/myimages:$FGLAPPDIR/icons/myimage2font.txt:$FGLDIR/lib/image2font.txt"
```

**Note:** During development (when executing programs on a server), consider defining environment variables such as FGLAPPDPR in the shell environment, along with the other environment variables that are defined with mobile.environment entries, as these are only read when executing on mobile devices.

**Related concepts**

Mobile applications on page 3565
These topics cover programming subjects about mobile applications

**Operating system environment variables**

Describes some well-known system environment variables that are used by Genero software components.

**LC_ALL (or LANG)**

Defines the current application locale on UNIX™ platforms.

The LC_ALL (or LANG) environment variable defines language, territory and codeset for programs running on UNIX™ platforms.

The codeset defined in LC_ALL is used by the runtime system to handle character strings.

It is important to set this variable properly to the character set used by your application.

If LC_ALL is not defined, LANG is used instead.

Read the UNIX™ man page of the `setlocale()` C function for more details about this variable.

**Related concepts**

Localization on page 512

Localization support allows you to implement programs that follow specific language and cultural rules.

**LD_LIBRARY_PATH**

Defines search paths to find shared libraries on UNIX™ platforms.

The LD_LIBRARY_PATH environment variable defines the list of search paths for shared libraries loaded by the dynamic linker on UNIX™ platforms.

On some operating systems, the environment variable defining the shared library search path may have a different name.

- On a system where a 32-bit and a 64-bit environment coexist, you may need to set `LD_LIBRARY_PATH_64` to execute the 64-bit programs.
- On HP/UX, set `SHLIB_PATH`.
- On AIX®, set `LIBPATH`.
- On Mac OS X®, the usage of `DYLD_LIBRARY_PATH` is discouraged. Therefore, shared libraries that are not part of the Genero runtime system (such as database client libraries) must be found in the standard system directories (`/usr/lib`, `/usr/local/lib`)

**PATH**

Defines the list of search paths to find executable files.

The PATH environment variable defines the list of search paths for executable files.

On UNIX™ platforms, PATH defines the search path list for executable programs.

On Windows™ platforms, PATH defines the search path for programs and DLLs.

The path separator is a colon (:) on UNIX™ and a semicolon (;) on Windows™.

**TERM**

Defines the type of terminal on UNIX™ platforms.

The TERM variable is used by UNIX™ and Genero applications to identify the terminal type when running in TUI mode.

By default or when `INFORMIXTERM` equals `termcap`, Genero reads terminal capabilities from the file defined by the TERMCAP environment variable. When `INFORMIXTERM` is set to `terminfo`, Genero reads terminal capabilities from the terminfo database of the system.

TERMCAP is the older implementation of terminal capabilities database. Therefore, it is not recommended to set `INFORMIXTERM=terminfo`. 
It is important to define this variable properly to match the text terminal hardware or the terminal emulation you are using.

**Related concepts**

**TERMCAP** on page 265
Defines the *termcap* terminal capabilities database on UNIX™ platforms.

**INFORMIXTERM** on page 284
Defines terminal control library to be used.

**Using a text terminal** on page 1123
This section covers topics about text terminal configuration when using the TUI mode.

**TERMCAP**
Defines the *termcap* terminal capabilities database on UNIX™ platforms.

**Usage**

For UNIX™ platforms, TERMCAP is an environment variable that defines the terminal capabilities file. This variable must be used in conjunction with TERM, when INFORMIXTERM is set to *termcap*, or when INFORMIXTERM is not set.

If the TERMCAP variable is not defined, Genero tries to open `/etc/termcap`. If no `/etc/termcap` file exists, the runtime system uses `$FGLDIR/etc/termcap`. You can add more terminal definitions in this file.

TERMCAP is the older implementation of terminal capabilities database. It is recommended that you set INFORMIXTERM=terminfo.

It is important to define terminal capabilities properly for your text terminal hardware or the terminal emulation you are using. Especially function keys (F1, F16) and display attributes (bold, reverse, colors) may not work if the escape sequences do not correspond to the terminal used.

For more details about the TERMCAP environment variable, please refer to your UNIX™ operating system manual.

**Related concepts**

**TERM** on page 264
Defines the type of terminal on UNIX™ platforms.

**INFORMIXTERM** on page 284
Defines terminal control library to be used.

**Using a text terminal** on page 1123
This section covers topics about text terminal configuration when using the TUI mode.

**TERMINFO**
Defines the *terminfo* terminal capabilities database.

On UNIX™ platforms, the TERMINFO environment variable points to the terminal capabilities database. This variable must be used along with TERM, when INFORMIXTERM is set to *terminfo*.

Setting this environment variable is generally not necessary. The default is defined by the UNIX™ system, it can be for example `/etc/terminfo`, `/usr/lib/terminfo`, or `/lib/terminfo`.

It is important to define terminal capabilities properly for your text terminal hardware, or the terminal emulation you are using. In particular, function keys (F1, F16) and display attributes (bold, reverse, colors) may not work if the escape sequences do not correspond to the terminal used.

For more details about the TERMINFO environment variable, please refer to your UNIX™ operating system manual.

**Related concepts**

**TERM** on page 264
Defines the type of terminal on UNIX™ platforms.

**INFORMIXTERM** on page 284
Defines terminal control library to be used.

**Using a text terminal** on page 1123

This section covers topics about text terminal configuration when using the TUI mode.

**TMPDIR, TMP, TEMP**

Defines the directory for temporary files.

The TMPDIR, TEMP and TMP environment variables define the directory where temporary files are created by the operating system and by some other software (TMPDIR is typically used on UNIX™ platforms, TEMP and TMP are used on Windows™)

On desktop and server platforms, consider using DBTEMP to define the temp file directory for runtime system temporary files.

On mobile devices, there is no need to define the TMPDIR (or DBTEMP) environment variable: The runtime system will automatically use the appropriate temporary directory within the app sandbox file system.

**Database client environment variables**

Programs connecting to a database server use a database driver that in turn uses a database client library. The database client software usually needs configuration settings that are defined with environment variables. Database client environment variable define information such as installation directory of the client software, localization settings, temporary directory, and more.

Refer to the database client software documentation for the required environment variable settings.

**Related concepts**

[Database client environment](on page 658)

To connect to a database server, Genero BDL programs use vendor's database client software.

**Genero environment variables**

This section lists and describes in detail all Genero specific environment variables.

**DBCENTURY**

Specifies the expansion for the century in **DATE** and **DATETIME** values.

The DBCENTURY environment variable specifies how to expand abbreviated one- and two-digit *year* specifications within **DATE** and **DATETIME** values, especially during field input.

**Important:** The DBCENTURY environment variable is also used by the IBM® Informix® database client and server to make date to string conversions.

Default value is "R" (prefix the entered value with the first two digits of the current year).

Values are case sensitive; only the four uppercase letters are valid.

**Table 113: DBCENTURY valid values**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Algorithm for Expanding Abbreviated Years</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Use the past, future, or current year closest to the current date.</td>
</tr>
<tr>
<td>F</td>
<td>Use the nearest year in the future to expand the entered value.</td>
</tr>
<tr>
<td>P</td>
<td>Use the nearest year in the past to expand the entered value.</td>
</tr>
<tr>
<td>R</td>
<td>Prefix the entered value with the first two digits of the current year.</td>
</tr>
</tbody>
</table>

If a year is entered as a single digit, it is first expanded to two digits by prefixing it with a zero; DBCENTURY then expands this value to four digits.

Three-digit years are not expanded.
Years before 99 AD (or CE) require leading zeros (to avoid expansion).

If the database server and the client system have different settings for DBCENTURY, the client system setting takes precedence for abbreviations of years in dates entered through the application. Expansion is sensitive to the time of execution and to the accuracy of the system clock-calendar. You can avoid the need to rely on DBCENTURY by requiring the user to enter four-digit years or by setting the CENTURY attribute in the form specification of DATE and DATETIME fields.

**Related concepts**

- Formatting **DATE values** on page 321
  Date values must be formatted when converted to strings.
- **Date, numeric and monetary formats** on page 532
  This section describes how Genero BDL handles date, time, numeric and monetary formats.

**DBDATE**

Defines the default display and input format for **DATE** values.

The DBDATE environment variable defines the default display and input format for **DATE** values.

**Important:** The DBDATE environment variable is also used by the IBM® Informix® database client and server to make date to string conversions.

DBDATE defines the order of the month, day, and year time units within a string representing a date with numeric month and day such as "24/04/2014".

Values of DBDATE must be a restricted combination of symbols representing the position of the year (Yn), month (M) and day (D), the separator and some optional configuration options. For example, DMY4/ defines a date format with the day unit at the first position, followed by the month and the year (on 4 digits): "dd/mm/yyyy".

The separator always goes at the end of the format string (for example, DMY2/). If no separator or an invalid character is specified, the slash ( / ) character is the default. Specifying a 0 (zero) as separator indicates that no separator is used.

The default value of DBDATE depends on the type of platform: On desktop/server platforms, the default setting for DBDATE is: MDY4/. On mobile platforms, DBDATE defaults to the regional settings defined on the device.

**Table 114: Valid DBDATE symbols**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning in DBDATE format string</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Day of month (can be one or two digits on input)</td>
</tr>
<tr>
<td>M</td>
<td>Month (can be one or two digits on input)</td>
</tr>
<tr>
<td>Y2</td>
<td>Year as two digits (can be one or two digits on input)</td>
</tr>
<tr>
<td>Y3</td>
<td>Year as three digits (Ming Guo format only)</td>
</tr>
<tr>
<td>Y4</td>
<td>Year as four digits (can be 1, 2, 3 or 4 digits on input)</td>
</tr>
<tr>
<td>/</td>
<td>Default time-unit separator for the default locale</td>
</tr>
<tr>
<td>C1</td>
<td>Ming Guo format modifier (years as digits)</td>
</tr>
<tr>
<td>-</td>
<td>Hyphen time-unit separator</td>
</tr>
<tr>
<td>.</td>
<td>Period time-unit separator</td>
</tr>
<tr>
<td>0</td>
<td>Indicates no time-unit separator</td>
</tr>
</tbody>
</table>

The combinations must follow a specific order:

```
| Y2 | Y3 | Y4 | D | M | Y2 | Y3 | Y4 | / | / | / | / | C1 |
```
When a form field and its corresponding variable are defined with the DATE type, values will be displayed depending on the DBDATE format, except if a FORMAT attribute is defined.

The DBDATE format is also used to automatically convert a character string to/from a DATE value in programs.

Note that DBDATE takes also effect when fetching DATE values from the database into CHAR/VARCHAR program variables. However, it is not recommended to fetch date information into string variables, it is recommended that you use DATE or DATETIME variables instead.

The C1 modifier can be used at the end of the DBDATE value in order to use Ming Guo date format with digit-based years. When using C1, you can use one of the Y4, Y3 or Y2 symbols for the year.

A Gregorian date format can look like "DYM4/", while a Ming Guo date format would look like "Y3MD/C1".

Date formatting specified in a USING clause or FORMAT attribute overrides the formatting specified in DBDATE.

**Related concepts**

[Formatting DATE values](#) on page 321
Date values must be formatted when converted to strings.

[Date, numeric and monetary formats](#) on page 532
This section describes how Genero BDL handles date, time, numeric and monetary formats.

[Using the Ming Guo date format](#) on page 532
Genero BDL can be configured to use the The Ming Guo calendar.

**DBDELMITER**
Defines the value separator for unload data files.

The DBDELMITER environment variable defines the value delimiter for LOAD and UNLOAD instructions.

If DBDELMITER is not defined, the default delimiter is a ( | ) pipe.

Do not use backslash or hex digits (0-9, A-F, a-f).

DBDELMITER defines also the default delimiter for the base.Channel in/out API.

**DBEDIT**
Defines the editor program for TEXT fields in TUI mode.

The DBEDIT environment variable defines the editor program to modify the values of form fields defined with the TEXT data type, when running programs on dumb terminals.

**DBFORMAT**
Defines the characters to be used for the currency symbol, decimal and thousands separators for numeric values.

The DBFORMAT environment variable defines the front currency symbol, the thousands separator, the decimal separator and the back currency symbol, when converting character strings to/from numeric values.

**Important:**

- When defined, the DBFORMAT environment variable takes precedence over DBMONEY.
- The DBFORMAT environment variable is also used by the IBM® Informix® database client and server to make date to string conversions.
- When using a graphical front-end, the decimal separator of the numeric keypad will produce the character defined by the DBFORMAT (or DBMONEY) environment variables.

The value of a DBFORMAT variable must use the following syntax:

```
[front]:[thousands]:decimal:[back]
```

1. **front** is the leading currency symbol, can be an asterisk ( * ) to avoid the front symbol. The front symbol can be omitted when not used.
2. **thousands** is a character that you specify as a valid thousands separator, can be an asterisk ( * ) to avoid the thousands separator. The thousands symbol can be omitted when not used.
3. *decimal* is a character that you specify as a valid decimal separator. Using the asterisk ( * ) will have the same result as when using a dot.

4. *back* is the trailing currency symbol, can be an asterisk ( * ) to avoid the trailing symbol. The *back* symbol can be omitted when not used.

**Note:** DBFORMAT takes precedence over DBMONEY.

If neither DBMONEY, nor DBFORMAT are defined, the default numeric formatting depends on the type of platform where the runtime system executes:

- On desktop/server platforms, the default numeric format defines the ( , ) comma as thousands separator, the ( . ) dot as decimal separator, and the ( $ ) dollar sign as front currency symbol for MONEY values. This corresponds to DBMONEY="$.", or DBFORMAT="$:,:.:".
- On mobile platforms, the numeric format defaults to the regional settings defined on the device. Normally, there is no need to modify these defaults.

DBFORMAT affects the string to/from numeric conversions, by defining the thousands separator, decimal separator and (front or back) currency symbol, depending on the numeric type:

- **MONEY:** Thousands separator (when using a format string), decimal separator and currency symbol.
- **DECIMAL, SMALLFLOAT, FLOAT:** Thousands separator (when using a format string) and decimal separator.
- **SMALLINT, INTEGER, BIGINT:** Thousands separator (when using a format string)

DBFORMAT can specify the leading and trailing currency symbols (but not their default positions within a monetary value) and the decimal and thousands separators. The decimal and thousands separators defined by DBFORMAT apply to both monetary and other numeric data.

DBFORMAT is typically used in conjunction with following language elements:

- The **FORMAT** field attribute in form files, to specify a formatting string with placeholders for the symbols defined by DBFORMAT.
- The **USING** operator, to specify a formatting string with placeholders for the symbols defined by DBFORMAT.
- When using numeric expressions in DISPLAY and PRINT instructions, when default formatting applies (when no USING operator is specified)
- Areas where character string to/from numeric conversion is needed, for example in a LET statement, where string expression is assigned a monetary or number variable, or in LOAD and UNLOAD instructions, to convert the numeric values in the flat files to pass data to or from the database.

With the default formatting, the *thousands* separator is not added to the result. To get the thousands separator, specify a format string including coma (the placeholder for the thousands separator), for example with USING "---, ---, ---&.& &".

Specify only one character for *decimal* or *thousands* symbols. The *front* or *back* currency symbol can be defined with more than one character, for example: DBFORMAT="* :, :, : FR".

Any printable character that your locale supports is valid for the *thousands* separator or for the *decimal* separator, except 0–9 digits, <,>,|,?, !=, [ and ].

The asterisk ( * ) specifies that a symbol or separator is not applicable; it is the default for any front, thousands, or back term that you do not define, for example with DBFORMAT="$: *, : : *".

The same character cannot be both the *thousands* and *decimal* separator. A blank space (ASCII 32) can be the *thousands* separator (and is conventionally used for this purpose in some locales). The asterisk ( * ) symbol is valid as the *decimal* separator, but is not valid as the *thousands* separator.

Enclosing the DBFORMAT specification in a pair of single quotation marks is recommended to prevent the shell from attempting to interpret (or execute) any of the DBFORMAT characters.

The setting in DBFORMAT affects how formatting masks of the **FORMAT** attribute and **USING** operator are interpreted. In formatting masks of **FORMAT** and **USING**, these symbols are not literal characters but are placeholders for what DBFORMAT specifies:

- The dollar ($) sign is a placeholder for the *front* currency symbol.
- The comma (,) is a placeholder for the *thousands* separator.
- The period (.) is a placeholder for the *decimal* separator.
- The at (@) sign is a placeholder for the *back* currency symbol.

This table illustrates the results of different combinations of DBFORMAT setting and format string on the same value.

**Table 115: Results of combinations of DBFORMAT setting and format string on the same value**

<table>
<thead>
<tr>
<th>Value</th>
<th>Format String</th>
<th>DBFORMAT</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234.56</td>
<td>$#,###.##</td>
<td>$:,,:</td>
<td>$1,234.56</td>
</tr>
<tr>
<td>1234.56</td>
<td>$#,###.##</td>
<td>:,:,:DM</td>
<td>1,234,56</td>
</tr>
<tr>
<td>1234.56</td>
<td>#,###.##@</td>
<td>$:,,:</td>
<td>1,234.56</td>
</tr>
<tr>
<td>1234.56</td>
<td>#,###.##@</td>
<td>:,:,:DM</td>
<td>1,234,56DM</td>
</tr>
<tr>
<td>1234.56</td>
<td>$#,###.##@</td>
<td><em>,:,::</em></td>
<td>1234.56</td>
</tr>
</tbody>
</table>

When the user enters numeric or currency values in fields, the runtime system behaves as follows:

- If a symbol is entered that was defined as a decimal separator in DBFORMAT, it is interpreted as the decimal separator.
- For **MONEY** fields, it disregards any *front* (leading) or *back* (trailing) currency symbol and any thousands separators that the user enters.
- For **DECIMAL** fields, the user must enter values without currency symbols.

When the runtime system displays or prints values:

- The DBFORMAT-defined leading or trailing currency symbol is displayed for **MONEY** values.
- If a leading or trailing currency symbol is specified by the **FORMAT** attribute for non-MONEY data types, the symbol is displayed.
- The thousands separator is not displayed unless it is included in a formatting mask of the **FORMAT** attribute or of the USING operator.

When **MONEY** values are converted to character strings by the **LET** statement, both automatic data type conversion and explicit conversion with a **USING** clause insert the DBFORMAT-defined separators and currency symbol into the converted strings.

For example, suppose DBFORMAT is set as follows:

```
*,:,:,SFr
```

The value 1234.56 will print or display as follows:

```
1234,56SFr
```

Here *SFr* stands for the Swiss Franc currency symbol. Values input by the user into a screen form are expected to contain commas, not periods, as their decimal separator because DBFORMAT has *,:,:,SFr* as its setting in this example.
Related concepts
Formatting DATE values on page 321
Date values must be formatted when converted to strings.

Date, numeric and monetary formats on page 532
This section describes how Genero BDL handles date, time, numeric and monetary formats.

Type conversions on page 311
Explains primitive data type conversion rules of the language.

FORMAT attribute on page 1366
The FORMAT attribute defines the data formatting of numeric and date time fields, for input and display.

DBMONEY
Defines the characters to be used for the currency symbol and decimal separator for numeric values, when DBFORMAT is not defined.

The DBMONEY environment variable defines the currency symbol and the decimal separator when converting character strings to/from numeric values.

Important:
• When defined, the DBFORMAT environment variable takes precedence over DBMONEY.
• The DBMONEY environment variable is also used by the IBM® Informix® database client and server to make date to string conversions.

The value of a DBMONEY variable must use the following syntax:

```
front.decimal separator.back
```

1. front is a character string representing a leading currency symbol that precedes the value.
2. back is a character string representing a trailing currency symbol that follows the value.
3. The decimal separator is mandatory and can only be a dot ( . ) or a comma ( , )

If neither DBMONEY, nor DBFORMAT are defined, the default numeric formatting depends on the type of platform where the runtime system executes:

• On desktop/server platforms, the default numeric format defines the ( , ) comma as thousands separator, the ( . ) dot as decimal separator, and the ( $ ) dollar sign as front currency symbol for MONEY values. This corresponds to DBMONEY="$.", or DBFORMAT="$:,,:".
• On mobile platforms, the numeric format defaults to the regional settings defined on the device. Normally, there is no need to modify these defaults.

DBMONEY can only define the currency symbol and decimal separator characters. To define the thousands separator, use the DBFORMAT environment variable instead. However, if only DBMONEY is used, an implicit thousands separator is selected.

The currency symbol in DBMONEY can be up to seven characters long and can contain any character except a comma or a period. It can be non-ASCII characters if the current locale supports a code set that defines the non-ASCII characters you use.

DBMONEY is used for implicit data conversion between numeric values and character strings, for example when using numeric values in form fields and reports. Other areas regarding string to/from numeric conversion are affected by the DBMONEY. For more details, see DBFORMAT on page 268.

The position of the currency symbol (relative to the decimal separator) indicates whether the currency symbol appears before or after the MONEY value. When the currency symbol is positioned in DBMONEY before the decimal separator, it is displayed before the value ($1234.56). When it is positioned after the decimal separator, it is displayed after the value (1234.56F).
The runtime system recognizes the period (.) and the comma (,) as decimal separators. All other characters are considered to be part of the currency symbol. For example, ",, FR" defines a MONEY format with the comma as decimal separator and the string " FR" (including the space) as the currency symbol.

Because only its position within a DBMONEY setting indicates whether a symbol is the front or back currency symbol, the decimal separator is required. If you use DBMONEY to specify a back symbol, for example, you must supply a decimal separator (a comma or period). Similarly, if you use DBMONEY to change the decimal separator from a period to a comma, you must also supply a currency symbol.

To avoid ambiguity in displayed numbers and currency values, do not use the thousands separator of DBFORMAT as the decimal separator of DBMONEY. For example, specifying comma as the DBFORMAT thousands separator dictates using the period as the DBMONEY decimal separator.

When using a graphical front-end, the decimal separator of the numeric keypad will produce the character defined by this environment variable.

**Related concepts**

- **DBFORMAT** on page 268
  
  Defines the characters to be used for the currency symbol, decimal and thousands separators for numeric values.

- **DBPATH**
  
  Defines the paths to search for Genero program resource files.

  For IBM® Informix® 4GL compatibility, DBPATH is used by the runtime system to find resource files such as form definitions.

  **Important:** The DBPATH environment variable is also used by the IBM® Informix® SE engine and SQLite, to define the path list to find database files. Genero has introduced the FGLRESOURCEPATH environment variable to not interfere with the database DBPATH settings. Consider dedicating DBPATH for database configurations, and use the FGLRESOURCEPATH to define program resource path list.

  DBPATH must contain a list of paths, separated by the operating system specific path separator.

  The path separator is platform specific (";" on UNIX™ platforms and ";" on Windows® platforms).

  See FGLRESOURCEPATH for more details about resource files search path.

- **DBPRINT**
  
  Defines the print device to be used by reports.

  The DBPRINT environment variable specifies the print device to be used by reports defined TO PRINTER.

  On UNIX™ systems, the DBPRINT environment variable typically contains the printer queue command (such as lp).

  When defining DBPRINT=FGLSERVER, the report is sent to the printer configured in the Genero Desktop Client (GDC). See also setReportFont on page 2663, setReportPrinter on page 2664 front calls.

  **Related concepts**

  - **Reports** on page 2026

- **DBSCREENDUMP**
  
  Defines the output file name for text screen shots.

  The DBSCREENDUMP environment variable defines the output file name for text screen shots when pressing Ctrl-P.

  When using the TUI mode, if the user pressed the Ctrl-P key, the runtime system will dump the current screen into the file defined by this variable.

  Unlike DBSCREENOUT, the output of DBSCREENDUMP includes the escape sequences of TTY attributes, which makes it less readable.

  **Related concepts**

  - **DBSCREENOUT** on page 273
Defines the output file name for text screen shots.

**DBSCREENOUT**
Defines the output file name for text screen shots.

The DBSCREENOUT environment variable defines the output file name for text screen shots when pressing Ctrl-P.

When using the TUI mode, if the user pressed the Ctrl-P key, the runtime system will dump the current screen into the file defined by this variable.

Unlike DBSCREENDUMP, the output of DBSCREENOUT excludes the escape sequences of TTY attributes.

**Related concepts**
- [DBSCREENDUMP](#) on page 272
  Defines the output file name for text screen shots.

**DBTEMP**
Defines the directory for temporary files.

The DBTEMP environment variable defines the directory for temporary files created by the runtime system.

If the DBTEMP variable is not defined, the runtime system uses the temporary directory as defined on the operating system. Depending on the platform, the TMPDIR, TMP, TEMP environment variables, or the default system temp directory will be used.

**Important:** The DBTEMP environment variable is also used by the IBM® Informix® database client and server for temporary files.

The temporary directory is used to create temporary files for:

1. **TEXT** or **BYTE** data located in a temporary file (LOCATE IN FILE without file name specification).
2. Temporary files of emulated scrollable cursors when the database engine does not support this feature.
3. Temporary file name generation with `os.Path.makeTempName()`.
4. Temporary files created by the Web Services API, such as `com.HTTPResponse.getFileResponse` on page 2897.

On mobile devices, do not set DBTEMP environment variable: The runtime system will automatically use the appropriate temporary directory within the app sandbox file system.

**FGL_LENGTH_SEMANTICS**
Defines the length semantics to be used in programs.

Define the FGL_LENGTH_SEMANTICS environment variable to specify byte or character length semantics, by setting the value to BYTE or CHAR, respectively.

If the variable is not set, byte length semantics will be used by default.

When using a single-byte character set such as ISO-8859-1, use byte length semantics (the default). If the application character set is UTF-8, it is recommended that you use char length semantics.

**Related concepts**
- [Length semantics settings](#) on page 521

**FGLAPPDIR**
Contains the path to the application directory when executing on a mobile device.

When executing on mobile devices, the FGLAPPDIR environment variable is an automatic environment variable that contains the path to the appdir directory, containing application program files (.42m, .42f, and other resources).

This variable is typically used to define environment variables with `mobile.environment` FGLPROFILE entries, relative to the mobile appdir where application program files and resources are located.

**Note:** During development (when executing programs on a server), consider defining the FGLAPPDIR in the shell environment, along with the other environment variables that are defined with `mobile.environment` entries, as these are only read when executing on mobile devices.
**Related concepts**

**Directory structure for GMI apps** on page 3604
Platform-specific rules need to be considered when deploying on iOS devices (GMI).

**Directory structure for GMA apps** on page 3589
Platform-specific rules need to be considered when deploying on Android™ devices (GMA).

**FGLAPPSERVER**
Defines the listening TCP port of the Web service in development context.

The FGLAPPSERVER environment variable defines the TCP port on which the web service server will be started.

If the FGLAPPSERVER environment variable is not set, the default TCP port is 80.

During development, define this environment variable before starting the web service server program, to let web service clients connect directly to the runtime system. You typically defined FGLAPPSERVER to the port 8090.

In production, Genero Application Server (GAS) is used to deploy web services servers. The GAS will automatically set FGLAPPSERVER. Do not manually set FGLAPPSERVER when GAS is involved.

**Related concepts**

The **WebServiceEngine class** on page 2828
The `com.WebServiceEngine` class provides an interface to manage the Web Services engine.

**Step 6: Create the server** on page 3369
Provide a file with a BDL function that starts your Web service with the Genero Web Service server instead of Axis.

`fgl_ws_server_start()` (version 1.3) on page 3557
Creates and starts the Web services server.

**Testing the GWS service in stand-alone mode** on page 3355
Test that your service is reachable and that it can generate the WSDL.

**Related reference**

Genero BDL errors on page 3154
System error messages sorted by error number.

**FGLCOV**
Enables coverage data collection.

The FGLCOV environment variable can be used to enable coverage data collection while executing programs.

When the FGLCOV variable is set, `fglrun` produces `module.42m.cov` files.

For more details, see **Source code coverage** on page 2168.

**FGLDBPATH**
Defines the path to database schema files for compilers.

The `fglcomp` on page 2071 and `fglform` compilers need database schema files to compile source modules and forms. The path to the database schema files can be specified with FGLDBPATH.

If FGLDBPATH is not defined, the current directory is the default path for the database schema files. You can provide a list of paths, separated by the operating system specific path separator. FGLDBPATH is only used in development.

FGLDBPATH must contain a list of paths, separated by the operating system specific path separator. The path separator is ":" on UNIX™ platforms and ";" on Windows® platforms.

**Related concepts**

Database schema on page 467
Defines database table structures with column type information to be reused in program variable definitions.

**FGLDIR**
Defines the installation directory of Genero Business Development Language.

The FGLDIR environment variable defines the installation directory of the runtime system and compilers of Genero.

When executing on a mobile device, the FGLDIR environment variable is automatically set by the front-end component, before starting the runtime system component. As result, it is possible to use the $FGLDIR keyword in FGLPROFILE environment variable settings when executing on mobile devices.

**Related concepts**
- **PATH** on page 264
- Defines the list of search paths to find executable files.

**FGLGBCDIR**
Defines the directory of the GBC component for Universal Rendering.

When using the Universal Rendering mode, it is possible to define the directory where the GBC component must be loaded from, by setting the FGLGBCDIR environment variable.

The GBC component will be searched in the following directories:

1. The `appdir/gbc` directory, where `appdir` is the directory where the program file is located,
2. The directory defined in the FGLGBCDIR environment variable,
3. The `$FGLDIR/web_utilities/gbc/gbc` directory.

If defined, make sure that FGLGBCDIR is set to a directory containing the GBC component files.

**Note:** The FGLGBCDIR environment variable is provided to select a given GBC in a development context when using the GUI direct mode. Do not use FGLGBCDIR in a production context: If a specific GBC is needed in production, consider shipping it in `appdir/gbc`. Otherwise, use the GBC available by default.

**Related concepts**
- **Connecting with a front-end** on page 1116
- **Graphical mode with Universal Rendering** on page 1112

**FGLGUI**
Defines the user interface mode to be used by the program.

The FGLGUI environment variable indicates whether the applications are run in TUI or GUI mode.

**Table 116: FGLGUI values**

<table>
<thead>
<tr>
<th>FGLGUI</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (zero)</td>
<td>The application executes in TUI (text) mode.</td>
</tr>
<tr>
<td>1 (default)</td>
<td>The application executes in GUI (graphical) mode and needs a front-end to display application windows.</td>
</tr>
</tbody>
</table>

**Related concepts**
- **Genero user interface modes** on page 1109
- User interface modes allow you to adapt the application form rendering to different types of displays.

**FGLGUIDEBUG**
Defines the debug level in GUI mode.

The FGLGUIDEBUG environment variable defines the debug level, when the GUI mode is used by the program.

By setting FGLGUIDEBUG to 1, the runtime system will display AUI protocol exchanges in the stderr output of the console running the program on the server.
The runtime system displays detailed information about user interface events that occur during program execution.

**Important:** This debug log is to be used in development context only. The output format can change in next product releases.

**Related concepts**
- **FGLGUI** on page 275
  Defines the user interface mode to be used by the program.

**FGLIMAGEPATH**
Defines the search paths for VM server image files.

**FGLIMAGEPATH basics**
The FGLIMAGEPATH environment variable is used by the runtime system, to find image resources on the server where the program executes, when the image name specified in the form element is not an URL that can be directly resolved and fetched by the front-end.

Image resources found through FGLIMAGEPATH will be transmitted to the front-end for display.

FGLIMAGEPATH defines a list of directories and/or image-to-font-glyph mapping files: If a path of FGLIMAGEPATH is a directory, it will be used for image file and font file lookup. If the element is a file name, it will be used as an image-to-font-glyph mapping file.

**FGLIMAGEPATH setting on mobile devices**
When executing on a mobile device, the environment variables must be defined with `mobile.environment` FGLPROFILE entries. The FGLAPPDIR and FGLDIR environment variables are automatically defined by the front-end component, and can be referenced with the `$FGLAPPDIR` and `$FGLDIR` placeholders, when defining FGLIMAGEPATH in FGLPROFILE:

```
mobile.environment.FGLIMAGEPATH
  = "$FGLAPPDIR/myimages:$FGLAPPDIR/icons/myimage2font.txt:$FGLDIR/lib/image2font.txt"
```

For more details about environment variable settings for mobile apps, see [Setting environment variables in FGLPROFILE (mobile)](page 263). [Setting environment variables in FGLPROFILE (mobile)] on page 263.

**Default behavior when FGLIMAGEPATH is not defined**
If the FGLIMAGEPATH environment variable is not defined, the runtime system will by default:

- Find image resource files in the current working directory where the BDL program executes.
  
  **Note:** When executing the app on an iOS device, instead of searching the current working directory, image resources are by default found in the `appdir` directory.

- Use `$FGLDIR/lib/image2font.txt` along with `$FGLDIR/lib/FontAwesome.ttf`, for image to font glyph mapping (to get default icons).

**Order of precedence in FGLIMAGEPATH**
It is possible to mix several image file directories with several image-to-font-glyph mapping files in FGLIMAGEPATH:

The list of mapping files and directories defines the order of precedence to resolve conflicts, when several image names can resolve to several image resources.

For example, if a form element defines an image as "smiley", and if FGLIMAGEPATH is defined as:

```
/opt/myapp/images:/opt/myapp/image2font.txt
```
If the /opt/myapp/images directory contains an image file "smiley.png", and the /opt/myapp/image2font.txt file contains a mapping for "smiley", the "smiley.png" file from /opt/myapp/images will be selected by the runtime system.

If FGLIMAGEPATH is defined as follows:

```
/opt/myapp/image2font.txt:/opt/myapp/images
```

The mapping for smiley to font glyph would take precedence.

**FGLIMAGEPATH syntax**

FGLIMAGEPATH must contain a list of paths, separated by the operating system specific path separator. The path separator is ";" on UNIX™ platforms and ";" on Windows® platforms.

For example, on UNIX:

```
$ export FGLIMAGEPATH="/var/myapp/myimages:$FGLDIR/lib/image2font.txt"
```

**Image-to-font-glyph mapping**

Image names can be mapped to font glyphs when at least one file path is specified in FGLIMAGEPATH. The runtime system distinguishes file paths (as image-to-font-glyph mapping files), from directory paths (as locations to file plain image files and font files).

**Important:** The directory and file name to the font file must be specified in FGLIMAGEPATH, except if the font file is located in the same directory as the mapping file.

A default mapping file ("image2font.txt") and its corresponding font file ("FontAwesome.ttf") are provided in $FGLDIR/lib. If FGLIMAGEPATH is not defined, the runtime system will use these files, to make the image name to font glyph mapping. If FGLIMAGEPATH is defined, the default mapping file will not be used. To get default Genero BDL icons, add $FGLDIR/lib/image2font.txt explicitly to your FGLIMAGEPATH path list.

**Important:** When providing your own customized font file, it must be a valid TTF file. For example, changing the file name is not sufficient to turn it into a different font: In order to produce a valid TTF file, use font management tools such as FontForge (http://fontforge.github.io/en-US/) or Fontello (http://fontello.com). Furthermore, to target Microsoft® Internet Explorer (version 11), you will need to patch the generated TTF file to remove embedding limitations from TrueType fonts, by setting the fsType field in the OS/2 table to zero. This modification can be done with freeware tools like ttembed.

The image-to-font-glyph mapping file must have the following syntax:

```
image-name=font-file:hexa-ordinal[:color-spec]
```

where:

1. **image-name** - is the name of the image to be mapped to a font character.
2. **font-file** - is the file name containing the font definitions.
3. **hexa-ordinal** - is the font glyph position in the font file, in hexadecimal notation.
4. **color-spec** - is the color to be used, in RGB hexadecimal format or as color alias as defined in presentation style colors. This field is optional: If not specified, the glyph will be displayed in a default color used by the front-end platform.

Lines starting with the # hash character are considered as comment lines and ignored.

For example:

```
# Common icons
camera=FontAwesome.ttf:f030
file=FontAwesome.ttf:f0f6:#8B0000
smiley=FontAwesome.ttf:f118:yellow
```
# Traffic lights

circle-red=FontAwesome.ttf:f111:red
circle-orange=FontAwesome.ttf:f111:orange
circle-green=FontAwesome.ttf:f111:green

FGLIMAGEPATH and gICAPI web components

For applications executing on a server and displaying on GDC/GMA/GMI front-ends in client/server mode (not through the GAS), the recommended solution is to locate gICAPI web component assets in `appdir/webcomponents`. Like image resources, the web component files will be automatically transferred to the front-end when connected in direct mode.

For backward compatibility, if the web component files are not located in the recommended directory, FGLIMAGEPATH can be used to define search paths for web component files. It is not recommended to use FGLIMAGEPATH to find web component files. For more details, see Deploying the gICAPI web component files on page 1947.

Note however that FGLIMAGEPATH must be used in direct mode, to find application image resources displayed inside a gICAPI web component. In such case, you need to add search paths for application images in FGLIMAGEPATH, and use the `ui.Interface.filenameToURI()` method to specify the image resource inside the web component. For more details, see Using image resources with the gICAPI web component on page 1952.

Related concepts
Providing the image resource on page 1149
There are several things you need to know about providing an image resource in a Genero program.

FGLLDPATH
Defines the search paths to load program modules.

The FGLLDPATH environment variable defines the search paths to load C extensions and `.42m` modules.

Note: The FGLLDPATH variable is used at link time and at run time.

A program can be composed by several p-code modules (`.42m`) and can use C extensions. When linking and when executing the program, the runtime system must known where to search for these modules. You can use the FGLLDPATH environment variable to define the search paths to load C extensions and p-code modules.

FGLLDPATH must contain a list of paths, separated by the operating system specific path separator. The path separator is `:` on UNIX™ platforms and `;` on Windows® platforms.

The directories are searched in the following order:

1. The current working directory.
2. The directory where the program file resides (the `.42m` module containing MAIN or the `.42r` program file).
3. A path defined in the FGLLDPATH environment variable.
4. The `$FGLDIR/lib` directory.

Note: FGLLDPATH is also used by the debugger to find program sources. For more details, see FGLSOURCEPATH on page 280.

Related concepts
FGLDIR on page 275
Defines the installation directory of Genero Business Development Language.

Importing modules on page 480
Use the IMPORT ... instruction to import BDL, C or Java external modules in the current module.

**FGLPROFILE**
Defines the configuration files to be used by the runtime system.

**Usage**
The FGLPROFILE environment variable defines a list of configuration files to be used by the runtime system.

Note: The runtime system always reads entries from the default configuration file located in $FGLDIR/etc/fglprofile, then the file(s) defined in the FGLPROFILE environment variable.

FGLPROFILE can define one unique configuration file, or a list of files to be loaded sequentially.

FGLPROFILE must contain a list of file paths, separated by the operating system specific path separator. The path separator is ":" on UNIX™ platforms and ";" on Windows® platforms.

Note: On mobile devices, you must deploy a file with the name "fglprofile" in the appdir directory. See FGLPROFILE for mobile apps on page 261 for more details.

For more details, see also The FGLPROFILE file(s) on page 255.

**Example**

On UNIX™ platforms:

```bash
$ FGLPROFILE="/opt/myapp/gui_settings.prf:/opt/myapp/sqldb_settings.prf"
$ export FGLPROFILE
```

On Windows® platforms:

```bash
C:\> set FGLPROFILE=C:\myapp\gui_settings.prf;C:\myapp\sqldb_settings.prf
```

**FGLRESOURCEPATH**
Defines search path for resource files.

**FGLRESOURCEPATH definition**
The FGLRESOURCEPATH environment variable is used to define the search paths for program resource files:

1. Form definition files (.42f),
2. Message files (.iem),
3. Action defaults files (.4ad),
4. Presentation styles files (.4st),
5. Start menu files (.4sm),
6. Toolbar files (.4tb),
7. Topmenu files (.4tm),
8. Localized strings files (.42s).

When the specified resource file is not an absolute path, the runtime system searches in directories in the following order:

1. The current working directory.
2. A path defined in the FGLRESOURCEPATH (or DBPATH) environment variable.
3. The $FGLDIR/lib directory.
4. The directory where the program file resides (the .42m module containing MAIN or the .42r program file).

The path separator is platform specific (";" on UNIX™ platforms and ";" on Windows® platforms).

Note: On mobile platforms, localize string files are found by default in the language sub directories of the app directory. For more details, see Loading localized strings at runtime on page 544.
FGLRESOURCEPATH versus DBPATH

For compatibility with Informix® 4GL, DBPATH is used by default to search for resource files such as form files and XML files used by the program.

However, DBPATH is also used by the Informix® database software to locate databases: Informix® Dynamic Server uses DBPATH to let you specify fallback servers if INFORMIXSERVER is not available, and former Informix® Standard Engine needs DBPATH to find .dbs database files.

This can be a problem when connecting from a machine where path format is not the same as on the remote database server: It is not possible to mix UNIX™ and DOS path formats in DBPATH.

To work around this Informix® limitation, FGLRESOURCEPATH can be used instead of DBPATH to specify the directories of program resource files. You are then free to define DBPATH as Informix® requires.

**Related concepts**

- **DBPATH** on page 272
  Defines the paths to search for Genero program resource files.

- **IBM Informix Dynamic Server** on page 658

FGLSERVER

Defines the graphical front-end for the application.

In GUI mode, FGLSERVER defines the host name and port of the graphical front-end the runtime system will connect to in order to display application forms.

If FGLSERVER is not defined, the runtime system (fglrn) assumes that the front-end executes on the same computer.

The values for the FGLSERVER environment variable must be specified with the following syntax:

```plaintext
{hostname|ip-address}[:server-num]
```

1. `hostname` is the name of a machine on the network.
2. `ip-address` is the IP V4 address (Ex: 10:0:0:105).
3. `server-num` identifies the front-end.

The `server-num` parameter defines the front-end server number (first is 0, second is 1, and so on). This defines implicitly the TCP port number the front-end is listening to, as an offset for the base port 6400. For example, `FGLSERVER=cobra:1` will use the TCP port 6401 (6400 + 1). This parameter is optional, when not specified, it defaults to zero (port 6400).

**Related concepts**

- **Genero user interface modes** on page 1109
  User interface modes allow you to adapt the application form rendering to different types of displays.

  - **Automatic front-end startup** on page 1121
  - **Front-end connection** on page 285
    To execute a Genero program with a graphical user interface, you need to specify the front-end (i.e. the graphical server) to the runtime system.

FGLSOURCEPATH

Defines the path to program source files.

The debugger needs to access the source files to display program code. By default, the current directory and the directories defined by FGLLDPATH are used to find source files.

The FGLSOURCEPATH environment variable is provided to distinguish execution directories (containing .42m files), from source directories (containing .4gl files), when the sources are not located in the same directory as the pcode files.
FGLSOURCEPATH must contain a list of paths, separated by the operating system specific path separator. The path separator is ":" on UNIX™ platforms and ";" on Windows™ platforms.

UNIX™ example:

```bash
$ FGLSOURCEPATH="/usr/app/source:/home/scott/sources"
$ export FGLSOURCEPATH
```

Windows™ example:

```cmd
C:\> set FGLSOURCEPATH=C:\app\sources;C:\scott\sources
```

**Related concepts**

**Integrated debugger** on page 2142

Describes the command-line debugger you can use to find bugs in your programs.

**FGLSQLDEBUG**

Defines the debug level for tracing SQL instructions.

If FGLSQLDEBUG is set to a value greater than zero, debug information is written to stderr for each SQL instruction executed by the program.

**Important**: Sensitive and personal data may be written to the output. Make sure that the log output is written to files that can only be read by application administrators.

FGLSQLDEBUG can be used on a production site in order to identify a problem related to SQL statements. However, the SQL debug log can produce a lot of data, slow down program execution and therefore must be used with care.

UNIX™ (shell) example:

```bash
$ FGLSQLDEBUG=1
$ export FGLSQLDEBUG
$ fglrun myprog 2>sqldbg.txt
```

The FGLSQLDEBUG environment variable can be set to different integer values, depending on the level of details you want to get from the runtime system and database driver.

**Tip**: Consider setting FGLSQLDEBUG to the highest level, to produce the maximum debug information.

**Table 118: Possible values for FGLSQLDEBUG**

<table>
<thead>
<tr>
<th>FGLSQLDEBUG</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Print debug message only when an SQL instruction produces an error.</td>
</tr>
<tr>
<td>0</td>
<td>No SQL debug message is produced.</td>
</tr>
<tr>
<td>1</td>
<td>Minimal debug information (SQL instruction, location, execution time).</td>
</tr>
<tr>
<td>2</td>
<td>Normal debug information (1 + error details).</td>
</tr>
<tr>
<td>3</td>
<td>Full debug information (1 + 2 + processing details, internals).</td>
</tr>
</tbody>
</table>

**Note**: The output format of FGLSQLDEBUG is for debug purpose only and may change in future product releases. Alternatively, you can use the `fgl_sqldebug()` function to set the SQL debug level by program.

**Related concepts**

**Debugging SQL statements** on page 603
The runtime system can display debug information for SQL statements executed by the program.

**FGLTRACE_FUNCTIONS**
Defines the list of functions to be followed by the program execution trace.

The FGLTRACE_FUNCTIONS environment variable defines the list of functions to be traced with the program execution trace option.

By default, the trace starts with the MAIN function and all functions of the program are traced.

In order to limit the trace to a given set of functions, define the FGLTRACE_FUNCTIONS environment variable with space-separated list of function names. The functions can be prefixed by a module name:

```
function-name
module-name.function-name
...
```

1. **module-name** is the name of a .42m module.
2. **function-name** is the name of a function.

**Important:** Unlike FGLTRACE_EXCLUDE, functions in FGLTRACE_FUNCTIONS can be specified with or without the module prefix. For example, if you want to include the check_order() function of the orders.4gl module, you can specify "orders.check_order" or "check_order" in FGLTRACE_FUNCTIONS.

For example, to enable the trace log in the "invoice_report_1" function (without specifying a module), and to enable the trace in the "add_customer" function, defined in the "custmod" module:

On UNIX™:

```
$ FGLTRACE_FUNCTIONS="invoice_report_1 custmod.add_customer"
$ export FGLTRACE_FUNCTIONS
```

On Windows™:

```
C:\> set FGLTRACE_FUNCTIONS=invoice_report_1 custmod.add_customer
```

**Note:** FGLTRACE_FUNCTIONS has a higher priority than FGLTRACE_EXCLUDE. The trace is enabled for a function listed in FGLTRACE_FUNCTIONS, when is it called from a function excluded by FGLTRACE_EXCLUDE.

**Related concepts**
- *Execution trace* on page 2171
  Print a function call stack of your program.

**FGLTRACE_EXCLUDE**
Defines the list of functions to be excluded from the program execution trace.

The FGLTRACE_EXCLUDE environment variable defines the list of patterns to identify functions (and class methods) that must be excluded from the call stack trace.

By default, all functions are traced, starting from the MAIN function, or from the functions listed in FGLTRACE_FUNCTIONS.

In order to exclude functions you don't want to trace, define the FGLTRACE_EXCLUDE environment variable with space-separated list of patterns:

```
exclude-pattern...
```

1. **exclude-pattern** is a string with wildcards like in a MATCHES expression:
   - The * wildcard represents 0 to n characters.
• The \? wildcard represent a single character.
• The \{ \} wildcards can be used to define a single character in the specified range (use a ^ starting caret for negation).

**Important:** Unlike FGLTRACE_FUNCTIONS, user functions in FGLTRACE_EXCLUDE must be specified with their module prefix. For example, if you want to exclude the check_order() function of the orders.4gl module, use the qualified name “orders.check_order” in FGLTRACE_EXCLUDE.

The FGLTRACE_EXCLUDE environment variable is typically used to exclude build-in functions and classes such as base.Array.*. Tracing build-in functions can produce a huge log and is not always relevant. For example, if the program uses the base.Array.getLength() method, the trace will report each call to the method, as in FOR i=1 TO arr.getLength().

**Tip:** To exclude all global built-in functions such as fgl_getenv(), use the <builtin>.* exclusion pattern.

For example, to exclude all methods from the build-in dynamic array class, from the util.JSON class, all om.* package calls and all functions of mymodule with the format debug_[0-9]???, define the variable as follows:

On UNIX™:

```
$ FGLTRACE_EXCLUDE="<builtin>.* base.Array.* util.JSON.* om.*
mymodule.debug_[0-9]???
" $ export FGLTRACE_EXCLUDE
```

On Windows®:

```
C:\> set FGLTRACE_EXCLUDE=<builtin>.* base.Array.* util.JSON.* om.*
mymodule.debug_[0-9]???
```

**Related concepts**

**Execution trace** on page 2171
Print a function call stack of your program.

**FGLWRTUMASK**
Defines the umask to be used by the license manager.

The FGLWRTUMASK environment variable is used by the fglWrt license manager to create the $FGLDIR/lock directory.

This variable defines the umask to create the $FGLDIR/lock directory.

The default is 000, which creates a directory with rwxrwxrwx file permissions.

**Related concepts**

**Installation** on page 37
This chapter contains installation and setup instructions.

**FGLWSDEBUG**
The FGLWSDEBUG environment variable enables web services library debugging.

When FGLWSDEBUG environment variable is set to 1, debug information is written to stderr for each operation done by the web services library.

**Important:** Sensitive and personal data may be written to the output. Make sure that the log output is written to files that can only be read by application administrators.

UNIX™ (shell) example:

```
$ FGLWSDEBUG=1
$ export FGLWSDEBUG
$ fglrun myprog 2>wsdbg.txt
```
Note: This debug log is to be used in development context only. The output format can change in future product releases.

Table 119: FGLWSDEBUG variable values

<table>
<thead>
<tr>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No data displayed; debug turned off.</td>
</tr>
<tr>
<td>1</td>
<td>Display socket errors.</td>
</tr>
<tr>
<td>2</td>
<td>Display HTTP bodies of incoming and outgoing requests (the XML content)</td>
</tr>
<tr>
<td>3</td>
<td>Display all information about incoming and outgoing requests (HTTP headers + HTTP bodies)</td>
</tr>
</tbody>
</table>

Related concepts
Debugging on page 3276
Turn on the debug mode to log the data sent or received by your Web service application.

GMIDIR
Defines the installation directory of Genero Mobile for iOS.
The GMIDIR environment variable defines the installation directory of the Genero Mobile for iOS archive, used to build iOS apps with the gmibuildtool.
When building apps with gmibuildtool, GMIDIR is used to find the GMI libraries and resource files.
By default, if GMIDIR is not defined, the location for GMI libraries is found from the location of the gmibuildtool. For example, if gmibuildtool is found in /opt/genero/gmi-1.20/bin/gmibuildtool, GMIDIR will be defined as /opt/genero/gmi-1.20.

Related concepts
Building iOS apps with Genero on page 3606
Genero provides a command-line tool to build applications for iOS devices.

INFORMIXTERM
Defines terminal control library to be used.
The INFORMIXTERM environment variable indicates what terminal capabilities database must be used by the runtime system when running a program in TUI mode on a dumb terminal.
Possible values of INFORMIXTERM are terminfo and termcap. If the variable is not set, it defaults to termcap.
When set to termcap (the default), the runtime system reads terminal capabilities from the file defined by the TERMCAP environment variable.
When set to terminfo, the runtime system reads terminal capabilities from the terminfo database of the system (ncurses).

Related concepts
TERM on page 264
Defines the type of terminal on UNIX™ platforms.
TERMCAP on page 265
Defines the *termcap* terminal capabilities database on UNIX™ platforms.

**Front-end connection**

To execute a Genero program with a graphical user interface, you need to specify the front-end (i.e. the graphical server) to the runtime system.

In development mode, the target front-end is defined with the `FGLSERVER` on page 280 environment variable. However, there are various technologies to render a Genero application, depending on the front-end platform (PC, mobile device, web browser).

Details about front-end configuration for the runtime system can be found in User interface basics on page 1105.

**Database server connections**

Before running a Genero program using a database, you must configure the connection parameters to access the database server.

There are different solutions to define database connection parameters, consider using and indirect database connection configuration, by using an abstract database name in programs, and define the real database source, driver with FGLPROFILE entries.

The database configuration details can be found the SQL support chapter of this manual.

**Related concepts**

*SQL support* on page 590

These topics cover SQL support in the Genero Business Development Language.

**Language basics**

These topics cover the basics for the Genero Business Development Language (BDL)

**Syntax features**

Genero BDL is an English-like programming language, easy to write and read.

**Lettercase insensitivity**

Genero BDL is case insensitive (with some exceptions).

Genero BDL does not make distinction between uppercase and lowercase letters, except within quoted strings.

You can mix uppercase and lowercase letters in the identifiers that you assign to language entities, but any uppercase letters in identifiers are automatically shifted to lowercase during compilation:

```
INPUT BY NAME ...
ON ACTION PrintReport     -- becomes "printreport" in 42m pcode
```

It is strongly recommended that you define a naming convention for your projects. For example, you can use underscore notation (*get_user_name*). If you plan to use the Java notation (*getUserName*), do not forget that Genero BDL is case insensitive (*getusername* is the same identifier as *getUserName*).
Tip: For better readability, and to be consistent with SQL syntax conventions, consider writing BDL language keywords in UPPERCASE, and other language elements like identifiers in lowercase:

```
INPUT BY NAME cust_rec.* ATTRIBUTES (UNBUFFERED)
```

Exceptions regarding case-sensitivity:

1. Genero BDL interfaces with Java classes and objects. Java symbols are case-sensitive.
2. The module name (and filename) used with `IMPORT FGL module` on page 481 `IMPORT FGL` is case sensitive.
3. Resource files using XML such as `.4ad` files are case-sensitive (regarding action names for example).

**Whitespace separators**

Whitespace characters are used to separate language elements.

Genero BDL is free-form, like C or Pascal, and generally ignores TAB characters, LINEFEED characters, comments, and extra blank spaces between instructions and language elements.

You can freely use these whitespace characters to enhance the readability of your source code.

Blank (ASCII 32) characters act as delimiters in some contexts. Blank spaces must separate successive keywords or identifiers, but cannot appear within a keyword or identifier.

**Note:** Pairs of double (" ) or single (’ ) quotation marks must delimit any character string that contains a blank space (ASCII 32) or other whitespace character, such as LINEFEED or RETURN.

**Quotation marks**

String literals need to be delimited with single or double quotation marks.

In the Genero BDL language, string literals are delimited by single (’) or double (" ) quotation marks:

```
'Valid character string'
"Another valid character string"
```

Do not mix double and single quotation marks as delimiters of the same string. The following is not a valid character string:

```
'Not A valid character string"
```

To include literal quotation marks within a quoted string, precede each literal quotation mark with the backslash (\), or else enclose the string between a pair of the opposite type of quotation marks:

```
MAIN
    DISPLAY "Type 'Y' if you want to reformat your disk."
    DISPLAY 'Type "Y" if you want to reformat your disk.'
    DISPLAY 'Type \"Y\" if you want to reformat your disk.'
END MAIN
```

A string literal can be written on multiple lines. The compiler merges lines by removing the newline character.

In the SQL language, the standard specifications recommend that you use single quotes for string literals and double quotes for database object identifiers like table or column names. When accessing a non-Informix database, double quotation marks might not be recognized as database object name delimiters. As a general rule, use single quoted string literals in SQL statements, and use non-quoted, lowercase database object identifiers.

**Related concepts**

Text literals on page 326
Text literals define a character string in an expression.

**Escape symbol**

Backslash ( \ ) is the escape character of Genero BDL.

The Genero BDL compiler treats a backslash ( \ ) as the default escape symbol, and treats the immediately following symbol as a literal, except for special characters such as \r or \t.

To specify anything that includes a literal backslash, enter double ( \\ ) backslashes wherever a single backslash is required. Similarly, use \\\\ to represent a literal double backslash.

```plaintext
MAIN
    DISPLAY "\a"   -- displays a
    DISPLAY "\r"   -- displays CR
    DISPLAY "\n"   -- displays NL
    DISPLAY "\ta"  -- displays <tab>a
    DISPLAY "\\"  -- displays \n    DISPLAY "\\\\" -- displays \n
END MAIN
```

**Related concepts**

Text literals on page 326

Text literals define a character string in an expression.

**Statement terminator**

The semicolon (;) is optional statement terminator in Genero BDL.

Genero BDL is a language that does not require a statement terminator like in C/C++ or Java.

However, you can use the semicolon ( ; ) as a statement terminator in some cases.

For example:

```plaintext
MAIN
    DISPLAY "Hello, World"  DISPLAY "Hello, World"
    DISPLAY "Hello, World"; DISPLAY "Hello, World"
END MAIN
```

The semicolon statement terminator is sometimes mandatory, when nesting instruction blocks like in the following example:

```plaintext
INPUT BY NAME ...
    ON ACTION other_input
        INPUT BY NAME ...
    END INPUT
```

**Source comments**

The --, # and { } characters can be used to add source comments.

**Comment markers**

A source comment is text in the source code to assist human readers, but which BDL ignores.

A source comment can be specified by any of the following:

- A pair of minus signs ( -- ) indicates a comment that terminates at the end of the current line. This comment indicator conforms to the ANSI standard for SQL.
- The hash character ( # ) indicates a comment that terminates at the end of the current line.
• A starting left-brace ({ }) starts a comment. It can be followed by any character (including line breaks). The comment ends when the closing right-brace (}) symbol is found.

```
MAIN
   -- DISPLAY "This line will be ignored."
# DISPLAY "This line will be ignored."
{ DISPLAY "This line will be ignored."
DISPLAY "This line will be ignored."
}
DISPLAY "Hello, World"
END MAIN
```

Notes:
• Within a quoted string, the compiler interprets comment indicators as literal characters, rather than as comment indicators.
• You cannot use curly brackets ({ }) to nest comments within comments.
• Comments cannot appear in the form section defining a layout grid, such as SCREEN, TABLE, TREE, or GRID.
• The # symbol cannot indicate comments in an SQL statement block nor in the text of a prepared statement.
• You cannot specify consecutive minus signs (-- ) in arithmetic expressions, as BDL interprets what follows as a comment. Instead, use a blank space or parentheses to separate consecutive arithmetic minus signs.

**I4GL/BDL language specific comments**

The --# specific comment indicator is used to distinguish Informix® 4GL code from Genero BDL code.

When writing a comment with --#, the remaining line content will be taken into account by the Genero BDL compiler, but it will be ignored by the I4GL compiler.

This conditional code compilation technique can be inverted by enclosing code blocks between --#{ and --#} comments:

```
MAIN
   --# DISPLAY "Ignored by I4GL, but compiled with BDL."
   --#{
      DISPLAY "Ignored by BDL, but compiled with I4GL."
   }
END MAIN
```

To summarize:
• Code lines starting with --# are compiled with Genero BDL, but ignored by Informix® 4GL.
• Code blocks surrounded with --#{ and --#} are compiled with Informix® 4GL, but ignored by Genero BDL.

**Code beautifier control comments**

Specific comments can be used to skip the formatting of a group of lines, by using the fgl-format off/on keywords:

```
# fgl-format off
DEFINE    cnt    INTEGER
DEFINE    ratio  DECIMAL(10,2)
# fgl-format on
```

For more details, see Source code beautifier on page 2174.
Identifiers

A Genero BDL identifier is a sequence of characters used to identify a program entity.

An identifier must conform to the following rules:

• It must include at least one character, without any limitation in size.
• Only ASCII letters, digits, and underscore (_) symbols are valid.
• Blanks, hyphens, and other non-alphanumeric characters are not allowed.
• The initial character must be a letter or an underscore.
• Common identifiers are not case sensitive, so my_Var and MY_VaR both denote the same identifier. However, in some cases, identifiers are case sensitive (like action names in the AUI tree). It is recommended to always write identifiers in lower case to avoid mistakes.

Within non-English locales, BDL identifiers can include non-ASCII characters in identifiers, if those characters are defined in the code set of the current locale. In multibyte East Asian locales that support languages whose written form is not alphabet-based (such as Chinese, Japanese, or Korean), an identifier does not need to begin with a letter. It is however recommended to program in ASCII.

Related concepts

User interface basics on page 1105
This section introduces to the foundation of the Genero user interface.

Localization on page 512
Localization support allows you to implement programs that follow specific language and cultural rules.

Preprocessor directives

Preprocessor directives can be used in Genero BDL sources.

Genero BDL supports preprocessing instructions, which allow you to write macros and conditional compilation rules as in the following example:

```gdl
&include "myheader.4gl"
FUNCTION debug( msg )
  DEFINE msg STRING
  &ifdef DEBUG
  DISPLAY msg
  &endif
END FUNCTION
```

Note: Use the preprocessor with care, and only when there is no native language solution. Do not overcrowd your source code with preprocessing directives, that would make the code unreadable and unmaintainable.

Related concepts

Source preprocessor on page 2132
A typical preprocessor like in the C language.

Primitive Data types

Selecting the correct data type assists you in the input, storage, and display of your data.

Related concepts

Type conversions on page 311
Explains primitive data type conversion rules of the language.

Variables on page 390
Explains how to define program variables.

Compiling source files on page 2106
BIGINT

The BIGINT data type is used for storing very large whole numbers.

Syntax

```
BIGINT
```

Usage

The storage of BIGINT variables is based on 8 bytes of signed data ( = 64 bits ).
BIGINT variables can be initialized with integer literals:

```
MAIN
    DEFINE i BIGINT
    LET i = 9223372036854775600
    DISPLAY i
END MAIN
```

When assigning a whole number that exceeds the BIGINT range, the overflow error -1284 will be raised.

BIGINT variables are initialized to zero in functions, modules and globals.

Data type conversion can be controlled by catching the runtime exceptions. For more details, see Handling type conversion errors on page 317.

Related concepts

INTEGER on page 300
The INTEGER data type is used for storing large whole numbers.

SMALLINT on page 304
The SMALLINT data type is used for storing small whole numbers.

TINYINT on page 306
The TINYINT data type is used for storing very small whole numbers.

BYTE

The BYTE data type stores any type of binary data, such as images or sounds.

Syntax

```
BYTE
```

Usage

A BYTE or TEXT variable is a handle for a large object (LOB), that is stored in a file or in memory. Such data type is a complex type that cannot be used like INTEGER or CHAR basic types: It is designed to handle a large amount of data and has different semantics as simple types. The main difference with simple data types, is the fact that you must specify the storage with the LOCATE instruction, before using BYTE and TEXT variables.

The maximum size of data that can be handled by BYTE and TEXT variable is theoretically $2^{31}$ bytes (~2.14 Gigabytes), but the practical limit depends on the disk or memory resources available to the process.

BYTE and TEXT variable must be initialized with the LOCATE instruction before usage.
**Note:** APIs such as the util.JSON class will automatically allocate TEXT/BYTE variables in memory if these are not located, when converting a JSON string to an FGL variable with the `util.JSON.parse()` method.

The LOCATE instruction basically defines where the large data object has to be stored (in a named file, in a temporary file, or in memory). This instruction will actually allow you to fetch a LOB into memory or into a file, or insert a LOB from memory or from a file into the database. When located in a temporary file (IN FILE), the temp directory can be defined by the DBTEMP environment variable.

```
DEFINE t TEXT
LET t = "aaaaa" -- invalid, t is not located
LOCATE t IN MEMORY
LET t = "aaaaa" -- valid, now t is located in memory
```

**Important:**

TEXT/BYTE variables are not comparable: Comparing TEXT/BYTE variable does not result in an error, but the result of the comparison operator will always be NULL, which is equivalent to FALSE in an IF statement:

```
DEFINE t1, t2 TEXT
IF t1 == t2 THEN -- Always NULL/FALSE
    ...
END IF
```

With BYTE and TEXT types, you can insert/update/fetch large objects of the database. The native database type to be used depends on the type of database server. After defining the storage with LOCATE, load/assign its value, and then you can use it directly in the SQL statements, or fetch data from LOB columns of the database, like simple data types:

```
DEFINE t1, t2 TEXT
...
CREATE TABLE mytable ( id INT, data TEXT )
...
LOCATE t1 IN MEMORY
CALL t1.readFile("lob.4gl")
INSERT INTO mytable VALUES ( 1, t1 )
LOCATE t2 IN FILE
SELECT data INTO t2 FROM mytable WHERE id=1
...
```

BYTE and TEXT types implement the readFile() and writeFile() methods to read/write the whole large object data from/to files. These methods can be used to easily interface with other software components:

```
DEFINE t TEXT
LOCATE t IN MEMORY
CALL t.readFile("orig.txt")
CALL t.writeFile("copy.txt")
```

For more details about LOB types methods, see BYTE data type as class on page 2378 and TEXT data type as class on page 2391.

When initializing a BYTE or TEXT variable to NULL (INITIALIZE var TO NULL), if the variable is located in a file, the file is truncated (file size will be zero). If the variable is located in memory, the data in memory will be truncated. A subsequent usage of the variable (for example, FETCH INTO or LET assignment) is still possible:

```
DEFINE b BYTE
LOCATE b IN FILE "picture.png"
INITIALIZE b TO NULL
-- The file "picture.png" is now empty.
```

Resources allocated to a BYTE or TEXT variable can be deallocated with the FREE instruction.
**Note:** When the TEXT or BYTE variable is already located, a new LOCATE will free the allocated resource: If the prior LOCATE was using the IN FILE clause, the temporary file is dropped, if the prior LOCATE was using IN MEMORY, the memory is freed.

A FREE will remove the file if the LOB variable is located in a (named or temporary) file. When located in memory, the FREE instruction will deallocate the memory. After freeing the resources of a LOB variable, it must be re-located with a LOCATE instruction:

```
DEFINE b BYTE
LOCATE b IN FILE
CALL b.readFile("picture.png") -- ok
FREE b
CALL b.readFile("picture.png") -- Invalid, b is not located.
LOCATE b IN MEMORY
CALL b.readFile("picture.png") -- ok
```

**Important:**

TEXT and BYTE are reference types. This implies that assigning two variables (LET, passing a variable as parameter to a function, returning a result from a function) does not copy the value (Only the handle is copied. As a result, modifying the data with a TEXT/BYTE variable assigned from another TEXT/BYTE variable will in fact modify the same LOB data. Furthermore, the storage resource (file or memory) that was used by the assigned variable becomes unreferenced and is lost:

```
DEFINE b1, b2 BYTE -- Could be TEXT: same behavior
LOCATE b1 IN FILE "mydata" -- reference file directly
LOCATE b2 IN MEMORY -- use memory instead of file
CALL b2.readFile("mydata") -- read file content into memory
# FREE b2 -- this should be done to free memory before LET
LET b2 = b1 -- Now b2 points directly to the file (like b1)
INITIALIZE b1 TO NULL -- truncates reference file
DISPLAY IIF( b2 IS NULL, "b2 is null", "b2 is not null")
-- Displays "b2 is null"
```

In the next (invalid) code example, we try to save the value of the img BYTE variable in a temporary variable (tmp), with the typical programming pattern to save the value before modification. In fact the LET tmp=img assignment does not copy the data of the LOB like for simple data types (STRING, VARCHAR, DECIMAL), only the reference (i.e. handle) to the data is copied:

```
-- WARNING: THIS IS AN INVALID CODE EXAMPLE
DEFINE img, tmp BYTE
LOCATE img IN MEMORY
CALL img.readFile("picture1.png")
LOCATE tmp IN MEMORY
LET tmp = img -- Expecting to save the current data, but now
-- both variables reference the same data...
CALL img.readFile("picture2.png")
LET img = tmp -- Does not restore the old value: Same data.
```

If you need to clone a large object, use the `writeFile() / readFile()` methods.

**Related concepts**

Runtime images on page 1154
Explains how to display pictures at runtime.

TEXT on page 307
The TEXT data type stores large text data.

**BOOLEAN**

The BOOLEAN data type stores a logical value, TRUE or FALSE.

**Syntax**

```
BOOLEAN
```

**Usage**

Boolean data types have two possible values: TRUE (integer 1) and FALSE (integer 0).

Variables of this type can be used to store the result of a boolean expression:

```
DEFINE result BOOLEAN
LET result = ( length("abcdef") > 0 )
```

Data type conversion can be controlled by catching the runtime exceptions. For more details, see Handling type conversion errors on page 317.

Boolean variables are typically used to store the result of a boolean expression:

```
FUNCTION checkOrderStatus( cid )
  DEFINE oid INT, b BOOLEAN
  LET b = ( isValid(oid) AND isStored(oid) )
  IF NOT b THEN
    ERROR "The order is not ready."
  END IF
END FUNCTION
```

Note that the database vendor specific implementation of the boolean SQL type may not correspond exactly to the Genero BOOLEAN type. For example, IBM® Informix® SQL boolean type accepts the 't' and 'f' values, while the BOOLEAN Genero type expects 0/FALSE and 1/TRUE integer values only. You can however use a BOOLEAN variable in SQL statements: IBM® Informix® will handle the conversion, and for other databases, the db drivers handle the conversion. Note also that the TRUE/FALSE constants are Genero language constants: The SQL syntax of the database may not support these keywords, for example in an statement such as `INSERT INTO mytable (key,bcol) VALUES (455,TRUE)`. For more details, see SQL portability on page 607.

**CHAR(size)**

The CHAR data type is a fixed-length character string data type.

**Syntax**

```
CHARACTER [size]
```

1. size defines the maximum length of the character string, in byte or char units (depending on the character length semantics)
2. The maximum size of a CHAR type is 65534.
3. If no size is specified, it defaults to 1.

**Usage**

The CHAR type is typically used to store fixed-length character strings such as short codes (XB124), phone numbers (650-23-2345), vehicle identification numbers.

CHAR and CHARACTER are synonyms.
The size can be expressed in bytes or characters, depending on the length semantics used in programs. For more details about character length semantics, see Length semantics settings on page 521.

When size is not specified, the default length is 1.

CHAR variables are initialized to NULL in functions, modules and globals.

Text literals can be assigned to character string variables:

```plaintext
MAIN
    DEFINE c CHAR(10)
    LET c = "abcdef"
END MAIN
```

When assigning a non-NULL value, CHAR variables are always blank-padded:

```plaintext
MAIN
    DEFINE c CHAR(10)
    LET c = "abcdef"
    DISPLAY "[", c ,"]"   -- displays [abcdef    ]
END MAIN
```

Trailing blanks of a CHAR value are not significant in comparisons:

```plaintext
MAIN
    DEFINE c CHAR(5)
    LET c = "abc"
    IF c == "abc" THEN    -- evaluates to TRUE
        DISPLAY "equals"
    END IF
END MAIN
```

Numeric and date-time values can be directly assigned the character strings:

```plaintext
MAIN
    DEFINE c CHAR(50), da DATE, dec DECIMAL(10,2)
    LET da = TODAY
    LET dec = 345.12
    LET c = da, " : ", dec
END MAIN
```

When you insert character data from CHAR variables into CHAR columns in a database table, the column-value is blank-padded to the size of the column. Likewise, when you fetch CHAR column values into CHAR variables, the program variable is blank-padded to the size of the variable.

```plaintext
MAIN
    DEFINE c CHAR(10)
    DATABASE test1
    CREATE TABLE table1 ( k INT, x CHAR(10) )
    LET c = "abc"
    INSERT INTO table1 VALUES ( 1, c )
    SELECT x INTO c FROM table1 WHERE k = 1
    DISPLAY "[", vc ,"]"     -- displays [abc  ]
END MAIN
```

In SQL statements, the behavior of the comparison operators when using CHAR values may vary from one database to the other. However, most database engines ignore trailing blanks when computing CHAR values. For more details, see SQL portability on page 607.

Character string manipulation with CHAR, VARCHAR and STRING types can have a cost when accessing parts of large strings, when using UTF-8 with char length semantics. Consider using base.StringBuffer objects when doing heavy string manipulations. For more details, see Manipulating character strings on page 588.
Related concepts

**VARCHAR(size)** on page 309
The **VARCHAR** data type is a variable-length character string data type, with a maximum size.

**STRING** on page 305
The **STRING** data type is a variable-length, dynamically allocated character string data type, without limitation.

**DATE**
The **DATE** data type stores calendar dates with a Year/Month/Day representation.

**Syntax**

```plaintext
DATE
```

**Usage**
Storage of **DATE** variables is based on a 4 byte integer representing the number of days since 1899/12/31.
The value range is from 0001-01-1 (-693594) to 9999-12-31 (2958464).
**DATE** variables are initialized to zero (=1899/12/31) in functions, modules and globals.
Several built-in functions and operators specific to the **DATE** type are available, such as **MDY()** and **TODAY**. For more details, see [Date and time operators](#) on page 367.
Data type conversions, input and display of **DATE** values are ruled by environment settings, such as the **DBDATE** and **DBCENTURY** enviroment variables. Dates can be formatted with the **USING** operator. For more details, see [Formatting DATE values](#) on page 321.

**Note:** As date-to-string conversion is based on an environment settings, it is not recommended that you hard code strings representing dates:

```plaintext
LET date_var = "24/12/1998"      -- DBDATE dependant code
LET date_var = MDY(12,24,1998)   -- Portable code
```

To add or substract a given number of days to a **DATE**, simply use a + or − arithmetic operator followed by an integer expression representing a number of days:

```plaintext
MAIN
  DEFINE d DATE
  LET d = TODAY
  LET d = d + 10    -- Add 10 days
  LET d = d - 20    -- Substract 20 days
  DISPLAY "d = ", d USING "yyyy-mm-dd"
END MAIN
```

The difference of two dates returns the number of days:

```plaintext
MAIN
  DEFINE d1, d2 DATE
  LET d1 = MDY(12,24,1998)
  LET d2 = MDY(5,11,2010)
  DISPLAY "d2 - d1 = ", (d2-d1)
END MAIN
```

**DATE** values can be converted directly from/to **DATETIME** values:

```plaintext
MAIN
  DEFINE d DATE,
        dt DATETIME YEAR TO FRACTION(3)
```
LET d = TODAY
LET dt = d; DISPLAY "dt = ", dt
LET dt = CURRENT
LET d = dt; DISPLAY "d = ", d
END MAIN

In order to add or substract a number of months to a DATE, use the UNITS operator:

MAIN
DEFINE d0, d date
LET d0 = MDY(01, 31, 2015)
LET d = d0 + 1 UNITS MONTH; DISPLAY d
LET d = d0 - 1 UNITS MONTH; DISPLAY d
LET d = d0 - 2 UNITS MONTH; DISPLAY d
END MAIN

Note: In fact, the UNITS operator will produce an INTERVAL. Then the DATE value is converted to a DATETIME, to add or substract the INTERVAL value. Finally the DATETIME is converted to a DATE, in order to assign the result to the target variable.

Related concepts
DATETIME qual1 TO qual2 on page 296
The DATETIME data type stores date and time data with time units from the year to fractions of a second.

DATETIME qual1 TO qual2
The DATETIME data type stores date and time data with time units from the year to fractions of a second.

Syntax

```
DATETIME YEAR TO FRACTION [ ( scale ) ]
DATETIME YEAR TO SECOND
DATETIME YEAR TO MINUTE
DATETIME YEAR TO HOUR
DATETIME YEAR TO DAY
DATETIME YEAR TO MONTH
DATETIME YEAR TO YEAR
DATETIME MONTH TO FRACTION [ ( scale ) ]
DATETIME MONTH TO SECOND
DATETIME MONTH TO MINUTE
DATETIME MONTH TO HOUR
DATETIME MONTH TO DAY
DATETIME MONTH TO MONTH
DATETIME DAY TO FRACTION [ ( scale ) ]
DATETIME DAY TO SECOND
DATETIME DAY TO MINUTE
DATETIME DAY TO HOUR
DATETIME DAY TO DAY
DATETIME HOUR TO FRACTION [ ( scale ) ]
DATETIME HOUR TO SECOND
DATETIME HOUR TO MINUTE
DATETIME HOUR TO HOUR
DATETIME MINUTE TO FRACTION [ ( scale ) ]
DATETIME MINUTE TO SECOND
DATETIME MINUTE TO MINUTE
DATETIME SECOND TO FRACTION [ ( scale ) ]
DATETIME SECOND TO SECOND
DATETIME FRACTION TO FRACTION [ ( scale ) ]
```

1. scale defines the scale of the fractional part, it can be 1, 2, 3, 4 or 5.
Usage
The DATETIME data type stores an instance in time, expressed as a calendar date and time-of-day.

The qualifiers following the DATETIME keyword define the precision of the DATETIME type. While many sort of datetime types can be defined with all possible qualifier combinations, only a limited set of DATETIME types are typical used in applications:

- DATETIME HOUR TO MINUTE, DATETIME HOUR TO SECOND, DATETIME HOUR TO FRACTION(scale): To hold a time value.
- DATETIME YEAR TO MINUTE, DATETIME YEAR TO SECOND, DATETIME YEAR TO FRACTION(scale): To hold a date with time value.

DATETIME YEAR TO DAY is equivalent to DATE, consider used DATE instead.

The year of a DATETIME can range from 1 to 9999.

When the FRACTION qualifier is specified without a precision, the precision defaults to 3.

DATETIME arithmetic is based on the INTERVAL data type, and can be combined with DATE values:

<table>
<thead>
<tr>
<th>Left Operand Type</th>
<th>Operator</th>
<th>Right Operand Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATETIME</td>
<td>-</td>
<td>DATETIME</td>
<td>INTERVAL</td>
</tr>
<tr>
<td>DATETIME</td>
<td>-</td>
<td>DATE</td>
<td>INTERVAL</td>
</tr>
<tr>
<td>DATETIME</td>
<td>-</td>
<td>INTERVAL</td>
<td>DATETIME</td>
</tr>
<tr>
<td>DATETIME</td>
<td>+</td>
<td>INTERVAL</td>
<td>DATETIME</td>
</tr>
</tbody>
</table>

DATETIME variables are initialized to NULL in functions, modules and globals.

The CURRENT operator provides current system date/time:

DEFINE dt DATETIME YEAR TO SECOND
LET dt = CURRENT

DATETIME variables can be assigned with datetime literals, by using the DATETIME() q1 TO q2 notation:

DEFINE dt DATETIME YEAR TO SECOND
LET dt = DATETIME(2014-02-21 13:45:34) YEAR TO SECOND

DATETIME variables can be assigned from string literals, by using the format YYYY-MM-DD hh:mm:ss.ffffff, or the ISO 8601 format sub-set (with the T separator between the date and time part, and with optional +/-nn UTC indicator or timezone offset):

DEFINE dt DATETIME YEAR TO FRACTION(5)
LET dt = "2012-10-05 11:34:56.99999"
LET dt = "2012-10-05T11:34:56.99999+02:00"

When converting a DATETIME to a string, the format YYYY-MM-DD hh:mm:ss.ffffff is used.

Data type conversion can be controlled by catching the runtime exceptions. For more details, see Handling type conversion errors on page 317.

A DATETIME value can be converted to a different DATETIME (or DATE) with a different precision by using the EXTEND() operator:

MAIN
DEFINE dt1 DATETIME YEAR TO MONTH
DEFINE dt2 DATETIME YEAR TO FRACTION(5)
Datetime conversion functions are provided in the `util.Datetime` class, for example to convert local datetime to UTC datetime values:

```lnl
IMPORT util
MAIN
    DEFINE dt DATETIME YEAR TO FRACTION(5)
    LET dt = "2012-10-05 11:34:56.99999"
    DISPLAY util.Datetime.toUTC( dt )
END MAIN
```

Related concepts

**DATE** on page 295
The `DATE` data type stores calendar dates with a Year/Month/Day representation.

**INTERVAL qual1 TO qual2** on page 301
The `INTERVAL` data type stores spans of time as Year/Month or Day/Hour/Minute/Second/Fraction units.

**DECIMAL(p,s)**
The `DECIMAL` data type is provided to handle large numeric values with exact decimal storage.

Syntax

```
DECIMAL [ ( precision[,scale] ) ]
```

1. `precision` defines the number of significant digits (limit is 32, default is 16).
2. `scale` defines the number of digits to the right of the decimal point.
3. When no `scale` is specified, the data type defines a floating point number.
4. When no `(precision, scale)` is specified, it defaults to `DECIMAL(16)`.

Usage

Use the `DECIMAL` data type when you need to store values that have a fixed number of digits on the right and left of the decimal point (`DECIMAL(p,s)`), or to store a floating point decimal with an exact number of significant digits (`DECIMAL(p)`).

`DEC`, `DECIMAL` and `NUMERIC` are synonyms.

`DECIMAL` variables are initialized to `NULL` in functions, modules and globals.

When using `DECIMAL(p,s)` with a precision and scale, you define a decimal for fixed point arithmetic, with `p` significant digits and `s` digits on the right of the decimal point. For example, `DECIMAL(8,2)` can hold the value `123456.78` (8 (p) = 6 digits on the left + 2 (s) digits on the right of the decimal point).

When using `DECIMAL(p)` with a precision but no scale, you define a floating-point number with `p` significant digits. For example, `DECIMAL(8)` can store `12345678`, as well as `0.12345678`.

**Note:** In most database implementations, the decimal data type always has a fixed number of decimal digits. Use `DECIMAL` types with precision and scale to implement portable code, and avoid mistakes if default sizes apply when precisions and/or scale are omitted in SQL statements. For example, with Oracle®, a `NUMBER(p)` is equivalent to a `DECIMAL(p,0)` in BDL, not `DECIMAL(p)`.

When using `DECIMAL` without a precision and scale, it defaults to `DECIMAL(16)`, a floating-point number with a precision of 16 digits.
DEFINE d2 DECIMAL(10,3)
LET d1 = 1234.4567
LET d2 = d1 / 3 -- Rounds decimals to 3 digits
DISPLAY d1, d2
END MAIN

DECLARATIV values can be converted to strings based on the DBFORMAT (or DBMONEY) environment variable (defines the decimal separator) setting.

Value ranges

The largest absolute value that a DECIMAL (p, s) can store without errors is 10^{p-s} - 10^s. The stored value can have up to 30 significant decimal digits in its fractional part, or up to 32 digits to the left of the decimal point.

When using DECIMAL (p, s) the range of values is defined by the p, the number of significant digits. For example, a variable defined as DECIMAL (5, 3) can store values in the range -99.999 to 99.999. The smallest positive non zero value is 0.001.

When using DECIMAL (p) the magnitude can range from -N*10^{-124} to N*10^{124}, where N can have up to p significant digits and be 0<N<10. For example, a variable defined as DECIMAL (5) can store values in the range -9.9999e-124 to 9.9999e+124. The smallest positive non zero value is 9.9999e-130.

Exceptions

When the default exception handler is used, if you try to assign a value larger than the decimal definition (for example, 12345.45 into DECIMAL (4, 2)), no out of range error occurs, and the variable is assigned with NULL. If WHENEVER ANY ERROR is used, it raises error -1226. If you do not use WHENEVER ANY ERROR, the STATUS variable is not set to -1226.

Data type conversion can be controlled by catching the runtime exceptions. For more details, see Handling type conversion errors on page 317.

Computation and rounding rules

When computing or converting decimal values, the "round half away from zero" rule will apply: If the fraction of the value v is exactly 0.5, then r = v + 0.5 if v is positive, and r = v - 0.5 if v is negative. For example, when the result must be rounded to a whole number, 23.5 gets rounded to 24, and -23.5 gets rounded to -24.

In the next example, the division result of 11 / 3 gives the infinite decimal value 3.666666... (with an infinite decimal part). However, this value cannot be stored in a fixed point decimal type. When stored in a DECIMAL (10, 2), the value will be rounded to 3.67, and when multiplying 3.67 by 3, the result will be 11.01, instead of 11:

MAIN
DEFINE v DECIMAL(10,2)
LET v = 11 / 3
DISPLAY "1. v = ", v USING "---&.&&&&&&&&"
LET v = v * 3
DISPLAY "2. v = ", v USING "---&.&&&&&&&&"
END MAIN

Output:
1. v = 3.67000000
2. v = 11.01000000

High-precision math functions

A couple of precision math functions are available, to be used with DECIMAL values. These functions have a higher precision as the standard C library functions based on C double data type, which is equivalent to FLOAT:

• FGL_DECIMAL_TRUNCATE()
• FGL_DECIMAL_SQRT()
• FGL_DECIMAL_EXP()
• FGL_DECIMAL_LOGN()
• FGL_DECIMAL_POWER()

**Related concepts**

**MONEY**(p,s) on page 303
The **MONEY** data type is provided to store currency amounts with exact decimal storage.

**FLOAT**

The **FLOAT** data type stores values as double-precision floating-point binary numbers with up to 16 significant digits.

**Syntax**

```markdown
FLOAT \[(\text{precision})\]
```

1. **FLOAT** and **DOUBLE PRECISION** are synonyms.
2. The **precision** can be specified, but it has no effect in programs.

**Usage**

The storage of **FLOAT** variables is based on 8 bytes of signed data ( =64 bits), this type is equivalent to the **double** data type in C.

**Note:** This data type is not recommended for exact decimal storage; use the **DECIMAL** type instead. **FLOAT** variables are initialized to zero in functions, modules and globals.

**FLOAT** values can be converted to strings based on the **DBFORMAT** (or **DBMONEY**) environment variable setting. Data type conversion can be controlled by catching the runtime exceptions. For more details, see **Handling type conversion errors** on page 317.

**Related concepts**

**DECIMAL**(p,s) on page 298
The **DECIMAL** data type is provided to handle large numeric values with exact decimal storage.

**INTEGER**

The **INTEGER** data type is used for storing large whole numbers.

**Syntax**

```markdown
INTEGER
```

1. **INT** and **INTEGER** are synonyms.

**Usage**

The storage of **INTEGER** variables is based on 4 bytes of signed data ( = 32 bits ).

The value range is from -2,147,483,647 to +2,147,483,647.

**INTEGER** variables can be initialized with **integer literals**:

```markdown
MAIN
    DEFINE i INTEGER
    LET i = 1234567
    DISPLAY i
END MAIN
```
When assigning a whole number that exceeds the INTEGER range, the overflow error -1215 will be raised.

INTEGER variables are initialized to zero in functions, modules and globals.

The INTEGER type can be used to define variables storing values from SERIAL columns.

Data type conversion can be controlled by catching the runtime exceptions. For more details, see Handling type conversion errors on page 317.

**Related concepts**

**BIGINT** on page 290
The BIGINT data type is used for storing very large whole numbers.

**SMALLINT** on page 304
The SMALLINT data type is used for storing small whole numbers.

**TINYINT** on page 306
The TINYINT data type is used for storing very small whole numbers.

**INTERVAL qual1 TO qual2**
The INTERVAL data type stores spans of time as Year/Month or Day/Hour/Minute/Second/Fraction units.

### Syntax 1: year-month class interval
```
INTERVAL YEAR[(precision)] TO MONTH
INTERVAL YEAR[(precision)] TO YEAR
INTERVAL MONTH[(precision)] TO MONTH
```

### Syntax 2: day-time class interval
```
INTERVAL DAY[(precision)] TO FRACTION[(scale)]
INTERVAL DAY[(precision)] TO SECOND
INTERVAL DAY[(precision)] TO MINUTE
INTERVAL DAY[(precision)] TO HOUR
INTERVAL DAY[(precision)] TO DAY
INTERVAL HOUR[(precision)] TO FRACTION[(scale)]
INTERVAL HOUR[(precision)] TO SECOND
INTERVAL HOUR[(precision)] TO MINUTE
INTERVAL HOUR[(precision)] TO HOUR
INTERVAL MINUTE[(precision)] TO FRACTION[(scale)]
INTERVAL MINUTE[(precision)] TO SECOND
INTERVAL MINUTE[(precision)] TO MINUTE
INTERVAL SECOND[(precision)] TO FRACTION[(scale)]
INTERVAL SECOND[(precision)] TO SECOND
INTERVAL FRACTION TO FRACTION[(scale)]
```

1. *precision* defines the number of significant digits of the first qualifier, it must be an integer from 1 to 9. For YEAR, the default is 4. For all other time units, the default is 2. For example, YEAR(5) indicates that the INTERVAL can store a number of years with up to 5 digits.

2. *scale* defines the scale of the fractional part, it can be 1, 2, 3, 4 or 5.

**Usage**
The INTERVAL data type stores a span of time, the difference between two points in time. It can also be used to store quantities that are measured in units of time, such as ages or times.
The INTERVAL data type falls into two classes. These are mutually exclusive because year and month are not fixed-length units of time, and therefore incompatible with INTERVAL data types whose time units are smaller than month:

- **Year-Time** intervals store a span of years, months or both.
- **Day-Time** intervals store a span of days, hours, minutes, seconds and fraction of seconds, or a contiguous subset of those units.

INTERVAL variables are initialized to NULL in functions, modules and globals.

INTERVAL variables can be assigned from interval literals, by using the `INTERVAL(q1 TO q2)` notation:

```
DEFINE iv INTERVAL DAY(5) TO SECOND
LET iv = INTERVAL(-7634 14:23:55) DAY(5) TO SECOND
```

INTERVAL variables can be assigned from string literals, by using the format `YYYY-MM` or `DD hh:mm:ss.fffff`, depending on the interval class:

```
DEFINE iv INTERVAL DAY(5) TO SECOND
LET iv = "-7634 14:23:55"
```

INTERVAL variables defined with a single time unit can be assigned from integer values, by using the `UNITS` operator:

```
DEFINE iv INTERVAL SECOND(5) TO SECOND
LET iv = 567 UNITS SECOND
```

The INTERVAL type is used for DATETIME computation.

Depending on the data type of the operands, DATETIME or DECIMAL, the arithmetic operations give different resulting types:

<table>
<thead>
<tr>
<th>Left Operand Type</th>
<th>Operator</th>
<th>Right Operand Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTERVAL</td>
<td>*</td>
<td>DECIMAL</td>
<td>INTERVAL</td>
</tr>
<tr>
<td>INTERVAL</td>
<td>/</td>
<td>DECIMAL</td>
<td>INTERVAL</td>
</tr>
<tr>
<td>INTERVAL</td>
<td>-</td>
<td>INTERVAL</td>
<td>INTERVAL</td>
</tr>
<tr>
<td>INTERVAL</td>
<td>+</td>
<td>INTERVAL</td>
<td>INTERVAL</td>
</tr>
<tr>
<td>DATETIME</td>
<td>-</td>
<td>INTERVAL</td>
<td>DATETIME</td>
</tr>
<tr>
<td>DATETIME</td>
<td>+</td>
<td>INTERVAL</td>
<td>DATETIME</td>
</tr>
<tr>
<td>DATETIME</td>
<td>-</td>
<td>DATETIME</td>
<td>INTERVAL</td>
</tr>
</tbody>
</table>

The next example shows how to use INTERVAL with DATETIME variables:

```
MAIN
DEFINE iym1, iym2 INTERVAL YEAR TO MONTH,
    dt1, dt2 DATETIME YEAR TO MINUTE,
    diff INTERVAL DAY(5) TO MINUTE
LET iym1 = "2342-4"
LET iym2 = "-55-11"
DISPLAY iym1 + iym2
LET dt1 = CURRENT
LET dt2 = "2010-12-24 00:00"
LET diff = dt1 - dt2
DISPLAY diff
LET diff = INTERVAL(-7634 14:23) DAY(5) TO MINUTE
```
For example, in the expression above `DISPLAY iym1 + iym2`, both values are from the same INTERVAL class, that is both are year-month, and the result of the DATETIME+INTERVAL calculation is a DATETIME value:

Result: DATETIME 2286-05 YEAR TO MONTH

INTERVAL values can be negative.

In order to check if an INTERVAL is negative, use the UNITS operator, to produce an interval constant for the comparison. Using numeric constants will not work:

```
MAIN
  DEFINE start, end DATETIME YEAR TO SECOND
  DEFINE diff INTERVAL SECOND(9) TO SECOND
  LET start = CURRENT + 100 UNITS SECOND
  LET end   = CURRENT - 200 UNITS SECOND
  LET diff = end - start
  IF diff < 0 THEN
    DISPLAY "this will not display!"
  END IF
  IF diff < 0 UNITS SECOND THEN
    DISPLAY "negative interval"
  ELSE
    DISPLAY "positive interval"
  END IF
END MAIN
```

Data type conversion can be controlled by catching the runtime exceptions. For more details, see Handling type conversion errors on page 317.

**Related concepts**

**DATE** on page 295
The DATE data type stores calendar dates with a Year/Month/Day representation.

**DATETIME qual1 TO qual2** on page 296
The DATETIME data type stores date and time data with time units from the year to fractions of a second.

**MONEY(p,s)**

The MONEY data type is provided to store currency amounts with exact decimal storage.

**Syntax**

```
MONEY [ (precision[, scale]) ]
```

1. `precision` defines the number of significant digits (limit is 32, default is 16).
2. `scale` defines the number of digits to the right of the decimal point.
3. When no `scale` is specified, it defaults to 2.
4. When no `(precision, scale)` is specified, it defaults to `MONEY(16, 2)`.

**Usage**

The MONEY data type is provided to store currency amounts. Its behavior is similar to the DECIMAL data type, with some important differences:

A MONEY variable is displayed with the currency symbol defined in the DBFORMAT (or DBMONEY) environment variable.
When the `scale` is not specified for the `MONEY` type, the default is 2. A `MONEY` without `precision` and `scale` defaults to `MONEY(16,2)`.

Data type conversion can be controlled by catching the runtime exceptions. For more details, see Handling type conversion errors on page 317.

See `DECIMAL(p,s)` on page 298 to learn other facts about the `MONEY(p,s)` data type.

**SMALLFLOAT**

The `SMALLFLOAT` data type stores values as single-precision floating-point binary numbers with up to 8 significant digits.

**Syntax**

<table>
<thead>
<tr>
<th>SMALLFLOAT</th>
</tr>
</thead>
</table>

1. `SMALLFLOAT` and `REAL` are synonyms.

**Usage**

The storage of `SMALLFLOAT` variables is based on 4 bytes of signed data (=32 bits), this type is equivalent to the `float` data type in C.

**Note:** This data type is not recommended for exact decimal storage; use the `DECIMAL` data type instead.

`SMALLFLOAT` variables are initialized to zero in functions, modules and globals.

`SMALLFLOAT` values can be converted to strings based on the `DBFORMAT` (or `DBMONEY`) environment variable setting.

Data type conversion can be controlled by catching the runtime exceptions. For more details, see Handling type conversion errors on page 317.

**Related concepts**

`DECIMAL(p,s)` on page 298

The `DECIMAL` data type is provided to handle large numeric values with exact decimal storage.

**SMALLINT**

The `SMALLINT` data type is used for storing small whole numbers.

**Syntax**

<table>
<thead>
<tr>
<th>SMALLINT</th>
</tr>
</thead>
</table>

**Usage**

The storage of `SMALLINT` variables is based on 2 bytes of signed data (=16 bits).

The value range is from -32,767 to +32,767.

`SMALLINT` variables can be initialized with integer literals:

```
MAIN
  DEFINE i SMALLINT
  LET i = 1234
  DISPLAY i
END MAIN
```

When assigning a whole number that exceeds the `SMALLINT` range, the overflow error `-1214` will be raised.
SMALLINT variables are initialized to zero in functions, modules and globals.

Data type conversion can be controlled by catching the runtime exceptions. For more details, see Handling type conversion errors on page 317.

Related concepts
- INTEGER on page 300
  The INTEGER data type is used for storing large whole numbers.
- BIGINT on page 290
  The BIGINT data type is used for storing very large whole numbers.
- TINYINT on page 306
  The TINYINT data type is used for storing very small whole numbers.

STRING

The STRING data type is a variable-length, dynamically allocated character string data type, without limitation.

Syntax

```plaintext
STRING
```

Usage

The STRING data type is typically used to implement utility functions manipulating character string with unknown size, and in some special cases, in SQL statements.

STRING variables are initialized to NULL in functions, modules and globals.

The behavior of a STRING variable is similar to the VARCHAR data type, except that there is no theoretical size limit.

STRING variables can be initialized from string literals:

```plaintext
MAIN
  DEFINE s STRING
  LET s = "abcdef"
END MAIN
```

Variables declared with the STRING data type can be used to call STRING-type methods such as `getLength()` or `toUpperCase()`. For more details, see STRING data type as class on page 2379:

```plaintext
MAIN
  DEFINE s STRING
  LET s = "abc"
  DISPLAY s.toUpperCase()
END MAIN
```

STRING variables have significant trailing blanks (i.e. "abc " is different from "abc"). However, in comparisons, trailing blanks do not matter:

```plaintext
MAIN
  DEFINE s STRING
  LET s = "abc  " -- a b c + 2 whitespaces
  DISPLAY "1: s.length:", s.getLength()
  DISPLAY ", s, [" -- displays ", s, ["abc  "]
  DISPLAY IIF(s=="abc","Equals",NULL)
END MAIN
```

Unlike CHAR and VARCHAR, a STRING can hold a value of zero length without being NULL. For example, if you trim a string variable with the `trim()` method and if the original value is a set of blank characters, the result is an
empty string. But testing the variable with the IS NULL operator will evaluate to FALSE. Using a VARCHAR with the CLIPPED operator would give a NULL string in this case:

```
MAIN
  DEFINE s STRING
  LET s = "     " -- 5 spaces
  LET s = s.trim()
  DISPLAY "s = [", s, "] len=", s.getLength()
  DISPLAY IIF(s IS NULL, "NULL", "not NULL")
END MAIN
```

outputs:

```
s = [] len=       0
not NULL
```

STRING typed variables can be used in some special cases to hold SQL character string data, when the size of the SQL data string is not known (string expressions, large strings like JSON documents). In order to store character string data stored in a database, consider using the CHAR or VARCHAR types instead of STRING.

In STRING methods, positions and length parameters (or return values) can be expressed in bytes or characters, depending on the length semantics used in programs. For more details, see Length semantics settings on page 521.

Character string manipulation with CHAR, VARCHAR and STRING types can have a cost when accessing parts of large strings, when using UTF-8 with char length semantics. Consider using base.StringBuffer objects when doing heavy string manipulations. For more details, see Manipulating character strings on page 588.

**Related concepts**
- STRING data type methods on page 2382
- CHAR(size) on page 293
- VARCHAR(size) on page 309

**TINYINT**

The TINYINT data type is used for storing very small whole numbers.

**Syntax**

```
TINYINT
```

**Usage**

The storage of TINYINT variables is based on 1 byte of signed data ( = 8 bits ).

The value range is from -128 to +127.

TINYINT variables can be initialized with integer literals:

```
MAIN
  DEFINE i TINYINT
  LET i = 101
  DISPLAY i
END MAIN
```

When assigning a whole number that exceeds the TINYINT range, the overflow error -8097 will be raised.

TINYINT variables are initialized to zero in functions, modules and globals.

The TINYINT variables cannot be NULL.
Data type conversion can be controlled by catching the runtime exceptions. For more details, see Handling type conversion errors on page 317.

**Related concepts**

**SMALLINT** on page 304
The SMALLINT data type is used for storing small whole numbers.

**INTEGER** on page 300
The INTEGER data type is used for storing large whole numbers.

**BIGINT** on page 290
The BIGINT data type is used for storing very large whole numbers.

**TEXT**
The TEXT data type stores large text data.

**Syntax**

```
TEXT
```

**Usage**

A BYTE or TEXT variable is a handle for a large object (LOB), that is stored in a file or in memory. Such data type is a complex type that cannot be used like INTEGER or CHAR basic types: It is designed to handle a large amount of data and has different semantics as simple types. The main difference with simple data types, is the fact that you must specify the storage with the LOCATE instruction, before using BYTE and TEXT variables.

The maximum size of data that can be handled by BYTE and TEXT variable is theoretically $2^{31}$ bytes (~2.14 Gigabytes), but the practical limit depends on the disk or memory resources available to the process.

BYTE and TEXT variable must be initialized with the LOCATE instruction before usage.

**Note:** APIs such as the util.JSON class will automatically allocate TEXT/BYTE variables in memory if these are not located, when converting a JSON string to an FGL variable with the util.JSON.parse() method.

The LOCATE instruction basically defines where the large data object has to be stored (in a named file, in a temporary file, or in memory). This instruction will actually allow you to fetch a LOB into memory or into a file, or insert a LOB from memory or from a file into the database. When located in a temporary file (IN FILE), the temp directory can be defined by the DBTEMP environment variable.

```
DEFINE t TEXT
LET t = "aaaa" -- invalid, t is not located
LOCATE t IN MEMORY
LET t = "aaaa" -- valid, now t is located in memory
```

**Important:**

TEXT/BYTE variables are not comparable: Comparing TEXT/BYTE variable does not result in an error, but the result of the comparison operator will always be NULL, which is equivalent to FALSE in an IF statement:

```
DEFINE t1, t2 TEXT
IF t1 == t2 THEN -- Always NULL/FALSE
...
END IF
```

With BYTE and TEXT types, you can insert/update/fetch large objects of the database. The native database type to be used depends on the type of database server. After defining the storage with LOCATE, load / assign its value, and then you can use it directly in the SQL statements, or fetch data from LOB columns of the database, like simple data types:

```
DEFINE t1, t2 TEXT
```
CREATE TABLE mytable ( id INT, data TEXT )

LOCATE t1 IN MEMORY
CALL t1.readFile("lob.4gl")
INSERT INTO mytable VALUES ( 1, t1 )
LOCATE t2 IN FILE
SELECT data INTO t2 FROM mytable WHERE id=1

BYTE and TEXT types implement the readFile() and writeFile() methods to read/write the whole large object data from/to files. These methods can be used to easily interface with other software components:

DEFINE t TEXT
LOCATE t IN MEMORY
CALL t.readFile("orig.txt")
CALL t.writeFile("copy.txt")

For more details about LOB types methods, see BYTE data type as class on page 2378 and TEXT data type as class on page 2391.

When initializing a BYTE or TEXT variable to NULL (INITIALIZE var TO NULL), if the variable is located in a file, the file is truncated (file size will be zero). If the variable is located in memory, the data in memory will be truncated. A subsequent usage of the variable (for example, FETCH INTO or LET assignment) is still possible:

DEFINE b BYTE
LOCATE b IN FILE "picture.png"
INITIALIZE b TO NULL
-- The file "picture.png" is now empty.

Resources allocated to a BYTE or TEXT variable can be deallocated with the FREE instruction.

Note: When the TEXT or BYTE variable is already located, a new LOCATE will free the allocated resource: If the prior LOCATE was using the IN FILE clause, the temporary file is dropped, if the prior LOCATE was using IN MEMORY, the memory is freed.

A FREE will remove the file if the LOB variable is located in a (named or temporary) file. When located in memory, the FREE instruction will deallocate the memory. After freeing the resources of a LOB variable, it must be re-located with a LOCATE instruction:

DEFINE b BYTE
LOCATE b IN FILE
CALL b.readFile("picture.png") -- ok
FREE b
CALL b.readFile("picture.png") -- Invalid, b is not located.
LOCATE b IN MEMORY
CALL b.readFile("picture.png") -- ok

Important:

TEXT and BYTE are reference types. This implies that assigning two variables (LET, passing a variable as parameter to a function, returning a result from a function) does not copy the value (Only the handle is copied. As a result, modifying the data with a TEXT/BYTE variable assigned from another TEXT/BYTE variable will in fact modify the same LOB data. Furthermore, the storage resource (file or memory) that was used by the assigned variable becomes unreferenced and is lost:

DEFINE b1, b2 BYTE -- Could be TEXT: same behavior
LOCATE b1 IN FILE "mydata" -- reference file directly
LOCATE b2 IN MEMORY -- use memory instead of file
CALL b2.readFile("mydata") -- read file content into memory
# FREE b2 -- this should be done to free memory before LET
LET b2 = b1 -- Now b2 points directly to the file (like b1)
INITIALIZE b1 TO NULL -- truncates reference file
DISPLAY IIF( b2 IS NULL, "b2 is null", "b2 is not null")
-- Displays "b2 is null"

In the next (invalid) code example, we try to save the value of the img BYTE variable in a temporary variable (tmp), with the typical programming pattern to save the value before modification. In fact the LET tmp=img assignment does not copy the data of the LOB like for simple data types (STRING, VARCHAR, DECIMAL), only the reference (i.e. handle) to the data is copied:

-- WARNING: THIS IS AN INVALID CODE EXAMPLE
DEFINE img, tmp BYTE
LOCATE img IN MEMORY
CALL img.readFile("picture1.png")
LOCATE tmp IN MEMORY
LET tmp = img -- Expecting to save the current data, but now
-- both variables reference the same data...
CALL img.readFile("picture2.png")
LET img = tmp -- Does not restore the old value: Same data.

If you need to clone a large object, use the writeFile() / readFile() methods.

It is possible to assign TEXT variables to/from VARCHAR, CHAR and STRING variables.

**Related concepts**

BYTE on page 290

The BYTE data type stores any type of binary data, such as images or sounds.

**VARCHAR(size)**

The VARCHAR data type is a variable-length character string data type, with a maximum size.

**Syntax**

```
VARCHAR [ size [ , reserve ] ]
```

1. **size** defines the maximum length of the character string, in byte or char units (depending on the character length semantics)
2. The maximum size of a VARCHAR type is 65534.
3. When no size is specified, it defaults to 1.
4. **reserve** is ignored; Its inclusion in the syntax is permitted for compatibility with the SQL data type.

**Usage**

The VARCHAR type is typically used to store variable-length character strings such as names, addresses and comments.

The size can be expressed in bytes or characters, depending on the length semantics used in programs. For more details about character length semantics, see Length semantics settings on page 521.

When size is not specified, the default length is 1.

VARCHAR variables are initialized to NULL in functions, modules and globals.

**Text literals** can be assigned to character string variables:

```
VARCHAR variables store trailing blanks (trailing blanks are displayed or printed in reports, and stored in database columns):

```plaintext
MAIN
    DEFINE vc VARCHAR(10)
    LET vc = "abc "          -- a b c + 2 whitespaces
    DISPLAY "[", vc ,"]"     -- displays [abc ]
END MAIN
```

Trailing blanks of a VARCHAR value are not significant in comparisons:

```plaintext
MAIN
    DEFINE vc VARCHAR(10)
    LET vc = "abc "          -- a b c + 2 whitespaces
    IF vc == "abc " THEN     -- evaluates to TRUE
        DISPLAY "equals"
    END IF
END MAIN
```

Numeric and date-time values can be directly assigned the character strings:

```plaintext
MAIN
    DEFINE vc VARCHAR(50), da DATE, dec DECIMAL(10,2)
    LET da = TODAY
    LET dec = 345.12
    LET vc = da, ": ", dec
END MAIN
```

When you insert character data from VARCHAR variables into VARCHAR columns in a database table, the trailing blanks are kept. Likewise, when you fetch VARCHAR column values into VARCHAR variables, trailing blanks are kept.

```plaintext
MAIN
    DEFINE vc VARCHAR(10)
    DATABASE test1
    CREATE TABLE table1 ( k INT, x VARCHAR(10) )
    LET vc = "abc "          -- two trailing blanks
    INSERT INTO table1 VALUES ( 1, vc )
    SELECT x INTO vc FROM table1 WHERE k = 1
    DISPLAY "[", vc ,"]"     -- displays [abc ]
END MAIN
```

In SQL statements, the behavior of the comparison operators when using VARCHAR values differs from one database to the other. IBM® Informix® is ignoring trailing blanks, but most other databases take trailing blanks of VARCHAR values into account. For more details, see SQL portability on page 607.

Character string manipulation with CHAR, VARCHAR and STRING types can have a cost when accessing parts of large strings, when using UTF-8 with char length semantics. Consider using `base.StringBuffer` objects when doing heavy string manipulations. For more details, see Manipulating character strings on page 588.

**Related concepts**

- `CHAR(size)` on page 293
- The CHAR data type is a fixed-length character string data type.
- `STRING` on page 305
The STRING data type is a variable-length, dynamically allocated character string data type, without limitation.

**Type conversions**

Explains primitive data type conversion rules of the language.

**When does type conversion occur?**

In Genero BDL, primitive data type conversion is implicit when possible.

The runtime system performs data conversion implicitly without objection, as long as the data conversion is valid. A date value can be converted to a character string, but a character string can only be converted to a date if the string represents a valid date in the current date format settings (DBDATE).

Implicit data type conversion can for example occur in the following cases:

- In a LET assignment,
- In an expression, when operands are not of the same data type,
- In DISPLAY instructions, or PRINT instructions in reports,
- In dialogs, when values must be converted to strings to be displayed in form fields,
- When passing and returning values to/from a function,
- When serializing numeric values in UNLOAD, JSON methods, etc.

In the next code example, implicit data type conversion occurs:

1. When assigning the result of the DECIMAL expression to the VARCHAR variable v,
2. When assigning a VARCHAR value to the DECIMAL variable d,
3. When passing the DECIMAL value d to function func(), expecting a VARCHAR,
4. When returning the VARCHAR value from the func() function,
5. When displaying the DECIMAL value (formatting rules apply).

```plaintext
MAIN
   DEFINE v VARCHAR(50),
       d DECIMAL(10,2)
   LET v = 1234.50 * 2   -- 1.
   LET d = v             -- 2.
   LET d = func(d)       -- 3. and 4.
   DISPLAY d             -- 5.
END MAIN

FUNCTION func(v)
   DEFINE v VARCHAR(50)
   DISPLAY v
   RETURN v   -- 4.
END FUNCTION
```

**Related concepts**

Runtime stack on page 493

The runtime stack is used to pass/return values to/from functions.

**Data type conversion reference**

This topic lists type conversion rules for all data types.

**Boolean type conversions**

A BOOLEAN value is an integer value 1 or 0 and thus can be converted to/from any other numeric type of the language.
When converting a numeric value to a BOOLEAN, any value different from 0 becomes TRUE, otherwise (zero) is FALSE.

```plaintext
DEFINE hasContent BOOLEAN, s STRING
LET s = "abc"
LET hasContent = s.getLength()
```

When converting a string (CHAR, VARCHAR or STRING) to BOOLEAN, the string will be converted to a number first, then the number-to-boolean conversion applies. If the string value cannot convert to a numeric value (for example, "abc"), the boolean value becomes NULL.

When converting a BOOLEAN to a string, the result will be "1" or "0" string values, depending on the boolean value.

**Large object type conversions**

A TEXT value can be converted to/from CHAR, VARCHAR or STRING.

The BYTE type cannot be converted to/from any other type.

**Integers to decimal types**

TINYINT, SMALLINT, INTEGER and BIGINT values can be converted to SMALLFLOAT, FLOAT, DECIMAL or MONEY as long as the decimal type is defined with sufficient digits to hold the whole number.

If the integer value exceeds the range of the receiving data type, an overflow error occurs.

**Decimal to integer types**

When converting a SMALLFLOAT, FLOAT DECIMAL or MONEY to a TINYINT, SMALLINT, INTEGER or BIGINT, the fractional part of the decimal value is truncated.

```plaintext
MAIN
    DEFINE d DECIMAL(10,2),
        i INTEGER
    LET d = 123.45
    LET i = d
    DISPLAY i   -- displays 123
END MAIN
```

If the decimal value exceeds the range of the receiving integer data type, an overflow error occurs.

**Decimal to decimal types**

Converting between SMALLFLOAT, FLOAT DECIMAL or MONEY types is allowed as long as the receiving type is defined with sufficient digits to hold the whole part of the original value.

If the original value contains more fractional digits than the receiving data type supports, low-order digits are discarded.

```plaintext
MAIN
    DEFINE d1 DECIMAL(10,2),
        d2 DECIMAL(5,1)
    LET d1 = 123.45
    LET d2 = d1
    DISPLAY d2    -- displays 123.5
END MAIN
```
Decimal to character types

Converting SMALLFLOAT, FLOAT, DECIMAL or MONEY values to CHAR, VARCHAR and STRING implies numeric formatting.

Numeric formatting is controlled by the DBMONEY and DBFORMAT environment variables.

The resulting string is left-aligned (for lossless conversions) or right-aligned (for visual conversions), depending on the conversion context; and the decimal part is kept depending on the numeric type.

```main
DEFINE m MONEY(8,2),
    s VARCHAR(10)
LET m = 123.45
LET s = m  -- Lossless conversion "$123.45"
DISPLAY m   -- Visual conversion   "$123.45"
END MAIN
```

Fixed point decimals (DECIMAL(p,s)) are converted to strings that can fit in a CHAR(p+2): The string is built with up to p significant digits + 1 character for the sign + 1 character for the decimal point. The result of a DECIMAL(p,s) to string conversion is never longer than p + 2 characters. For example, a DECIMAL(5,2) can produce "-999.99" (5 + 2 = 7c).

Floating point decimals (DECIMAL(p)) are converted to strings that can fit in a CHAR(p+7): The string is built with up to p significant digits + 1 character for the sign + 1 character for the decimal point + the length of the exponent of needed ("e+123"). The result of a DECIMAL(p) to string conversion is never longer than p + 7. For example, a DECIMAL(5) can produce ".12345e-123" (5 + 7 = 12c).

DECIMAL to string conversion depends on the context in which the conversion occurs:

1. **Visual conversion**: The result of this conversion will typically be presented to the end user. This conversion happens in DISPLAY, MESSAGE, ERROR, PRINT. The result of a visual conversion is right aligned (padded with leading blanks). This padding results in the same length for any value for a given decimal precision. The length of the result is the maximum possible length as described previously (p+2 for DECIMAL(p,s), p+7 for DECIMAL(p)).

   Visual conversion examples for DECIMAL(5,2):
   ```
<table>
<thead>
<tr>
<th>Values</th>
<th>1234567</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>-999.99</td>
<td>-999.99</td>
</tr>
<tr>
<td>12.3</td>
<td>12.30</td>
</tr>
<tr>
<td>12.34</td>
<td>12.34</td>
</tr>
</tbody>
</table>
   ```

   Visual conversion examples for DECIMAL(5):
   ```
<table>
<thead>
<tr>
<th>Values</th>
<th>123456789012</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>-99999</td>
<td>-99999.0</td>
</tr>
<tr>
<td>12.3</td>
<td>12.3</td>
</tr>
<tr>
<td>12.34</td>
<td>12.34</td>
</tr>
<tr>
<td>12.345</td>
<td>12.345</td>
</tr>
<tr>
<td>1.23e7</td>
<td>12300000.0</td>
</tr>
<tr>
<td>1e100</td>
<td>1e100</td>
</tr>
</tbody>
</table>
   ```

2. **Form field conversion**: This conversion concerns decimal numbers presented in form-fields. The result of this conversion is in best case the same as (1). The result of the conversion depends on the width of the form-field. If the width of the form-field is smaller than the perfect length, automatic rounding and exponential notation might be used.
3. **Lossless conversion**: Such conversion happens when assigning numbers to string variables (LET), passing numbers as parameters to functions expecting strings, returning numbers from functions to strings, serializing numbers (UNLOAD, XML or JSON APIs). These conversions must avoid the loss of significant digits. When using floating point decimals, this leads to a variable length of the resulting string. A conversion must be reversible: decimal to string to decimal must give the original value. If the target variable is shorter then the maximum possible length, then automatic rounding will occur.

Lossless conversion examples of DECIMAL(5,2):

<table>
<thead>
<tr>
<th>Values</th>
<th>1234567</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&quot;0.00&quot;</td>
</tr>
<tr>
<td>-999.99</td>
<td>&quot;-999.99&quot;</td>
</tr>
<tr>
<td>12.3</td>
<td>&quot;12.30&quot;</td>
</tr>
<tr>
<td>12.34</td>
<td>&quot;12.34&quot;</td>
</tr>
</tbody>
</table>

Lossless conversion examples of DECIMAL(5):

<table>
<thead>
<tr>
<th>Values</th>
<th>123456789012</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&quot;0.0&quot;</td>
</tr>
<tr>
<td>-99999</td>
<td>&quot;-99999.0&quot;</td>
</tr>
<tr>
<td>12.3</td>
<td>&quot;12.3&quot;</td>
</tr>
<tr>
<td>12.34</td>
<td>&quot;12.34&quot;</td>
</tr>
<tr>
<td>12.345</td>
<td>&quot;12.345&quot;</td>
</tr>
<tr>
<td>1.23e7</td>
<td>&quot;12300000.0&quot;</td>
</tr>
<tr>
<td>1e100</td>
<td>&quot;1e100&quot;</td>
</tr>
</tbody>
</table>

Automatic rounding occurs if the target string variable is shorter than the maximum possible length of the DECIMAL type. Such conversion might lose significant digits: The runtime system tries to round the value, to fit into the target variable.

<table>
<thead>
<tr>
<th>Values</th>
<th>Different target sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.98765</td>
<td>&quot;0.9877&quot; &quot;0.988&quot; &quot;0.99&quot;</td>
</tr>
<tr>
<td>123.45</td>
<td>&quot;123.45&quot; &quot;123.5&quot; &quot;123&quot;</td>
</tr>
</tbody>
</table>

Automatic switch to the exponential notation will occur if the integer part of the decimal value does not fit into the target string variable. For example, if the source variable is a DECIMAL(12) and the target variable is a CHAR(9):

<table>
<thead>
<tr>
<th>Values</th>
<th>123456789</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234567</td>
<td>&quot;1234567.0&quot;</td>
</tr>
<tr>
<td>12345678</td>
<td>&quot;12345678&quot;</td>
</tr>
<tr>
<td>123456789</td>
<td>&quot;123456789&quot;</td>
</tr>
<tr>
<td>1234567890</td>
<td>&quot;1.2346e10&quot;</td>
</tr>
<tr>
<td>12345678901</td>
<td>&quot;1.2346e11&quot;</td>
</tr>
</tbody>
</table>

The exponential notation will also be used if the absolute value of a floating point decimal is less than 1e-8 (0.00000001).

Default formatting of floating point DECIMAL(P) has been revised with Genero 2.50. If DECIMAL(P)-to-string conversion must round to 2 digits, use the fglrun.decToCharScale2 FGLPROFILE entry (this applies to all contexts):

```fglrun.deToCharScale2 = true```
Another FGLPROFILE entry can be used, to get the 2-digit rounding of DECIMAL(P) only in the context of the PRINT statement in reports. (the \texttt{fglrun.decToCharScale2} and \texttt{fglrun.decToCharScale2.print} parameters are mutually exclusive):

\begin{verbatim}
fglrun.decToCharScale2.print = true
\end{verbatim}

\textbf{Note:} Do not use the \texttt{fglrun.decToCharScale2*} configuration parameters, unless you have migration issues.

Formatting a \texttt{FLOAT} is the same as \texttt{DECIMAL(16)}. Any \texttt{FLOAT} value with up to 15 digits is exact. There is no precision loss when converting an exact \texttt{FLOAT} back and forth to/form a string. Some \texttt{FLOAT} values require 16, in some rare cases 17 digits for an exact string representation. 16 and 17 digits are not always exact: "8.000000000000001" and "8.000000000000002" represent the same float value.

Formatting a \texttt{SMALLFLOAT} is the same as \texttt{DECIMAL(7)}. Any \texttt{SMALLFLOAT} value with up to 6 digits is exact. There is no precision loss when converting an exact \texttt{SMALLFLOAT} back and forth to/form a string. Some \texttt{SMALLFLOAT} values require 7, in some rare cases 8 digits for an exact string representation. 7 and 8 digits \texttt{SMALLFLOAT} are not always exact: "0.0009999901" and "0.0009999902" represent the same \texttt{SMALLFLOAT} value.

If \texttt{FLOAT}/\texttt{SMALLFLOAT}-to-string conversion must round to 2 digits, use the \texttt{fglrun.floatToCharScale2} FGLPROFILE entry (this applies to all contexts):

\begin{verbatim}
fglrun.floatToCharScale2 = true
\end{verbatim}

Another FGLPROFILE entry can be used, to get the 2-digit rounding of \texttt{FLOAT}/\texttt{SMALLFLOAT} only in the context of the PRINT statement in reports. (the \texttt{fglrun.floatToCharScale2} and \texttt{fglrun.floatToCharScale2.print} parameters are mutually exclusive):

\begin{verbatim}
fglrun.floatToCharScale2.print = true
\end{verbatim}

\textbf{Note:} Do not use the \texttt{fglrun.floatToCharScale2*} configuration parameters, unless you have migration issues.

\section*{Character to decimal types}

A \texttt{CHAR}, \texttt{VARCHAR} and \texttt{STRING} value can be converted to a \texttt{TINYINT}, \texttt{SMALLINT}, \texttt{INTEGER}, \texttt{BIGINT}, \texttt{SMALLFLOAT}, \texttt{FLOAT} \texttt{DECIMAL} or \texttt{MONEY} value as long as the character string value represents a valid number.

If the original value contains more significant digits or more fractional digits than the receiving data type supports, low-order digits are discarded.

\begin{verbatim}
MAIN
    DEFINE d DECIMAL(10,2)
    LET d = "-123.45"
    DISPLAY d  -- displays -123.45
    LET d = "1234567890123.45"
    DISPLAY d  -- displays null
    LET d = "12345678.999"
    DISPLAY d  -- displays 12345679.00
END MAIN
\end{verbatim}

\section*{Date time to character types}

Converting \texttt{DATE}, \texttt{DATETIME} and \texttt{INTERVAL} values to \texttt{CHAR}, \texttt{VARCHAR} and \texttt{STRING} implies date time formatting.

\texttt{DATE} formatting is controlled by the \texttt{DBDATE} environment variable.

When converting a \texttt{DATETIME} to a string, the \texttt{YYYY-MM-DD hh:mm:ss.fffff} standard format is used.
When converting an INTERVAL to a string, either YYYY-MM-DD  hh:mm:ss.ffffff standard formats are used, depending on the interval class.

If the conversion result is longer than the receiving variable, the resulting character string is null.

```
MAIN
  DEFINE d DATE,
     s VARCHAR(20),
     v VARCHAR(5)
  LET d = MDY(12,24,2012)
  LET s = d
  DISPLAY s -- displays 12/24/2012
  LET v = d
  DISPLAY v -- displays null
END MAIN
```

### Character to date time types

Converting a CHAR, VARCHAR or STRING value to a DATE, DATETIME or INTERVAL is possible as long as the character string defines a well formatted date time or interval value.

When converting a character string to a DATE, the string must follow the date format defined by the DBDATE environment variable.

When converting a string to a DATETIME, the format must be YYYY-MM-DD hh:mm:ss.ffffff or follow the ISO 8601 format sub-set (with the T separator between the date and time part, and with optional UTC indicator or timezone offset)

```
MAIN
  DEFINE dt DATETIME YEAR TO SECOND
  LET dt = "2012-12-24 11:33:45"
  DISPLAY dt -- displays 2012-12-24 11:33:45
  LET dt = "2012-12-24T11:33:45+01:00"
  DISPLAY dt -- displays 2012-12-24 11:33:45 (if TZ=UTC+1h)
  LET dt = "2012-12-24T10:33:45Z"
  DISPLAY dt -- displays 2012-12-24 11:33:45 (if TZ=UTC+1h)
END MAIN
```

### Converting DATE to/from DATETIME types

When converting a DATETIME to another DATETIME with a different precision, truncation from the left or right can occur. If the target type has more fields than the source type, the year, month and day fields are filled with the current date.

When converting a DATE to a DATETIME, the datetime fields are filled with year, month and day from the date value and time fields are set to zero.

When converting a DATETIME to a DATE, an implicit EXTEND( datetime-value, YEAR TO DAY ) is performed.

```
MAIN
  DEFINE da DATE,
     dt1 DATETIME YEAR TO SECOND,
     dt2 DATETIME HOUR TO MINUTE
  LET da = MDY(12,24,2012)
  LET dt1 = da
  DISPLAY dt1 -- displays 2012-12-24 00:00:00
  LET dt2 = "23:45"
  LET dt1 = dt2
  DISPLAY dt1 -- displays <current date> 00:00:00
END MAIN
```
**Unsupported type conversions**

Other data type conversions not mentioned in this topic are not allowed and will result in a runtime error.

**Related concepts**

Primitive Data types on page 289

Selecting the correct data type assists you in the input, storage, and display of your data.

**Handling type conversion errors**

Runtime errors can be handled on type conversion failures.

By default, in cases of type conversion or overflow errors, the program continues, the target variable is set to NULL and the global STATUS variable is not set.

In order to detect data conversion and overflow errors, use the **WHENEVER ANY ERROR** statement.

This code example shows use of the **WHENEVER ANY ERROR** statement:

```plaintext
MAIN  -- DBDATE set to Y4MD-
     DEFINE v VARCHAR(50), d DATE
     LET v = "2012-99-99"      -- invalid date string
     LET d = v
     DISPLAY status, "/", NVL(d,"NULL") -- displays 0/NULL
     WHENEVER ANY ERROR CONTINUE
     LET d = v
     DISPLAY status, "/", NVL(d,"NULL") -- displays -1205/NULL
     WHENEVER ANY ERROR STOP
     LET d = "2012-11-23" -- valid date, ok
     DISPLAY status, "/", NVL(d,"NULL") -- displays 0/2012-11-23
     LET d = v -- program execution stopped with error -1205
END MAIN
```

The code above will produce the following output:

```
0/NULL
-1218/NULL
Program stopped at 'x.4gl', line number 10.
FORMS statement error number -1218.
String to date conversion error.
```

Conversion and overflow errors are implicitly trapped in TRY/CATCH blocks.

In this example, the INTERVAL variable is not large enough to hold the result of \( d_2 - d_1 \):

```plaintext
MAIN
     DEFINE d1, d2 DATETIME YEAR TO FRACTION(5)
     DEFINE i INTERVAL SECOND(2) TO SECOND
     LET d1 = "2015-11-06 17:40:21.436"
     LET d2 = "2015-11-06 10:40:21.436"
     TRY
       LET i = d2 - d1
     CATCH
       DISPLAY STATUS, " / ", err_get(STATUS)
     END TRY
END MAIN
```

Above code will produce the following output:

```
-1265 / Overflow occurred on a datetime or interval operation.
```

**Related concepts**

Program registers on page 567
Predefined global registers can be used in programs to detect errors, signals and events.

**Related reference**
- Genero BDL errors on page 3154
- System error messages sorted by error number.

## Formatting data

Explains data to string conversion options of the language.

### Formatting numeric values

Numeric values must be formatted when converted to strings.

#### When does numeric formatting take place?

Numeric formatting occurs when converting a number to a string with the `USING` operator, for example in a `LET`, `DISPLAY` or `PRINT` instruction, and when displaying numeric values in form fields defined with the `FORMAT` attribute.

Numeric values can be of type such as `INTEGER`, `FLOAT`, `DECIMAL`, `MONEY`, etc.

This example formats a `DECIMAL(10,2)` value with the `USING` operator:

```java
MAIN
    DEFINE d DECIMAL(10,2)
    LET d = -123456.78
    DISPLAY d USING "-,---,--&.&& @"
END MAIN
```

Front currency symbol, thousands separator, decimal separator and back currency symbol are defined with the `DBFORMAT` (or `DBMONEY`) environment variable. For example, with `DBFORMAT` defined as `".:,:,:E"`, there is no front currency symbol defined, the thousands separator is a dot, the decimal separator is a comma and the back currency symbol is `E`. With this setting, the previous code example will produce the following output:

```
-123,456.78 E
```

Default formatting occurs when `USING` or `FORMAT` are not used, and a numeric value has to be converted to a character string, for example when passing a `DECIMAL(p,s)` to a function expecting a `VARCHAR(n)`. For more details about default formatting, see Data type conversion reference on page 311.

This topic describes the syntax of the `format-string` in the `USING "format-string"` operator and `FORMAT = "format-string"` form field attribute.

### Formatting symbols for numbers

When formatting numeric values, the `format-string` of the `USING` operator or `FORMAT` attribute consists of a set of place holders that represent digits, currency symbols, thousands and decimal separators. For example, `"###.##@"` defines three places to the left of the decimal point and exactly two to the right, plus a "back" currency symbol at the end of the string.

**Note:** The `USING` operator or `FORMAT` attribute are required to display the thousands separator defined in `DBFORMAT`.

The `format-string` must use normalized placeholders described in Table 122: Format-string symbols for Numeric data types on page 319. The placeholders will be replaced by digits, blanks or by the elements defined in the `DBFORMAT` (or `DBMONEY`) environment variables. Any other character will be interpreted as a literal, and can be used at any place in the format string.
If the numeric value is too large to fit in the number of characters defined by the format, the result string is filled with a set of star characters (********), indicating an overflow. Note that this includes the +/- sign.

**Note:** If the numeric values can be negative numbers, specify one additional – (minus) placeholder for the sign. Otherwise, if the value is negative and not enough placeholders are defined, overflow star characters will be displayed.

The minus sign ( – ), plus sign ( + ), parentheses ( ( ) ), and dollar sign ( $ ) float. This means that when you specify multiple leading occurrences of one of these characters, the result string gets only a single character immediately to the left of the first digit.

**Table 122: Format-string symbols for Numeric data types**

<table>
<thead>
<tr>
<th>Placeholder</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>On the left of the decimal separator, the star * placeholder fills with asterisks any position that would otherwise be blank. When * is used on the right of the decimal separator, a zero digit is displayed instead of blank.</td>
</tr>
<tr>
<td>&amp;</td>
<td>The ampersand &amp; placeholder is used to define the position of a digit, and is replaced by a zero if that position would otherwise be blank.</td>
</tr>
<tr>
<td>#</td>
<td>On the left of the decimal separator, the hash # placeholder is replaced by a blank, if no digit is to be displayed at that position. On the right of the decimal separator, a zero digit is displayed instead of blank.</td>
</tr>
<tr>
<td>&lt;</td>
<td>The &quot;less than&quot; &lt; placeholder aligns the numeric value to the left: Consecutive &quot;less than&quot; characters cause left alignment and define digit positions.</td>
</tr>
<tr>
<td>–</td>
<td>The minus – placeholder displays a minus sign if the value is negative, or a blank if the value is positive. When grouping several minus signs in the format string, a single minus sign floats immediately to the left of the first digit. When – is used on the right of the decimal separator, a zero digit is displayed instead of blank or minus sign.</td>
</tr>
<tr>
<td>+</td>
<td>The plus + placeholder displays a minus sign if the value is negative, or a plus sign if the value is positive. When grouping several plus signs in the format string, a single plus sign floats immediately to the left of the first digit. When + is used on the right of the decimal separator, a zero digit is displayed instead of blank or plus/minus sign.</td>
</tr>
<tr>
<td>(</td>
<td>The opening parenthesis ( is displayed as left parenthesis for negative numbers. It is used to display &quot;accounting parentheses&quot; instead of a minus sign for negative numbers. Consecutive left parentheses display a single left parenthesis to the left of the first digit.</td>
</tr>
<tr>
<td>)</td>
<td>The closing parenthesis ) is displayed as right parenthesis for negative numbers. This placeholder is used in conjunction with a opening parenthesis to display &quot;accounting parentheses&quot; for negative numbers.</td>
</tr>
<tr>
<td>, (comma)</td>
<td>The comma placeholder is used to define the position for the thousand separator defined in DBFORMAT. The thousand separator will only be displayed if there is a number on the left of it.</td>
</tr>
<tr>
<td>. (period)</td>
<td>The period placeholder is used to define the position for the decimal separator defined in DBFORMAT. You can only have one decimal separator in a number format string.</td>
</tr>
<tr>
<td>$</td>
<td>The dollar sign is the placeholder for the front currency symbol defined in DBFORMAT. When you group several consecutive dollar signs, a single front currency symbol floats immediately to the left of the first digit. The front currency symbol can be defined in DBFORMAT with more than one character (EUR, USD).</td>
</tr>
<tr>
<td>@</td>
<td>The &quot;at&quot; sign is the placeholder for the back currency symbol defined in DBFORMAT. Put several consecutive @ signs at the end of the format string to display a currency symbol defined in DBFORMAT with more than one character.</td>
</tr>
</tbody>
</table>
### Table 123: Numeric formatting examples

<table>
<thead>
<tr>
<th>Format String</th>
<th>Value</th>
<th>DBFORMAT</th>
<th>Result string</th>
</tr>
</thead>
<tbody>
<tr>
<td>[#####.##]</td>
<td>0</td>
<td>:.;:</td>
<td>[__<strong>,</strong>_]</td>
</tr>
<tr>
<td>[#####.##]</td>
<td>-1234.56</td>
<td>:.;:</td>
<td>[___1234,56] (no sign!)</td>
</tr>
<tr>
<td>[#####.##]</td>
<td>-1234567.89</td>
<td>:.;:</td>
<td>[*********] (overflow)</td>
</tr>
<tr>
<td>[#####.##]</td>
<td>+1234.56</td>
<td>:.;:</td>
<td>[___1234,56]</td>
</tr>
<tr>
<td>[#####&amp;.&amp;]</td>
<td>0</td>
<td>:.;:</td>
<td>[____0,00]</td>
</tr>
<tr>
<td>[***<strong>.</strong>]</td>
<td>0</td>
<td>:.;:</td>
<td>[*****0,00]</td>
</tr>
<tr>
<td>[***<strong>.</strong>]</td>
<td>-12.34</td>
<td>:.;:</td>
<td>[***12,34] (no sign!)</td>
</tr>
<tr>
<td>[***<strong>.</strong>]</td>
<td>+12.34</td>
<td>:.;:</td>
<td>[***12,34]</td>
</tr>
<tr>
<td>[&lt;&lt;&lt;&lt;&lt;&lt;.&lt;&lt;]</td>
<td>-12.34</td>
<td>:.;:</td>
<td>[12,34] (no sign!)</td>
</tr>
<tr>
<td>[&lt;&lt;&lt;&lt;&lt;&lt;.&lt;&lt;]</td>
<td>+12.34</td>
<td>:.;:</td>
<td>[12,34]</td>
</tr>
<tr>
<td>[---,--&amp;.&amp;]</td>
<td>-1234.56</td>
<td>:.;:</td>
<td>[___-1.234,56]</td>
</tr>
<tr>
<td>[+++;++&amp;.&amp;]</td>
<td>-1234.56</td>
<td>:.;:</td>
<td>[___-1.234,56]</td>
</tr>
<tr>
<td>[+++;++&amp;.&amp;]</td>
<td>+1234.56</td>
<td>:.;:</td>
<td>[+1234,56]</td>
</tr>
<tr>
<td>[$---,--&amp;.&amp;]</td>
<td>-1234.56</td>
<td>E:.;:</td>
<td>[E-1.234,56]</td>
</tr>
<tr>
<td>[$---,--&amp;.&amp;]</td>
<td>+1234.56</td>
<td>E:.;:</td>
<td>[E_1.234,56]</td>
</tr>
<tr>
<td>[$$$---,--&amp;.&amp;]</td>
<td>+1234.56</td>
<td>E:.;:</td>
<td>[E_E1.234,56]</td>
</tr>
<tr>
<td>[$$$---,--&amp;.&amp;]</td>
<td>+1234.56</td>
<td>EUR:.;:</td>
<td>[EUR1.234,56]</td>
</tr>
<tr>
<td>[-,-,-$&amp;.&amp;]</td>
<td>-12.34</td>
<td>E:.;:</td>
<td>[____-E12,34]</td>
</tr>
<tr>
<td>[-,-,-$&amp;.&amp;]</td>
<td>-1234.56</td>
<td>E:.;:</td>
<td>[___-E1.234,56]</td>
</tr>
<tr>
<td>[-,-,$$$,$&amp;.&amp;]</td>
<td>-12.34</td>
<td>E:.;:</td>
<td>[___-E12,34]</td>
</tr>
<tr>
<td>[-,-,$$$,$&amp;.&amp;]</td>
<td>-1234.56</td>
<td>E:.;:</td>
<td>[___-E1.234,56]</td>
</tr>
<tr>
<td>[-,-,-&amp;.&amp;@]</td>
<td>-1234.56</td>
<td>:,E</td>
<td>[-1.234,56E]</td>
</tr>
<tr>
<td>[-,-,-&amp;.&amp;@]</td>
<td>+1234.56</td>
<td>:,E</td>
<td>[+1.234,56E]</td>
</tr>
<tr>
<td>[-,-,-&amp;.&amp;@@@]</td>
<td>+1234.56</td>
<td>:,EUR</td>
<td>[+1.234,56EUR]</td>
</tr>
<tr>
<td>[($)---,$--&amp;.&amp;]</td>
<td>-1234.56</td>
<td>E:.;:</td>
<td>[(E-1.234,56)]</td>
</tr>
<tr>
<td>[($)---,$--&amp;.&amp;]</td>
<td>-1234.56</td>
<td>E:.;:</td>
<td>[(E__1.234,56)] (no sign!)</td>
</tr>
<tr>
<td>[($$$$,$##&amp;.&amp;)]</td>
<td>0</td>
<td>E:.;:</td>
<td>[____<em>E,00</em>]</td>
</tr>
<tr>
<td>[($$,$$&amp;.&amp;)]</td>
<td>-12.34</td>
<td>E:.;:</td>
<td>[____(E12,34)] (no sign!)</td>
</tr>
<tr>
<td>[($$,$$&amp;.&amp;)]</td>
<td>+12.34</td>
<td>E:.;:</td>
<td>[____<em>E12,34</em>]</td>
</tr>
<tr>
<td>[($$,$$&amp;.&amp;)]</td>
<td>-1234.56</td>
<td>E:.;:</td>
<td>[_(E1.234,56)] (no sign!)</td>
</tr>
<tr>
<td>[($$,$$&amp;.&amp;)]</td>
<td>+1234.56</td>
<td>E:.;:</td>
<td>[_<em>E1.234,56</em>]</td>
</tr>
</tbody>
</table>

**Note:** In the first and last columns, the strings are shown with opening and closing square brackets. These brackets mark the start and the end of the string. In the result column, underscore characters are used to represent spaces. This is important when examining the result string, as it shows where spaces exist.

**Related concepts**
- Formatting DATE values on page 321
Date values must be formatted when converted to strings.

**Formatting DATETIME values** on page 322

Date-time values must be formatted when converted to strings.

**Primitive Data types** on page 289

Selecting the correct data type assists you in the input, storage, and display of your data.

### Formatting DATE values

Date values must be formatted when converted to strings.

#### When does DATE formatting take place?

Date formatting occurs when converting a `DATE` to a string with the `USING` operator, for example in a `LET`, `DISPLAY` or `PRINT` instruction, and when displaying date values in form fields defined with the `FORMAT` attribute.

This example formats a `DATE` value with the `USING` operator:

```plaintext
MAIN
  DEFINE d DATE
  LET d = MDY(12,24,2014)
  DISPLAY d USING "mmm ddd yyyy"
END MAIN
```

This code example produces the following output:

```
Dec Wed 2014
```

Default formatting occurs when `USING` or `FORMAT` are not used, and a date value has to be converted to a character string, for example when passing a `DATE` to a function expecting a `VARCHAR(n)`.

Default date formatting is based on the date format defined with the `DBDATE` environment variable. For more details about default formatting, see Data type conversion reference on page 311.

This topic describes the syntax of the `format-string` in the `USING "format-string"` operator and `FORMAT = "format-string"` form field attribute.

### Formatting symbols for DATE values

When formatting `DATE` values, the `format-string` of the `USING` operator or `FORMAT` attribute consists of a set of place holders that represent date parts as described in the following table:

**Table 124: Format-string symbols for DATE values**

<table>
<thead>
<tr>
<th>Placeholder</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dd</td>
<td>Day of the month as a 2-digit integer.</td>
</tr>
<tr>
<td>ddd</td>
<td>Three-letter English-language abbreviation of the day of the week. For example: Mon, Tue.</td>
</tr>
<tr>
<td>mm</td>
<td>Month as a 2-digit integer.</td>
</tr>
<tr>
<td>mmm</td>
<td>Three-letter English-language abbreviation of the month. For example: Jan, Feb.</td>
</tr>
<tr>
<td>yy</td>
<td>Year, as a 2-digits integer representing the 2 trailing digits.</td>
</tr>
<tr>
<td>yyyy</td>
<td>Year as a 3-digit number (Ming Guo format only)</td>
</tr>
<tr>
<td>yyyy</td>
<td>Year as a 4-digit number.</td>
</tr>
<tr>
<td>c1</td>
<td>Ming Guo format modifier, see Using the Ming Guo date format on page 532.</td>
</tr>
</tbody>
</table>
Any character different from the date-formatting placeholders is interpreted as a literal and will appear as-is in the resulting string.

The calendar used for date formatting is the Gregorian calendar. The c1 placeholder is a formatting symbol used to adapt the date to the Ming Guo calendar.

Table 125: Date formatting examples

<table>
<thead>
<tr>
<th>Format String</th>
<th>Date value</th>
<th>Result string</th>
</tr>
</thead>
<tbody>
<tr>
<td>dd/mm/yyyy</td>
<td>2011-10-24</td>
<td>24/10/2011</td>
</tr>
<tr>
<td>[dd/mm/yy]</td>
<td>2011-10-24</td>
<td>[24/10/11]</td>
</tr>
<tr>
<td>(ddd.) mmm. dd, yyyy</td>
<td>1999-09-23</td>
<td>(Thu.) Sep. 23, 1999</td>
</tr>
<tr>
<td>mmm dd'yy</td>
<td>2020-12-05</td>
<td>Dec 05’20</td>
</tr>
</tbody>
</table>

Related concepts
Using the Ming Guo date format on page 532
Genero BDL can be configured to use the Ming Guo calendar.

Formatting DATETIME values
Date-time values must be formatted when converted to strings.

When does DATETIME formatting take place?
Datetime formatting occurs when converting a DATETIME to a string, for example in a LET, DISPLAY or PRINT instruction, and when displaying datetime values in form fields.

By default, DATETIME values are formatted in the ISO format:

```
yyyy-mm-dd hh:mm:ss.fffff
```

A DATETIME value can be formatted with the `util.Datetime.format()` method:

```
IMPORT util
MAIN
  DEFINE dt DATETIME YEAR TO SECOND
  LET dt = CURRENT
  DISPLAY util.Datetime.format(dt, "%Y-%m-%d %H:%M:%S")
END MAIN
```

This code example produces the following output:

```
2015-12-23 11:45:33
```

Converting strings to DATETIME values
When a string represents a datetime value in ISO format, it can be directly converted to a DATETIME:

```
DEFINE dt DATETIME YEAR TO FRACTION(5)
LET dt = "2015-12-24 11:34:56.82373"
```

If you need to convert a string that does not follow the ISO format, use the `util.Datetime.parse()` method, by specifying a format string:

```
DEFINE dt DATETIME YEAR TO MINUTE
```
LET dt = util.Datetime.parse( "2014-12-24 23:45", "%Y-%m-%d %H:%M" )

### Formatting symbols for DATETIME values

When formatting DATETIME values, the format-string of the `util.Datetime.parse()` and `util.Datetime.format()` methods consists of a set of place holders that represent the different parts of a datetime value (year, month, day, hour, minute, second and fraction).

This shows the formatting symbols for DATETIME expressions. Any character different from the placeholders described in this table is interpreted as a literal and will appear as-is in the resulting string.

The calendar used for date formatting is the Gregorian calendar.

#### Table 126: Format-string symbols for DATETIME values

<table>
<thead>
<tr>
<th>Placeholder</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%a</td>
<td>The abbreviated name of the day of the week. <strong>Note:</strong> When parsing a datetime string, %a and %A are equivalent to detect the name of the day of the week in abbreviated form or full day name.</td>
</tr>
<tr>
<td>%A</td>
<td>The full name of the day of the week. The abbreviated month name.</td>
</tr>
<tr>
<td>%b or %h</td>
<td><strong>Note:</strong> When parsing a datetime string, %b/%h and %B are equivalent to detect the month name in abbreviated form or full month name.</td>
</tr>
<tr>
<td>%B</td>
<td>The full month name.</td>
</tr>
<tr>
<td>%c</td>
<td>The date and time representation.</td>
</tr>
<tr>
<td>%C</td>
<td>The century number (0-99)</td>
</tr>
<tr>
<td>%D</td>
<td>Equivalent to %m/%d/%y</td>
</tr>
<tr>
<td>%d</td>
<td>The day of month with 2 digits (01-31)</td>
</tr>
<tr>
<td>%e</td>
<td>The day of month with one or 2 digits (1-31)</td>
</tr>
<tr>
<td>%F</td>
<td>The fractional part of a second</td>
</tr>
<tr>
<td>%H</td>
<td>The hour with 2 digits (00-23).</td>
</tr>
<tr>
<td>%I</td>
<td>The hour on a 12-hour clock (1-12)</td>
</tr>
<tr>
<td>%y</td>
<td>The year on 2 digits (91)</td>
</tr>
<tr>
<td>%Y</td>
<td>The year on 4 digits (1991)</td>
</tr>
<tr>
<td>%m</td>
<td>The month as 2 digits (01-12)</td>
</tr>
<tr>
<td>%M</td>
<td>The minutes (00-59)</td>
</tr>
<tr>
<td>%n</td>
<td>A newline character</td>
</tr>
<tr>
<td>%p</td>
<td>The locale’s equivalent of AM or PM</td>
</tr>
<tr>
<td>%r</td>
<td>The 12-hour clock time. In the POSIX locale equivalent to %I:%M:%S %p</td>
</tr>
<tr>
<td>%R</td>
<td>Equivalent to %H:%M</td>
</tr>
<tr>
<td>%S</td>
<td>The seconds (00-59)</td>
</tr>
<tr>
<td>%t</td>
<td>A tab character</td>
</tr>
<tr>
<td>%T</td>
<td>Equivalent to %H:%M:%S</td>
</tr>
<tr>
<td>Placeholder</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>%x</td>
<td>The date, using the locale's date format.</td>
</tr>
<tr>
<td>%X</td>
<td>The time, using the locale's time format.</td>
</tr>
<tr>
<td>%w</td>
<td>The ordinal number of the day of the week (0-6), with Sunday = 0.</td>
</tr>
<tr>
<td>%y</td>
<td>The year within century (0-99)</td>
</tr>
<tr>
<td>%Y</td>
<td>The year, including the century (for example, 1991)</td>
</tr>
</tbody>
</table>

### Table 127: Datetime formatting examples

<table>
<thead>
<tr>
<th>Format String</th>
<th>Datetime value</th>
<th>Result string</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d/%m/%Y %H:%M</td>
<td>2011-10-24 11:23:45</td>
<td>24/10/2011 11:23</td>
</tr>
<tr>
<td>(%a.) %b. %d, %Y</td>
<td>1999-09-23</td>
<td>(Thu.) Sep. 23, 1999</td>
</tr>
</tbody>
</table>

### Formatting INTERVAL values

Interval values must be formatted when converted to strings.

#### When does INTERVAL formatting take place?

Interval formatting occurs when converting an `INTERVAL` to a string, for example in a `LET`, `DISPLAY` or `PRINT` instruction, and when displaying interval values in form fields.

By default, `INTERVAL` values are formatted in the ISO format, and depending on the interval class and type.

For example, an `INTERVAL YEAR(4) TO MONTH` will be formatted as:

```
[+] yyyy-mm
```

For an `INTERVAL DAY(n) TO FRACTION(5)`, the default format is:

```
[+] ldddd hh:mm:ss.fffffff
```

An `INTERVAL` value can be formatted with the `util.Interval.format()` method:

```
IMPORT util
MAIN
  DEFINE iv INTERVAL DAY(6) TO MINUTE
  LET iv = "-157 11:23"
  DISPLAY util.Interval.format(iv, "%d %H:%M")
END MAIN
```

This code example produces the following output:

```
-157 11:23
```

### Converting strings to INTERVAL values

When a string represents a interval value is ISO format, it can be directly converted to a `INTERVAL`:

```
DEFINE iv INTERVAL HOUR(6) TO FRACTION(5)
LET iv = "2023:34:56.82373"
```
If you need to convert a string that does not follow the ISO format, use the `util.Interval.parse()` method, by specifying a format string:

```java
DEFINE iv INTERVAL DAY(6) TO FRACTION(5)
LET iv = util.Interval.parse("-7467 + 23:45:34.12345", "%d + %H:%M:%S%F5")
```

### Formatting symbols for INTERVAL values

When formatting INTERVAL values, the format-string of the `util.Interval.parse()` and `util.Interval.format()` methods consists of a set of place holders that represent the different parts of an interval value (year, month, day, hour, minute, second and fraction).

**Table 128: Format-string symbols for INTERVAL values** on page 325 shows the formatting symbols for INTERVAL expressions. Any character different from the placeholders described in this table is interpreted as a literal and will appear as-is in the resulting string.

#### Table 128: Format-string symbols for INTERVAL values

<table>
<thead>
<tr>
<th>Placeholder</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%Y</td>
<td>Years (0-999999999)</td>
</tr>
<tr>
<td>%m</td>
<td>Months (0-999999999 if highest INTERVAL qualifier is MONTH(n), 0-11 otherwise)</td>
</tr>
<tr>
<td>%d</td>
<td>Days (0-999999999)</td>
</tr>
<tr>
<td>%H</td>
<td>Hours (0-999999999 if highest INTERVAL qualifier is HOUR(n), 00-23 otherwise)</td>
</tr>
<tr>
<td>%M</td>
<td>Minutes (0-999999999 if highest INTERVAL qualifier is MINUTE(n), 00-59 otherwise)</td>
</tr>
<tr>
<td>%S</td>
<td>Seconds (0-999999999 if highest INTERVAL qualifier is SECOND(n), 00-59 otherwise)</td>
</tr>
<tr>
<td>%F_n</td>
<td>The fractional part of a second, where n specifies the number of digits in the fractional part (1 to 5)</td>
</tr>
<tr>
<td>%t</td>
<td>A tab character</td>
</tr>
<tr>
<td>%n</td>
<td>A newline character</td>
</tr>
</tbody>
</table>

#### Table 129: Interval formatting examples

<table>
<thead>
<tr>
<th>Format String</th>
<th>Interval value</th>
<th>Result string</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d days %H:%M</td>
<td>54561 11:23</td>
<td>54561 days 11:23</td>
</tr>
<tr>
<td>%d days %H:%M:%S%F5</td>
<td>54561 11:23:45.12345</td>
<td>54561 days 11:23:45.12345</td>
</tr>
<tr>
<td>[%Y years and %m months]</td>
<td>1023-03</td>
<td>[1023 years and 03 months]</td>
</tr>
</tbody>
</table>

### Literals

Describes the syntax of literals (constant values) to be used in sources.

#### Integer literals

Integer literals define a whole number in an expression.

**Syntax**

```
[+|-] digit [...]
```
1. *digit* is a digit character from '0' to '9'.

**Usage**

Integer literals are in base-10 notation, without blank spaces and commas and without a decimal point.

Integer literals can be used to specify values for `DECIMAL(P, 0)`, `BIGINT`, `INTEGER`, `SMALLINT` and `TINYINT` data types.

**Example**

```
MAIN
  DEFINE n INTEGER
  LET n = 1234567
END MAIN
```

**Related concepts**

=Integer expressions on page 333

This section covers integer expression evaluation rules.

**Numeric literals**

Numeric literals define values with a decimal part in an expression.

**Syntax**

```
[+|-] digit[...]. digit[...] [e|E] [+|-] digit[...]
```

1. *digit* is a digit character from '0' to '9'.
2. Note that the decimal separator is always a dot, independently from DBMONEY.
3. The E notation can be used to specify the exponent.

**Usage**

Numeric/decimal literals in base-10 notation, without blank spaces and commas, with a decimal part after a dot.

Numeric literals can be used to specify values for `DECIMAL(P, S)`, `MONEY(P, S)`, `FLOAT` and `SMALLFLOAT` data types.

**Example**

```
MAIN
  DEFINE n DECIMAL(10, 2)
  LET n = 12345.67
  LET n = -1.23456e-10
END MAIN
```

**Related concepts**

=Numeric expressions on page 334

This section covers numeric expression evaluation rules.

**Text literals**

Text literals define a character string in an expression.

**Syntax 1 (using double quotes)**

```
" char [...] "
```
Syntax 2 (using single quotes)

' char [...] ' 

1. *char* is any character supported in the current locale, or a \ backslash escape character as described below:
   - `\`: the backslash character.
   - `"`: double-quote character.
   - `'`: single-quote character.
   - `\n`: newline character.
   - `\r`: carriage-return character.
   - `\0`: null character.
   - `\f`: form-feed character.
   - `\t`: tab character.
   - `\xNN`: ASCII character defined by the hexadecimal code `NN`.

Usage

A text literal (or character string literal) defines a character string constant containing valid characters in the current application character set.

The application character set is defined by the current locale.

A text literal can be written on multiple lines, the compiler merges lines by removing the newline character.

An empty string (`""`) is equivalent to `NULL`.

The escape character is the backslash character (`\`).

When using single quotes as delimiters, double quotes can be used as is inside the string, while single quotes must be doubled or escaped with a backslash:

```
DISPLAY '  2 double quotes: " "  2 single quotes: ' ' 
```

When using double quotes as delimiters, single quotes can be used as is inside the string, while double quotes must be doubled or escaped with a backslash:

```
DISPLAY "  2 double quotes: " "  2 single quotes: ' ' 
```

Special characters can be specified with backslash escape symbols. Use for example `\n` to insert a new-line character in a string literal:

```
DISPLAY "First line\nSecond line"
```

The `\xNN` hexadecimal notation allows you to specify control characters in a string literal. Only ASCII codes (`<=0x7F`) are allowed.

Example

```
MAIN
  DISPLAY "Some text in double quotes"
  DISPLAY 'Some text in single quotes'
  DISPLAY "Include double quotes: " " "
  DISPLAY 'Include single quotes: " " '
  DISPLAY 'Insert a newline character here: \n and continue with text.'
  DISPLAY "This is a text on multiple
```
lines.\You can insert a newline with back-slash at the end of the line."
IF "" IS NULL THEN
  DISPLAY 'Empty string is NULL'
END IF
END MAIN

Related concepts
Localized strings on page 538
Localized strings provide a means of writing applications in which the text of strings can be customized on site.

String expressions on page 334
This section covers string expression evaluation rules.

Localization on page 512
Localization support allows you to implement programs that follow specific language and cultural rules.

MDY(m,d,y) literals
MDY() literals define a DATE literal in an expression.

Syntax

\[ \text{MDY( month-integer, day-integer, year-integer )} \]

1. \text{month-integer} is an integer literal representing the month (1-12).
2. \text{day-integer} is an integer literal representing the day in the month (1-31).
3. \text{year-integer} is an integer literal representing a four-digit year.

Usage
Date MDY() literals are used to define the initializer value for a variable or the value of a constant of type DATE.

Note: Do not confuse the MDY() literal with an the MDY() operator.

Example

MAIN
  DEFINE d DATE = MDY(12,24,2018)
  DISPLAY "Date = ", d
END MAIN

Related concepts
Variable default values on page 395
Variables get a default value when defined.

Datetime literals
Datetime literals define date/time value in an expression.

Syntax

\[ \text{DATETIME ( dtrep ) quall TO qual2[(scale)]} \]

where \text{quall} can be one of:

\[ \begin{align*}
\text{YEAR} \\
\text{MONTH} \\
\text{DAY} \\
\text{HOUR} \\
\text{MINUTE}
\end{align*} \]
and \textit{qual2} can be one of:

- \textsc{year}
- \textsc{month}
- \textsc{day}
- \textsc{hour}
- \textsc{minute}
- \textsc{second}
- \textsc{fraction}
- \textsc{fraction}(1)
- \textsc{fraction}(2)
- \textsc{fraction}(3)
- \textsc{fraction}(4)
- \textsc{fraction}(5)

1. \textit{dtrep} is the datetime value representation in normalized format (YYYY-MM-DD hh:mm:ss.ffffff).
2. \textit{scale} defines the number of significant digits of the fractions of a second.
3. \textit{qual1} and \textit{qual2} qualifiers define the precision of the \textsc{datetime} literal.

\section*{Usage}
A datetime literal is specified with the \textsc{datetime()} notation, and is typically used in interval or datetime expressions, or to assign a \textsc{datetime} variable.

In order to get the current date and time, use the \textsc{current} operator.

\section*{Example}

\begin{verbatim}
MAIN
    DEFINE d1 DATETIME YEAR TO SECOND
    DEFINE d2 DATETIME HOUR TO FRACTION(5)
    LET d1 = DATETIME( 2002-12-24 23:55:56 ) YEAR TO SECOND
    LET d2 = DATETIME( 23:44:55.34532 ) HOUR TO FRACTION(5)
END MAIN
\end{verbatim}

\section*{Related concepts}

- \textbf{Datetime expressions} on page 335

This section covers date-time expression evaluation rules.

\section*{Interval literals}
Interval literals define an interval value in an expression.

\subsection*{Syntax 1: year-month class interval}

\begin{verbatim}
INTERVAL ( inrep ) \textsc{year}[(precision)] TO \textsc{month}
INTERVAL ( inrep ) \textsc{year}[(precision)] TO \textsc{year}
INTERVAL ( inrep ) \textsc{month}[(precision)] TO \textsc{month}
\end{verbatim}

\subsection*{Syntax 2: day-time class interval}

\begin{verbatim}
INTERVAL ( inrep ) \textsc{day}[(precision)] TO \textsc{fraction}[(scale)]
INTERVAL ( inrep ) \textsc{day}[(precision)] TO \textsc{second}
INTERVAL ( inrep ) \textsc{day}[(precision)] TO \textsc{minute}
INTERVAL ( inrep ) \textsc{day}[(precision)] TO \textsc{hour}
INTERVAL ( inrep ) \textsc{day}[(precision)] TO \textsc{day}
\end{verbatim}
INTERVAL (inrep) HOUR[(precision)] TO FRACTION[(scale)]
INTERVAL (inrep) HOUR[(precision)] TO SECOND
INTERVAL (inrep) HOUR[(precision)] TO MINUTE
INTERVAL (inrep) HOUR[(precision)] TO HOUR
INTERVAL (inrep) MINUTE[(precision)] TO FRACTION[(scale)]
INTERVAL (inrep) MINUTE[(precision)] TO SECOND
INTERVAL (inrep) MINUTE[(precision)] TO MINUTE
INTERVAL (inrep) SECOND[(precision)] TO FRACTION[(scale)]
INTERVAL (inrep) SECOND[(precision)] TO SECOND
INTERVAL (inrep) FRACTION TO FRACTION[(scale)]

1. inrep is the representation of the interval value in normalized format (YYYY-MM or DD hh:mm:ss.ffffff, depending on the interval class).

2. precision defines the number of significant digits of the first qualifier, it must be an integer from 1 to 9. For YEAR, the default is 4. For all other time units, the default is 2. For example, YEAR(5) indicates that the INTERVAL can store a number of years with up to 5 digits.

3. scale defines the scale of the fractional part, it can be 1, 2, 3, 4 or 5.

Usage
An interval literal is specified with the INTERVAL() notation, and is typically assigned in interval or datetime expressions, or to assign an INTERVAL variable.

Example

```
MAIN
  DEFINE i1 INTERVAL YEAR TO MONTH
  DEFINE i2 INTERVAL HOUR(5) TO SECOND
  LET i1 = INTERVAL(345-5) YEAR TO MONTH
  LET i2 = INTERVAL(34562:22:33) HOUR(5) TO SECOND
END MAIN
```

Related concepts
Interval expressions on page 336
This section covers interval expression evaluation rules.

Expressions
Shows the possible expressions supported in the language.

Understanding expressions
This is an introduction to language expressions.

What is an expression?
An expression is a sequence of operands, operators, and parentheses that the runtime system can evaluate as a single value. Operands are program variables, constants, functions returning a single value and literal values. Operators are used for arithmetic or string manipulation, and the parentheses are used to overwrite precedence of operators.

Language and SQL expressions
Expressions in SQL statements are evaluated by the database server, not by the runtime system. The set of operators that can appear in SQL expressions resembles the set of language operators, but they are not identical. A program can include SQL operators, but these are restricted to SQL statements. Similarly, most SQL operands are not valid in program expressions. The SQL identifiers of databases, tables, or columns can appear in a LIKE clause or field name
in program instructions, provided that these SQL identifiers comply with the naming rules of language. Here are some examples of SQL operands and operators that cannot appear in other language expressions:

- SQL identifiers, such as column names
- The SQL keywords USER and ROWID
- Built-in or aggregate SQL functions that are not part of the language
- The BETWEEN and IN operators
- The EXISTS, ALL, ANY, or SOME keywords of SQL expressions

Conversely, you cannot include language-specific operators in SQL expressions. For example:

- Arithmetic operators for exponentiation (**) and modulus (MOD)
- String operators ASCII, COLUMN, SPACE, SPACES, and WORDWRAP
- Field operators FIELD_TOUCHED(), GET_FLDBUF(), and INFIELD()
- The report operators LINENO and PAGENO

### Parentheses in expressions

Parentheses are used as in algebra, to override the default order of precedence of operators. In mathematics, this use of parentheses represents the "associative" operator. It is, however, a convention in computer languages to regard this use of parentheses as delimiters rather than as operators. (Do not confuse this use of parentheses to specify operator precedence with the use of parentheses to enclose arguments in function calls or to delimit other lists.)

In this example, the variable y is assigned the value of 2.

```sql
LET y = 15 MOD 3 + 2
```

In this example, y is assigned the value of 0 because the parentheses change the sequence of operations.

```sql
LET y = 15 MOD (3 + 2)
```

### Related concepts

- **Operators** on page 336
  This section describes basic syntax elements that can appear in expressions.

- **Literals** on page 325
  Describes the syntax of literals (constant values) to be used in sources.

### Boolean expressions

This section covers boolean expression evaluation rules.

Boolean expressions are a combination of AND, OR, NOT boolean operators, as well as comparison operators such as ==, >= or !=

The result of a boolean expression is a TRUE or FALSE boolean value, but it can also be NULL if one of the operands is NULL.

A boolean value is typically used in an IF block, WHILE block, or the WHEN clause in a CASE block.

**Note:** The language provides the TRUE and FALSE predefined constants to initialize boolean variables or return boolean values from functions.

There are three kind of boolean expressions:

- `expr AND expr`
- `expr OR expr`
- `NOT expr`

The (expr) operands of boolean expressions are boolean values.
**Important:** The syntax and semantics of boolean expressions in Genero BDL programs is not the same as *Boolean conditions* in SQL, as SQL statements are executed by the database engine.

If both operands are `NULL`, the result is `NULL`. If one of the operands is `NULL` and the other is non-null, the result depends on the operator type and the non-null value. For more details, see **AND**, **OR**, **NOT** operators.

**Tip:** Make sure that both operands using with a boolean or comparison operator are not null.

The following example shows a simple boolean expression using the **AND** operator:

```delphi
IF a AND b THEN
  DISPLAY "Both a and b are TRUE"
END IF
```

In the next example, a boolean expression uses two comparison expressions:

```delphi
IF (a == b) AND (a == c) THEN
  DISPLAY "a, b and c are equal"
END IF
```

The **NOT** operator will negate a boolean expression:

```delphi
IF NOT a THEN
  DISPLAY "a is FALSE"
END IF
```

Use a **BOOLEAN** variable to store the result of a boolean expression:

```delphi
MAIN
  DEFINE b BOOLEAN
  LET b = ( "a" == "b" )  -- result is FALSE
END MAIN
```

**Important:** It is bad practice to use non-boolean operands in boolean expressions, as in the following example:

```delphi
DEFINE var STRING, cnt INTEGER
IF var AND cnt>0 THEN
  ...
END IF
```

Good practice (for character strings for example) is to do the following:

```delphi
IF LENGTH(var)>0 AND cnt>0 THEN
or:
IF var IS NOT NULL AND cnt>0 THEN
```

If the operand is not of type boolean, it has to be converted to a boolean. If a conversion is required:

- Any numeric value evaluates to **FALSE**, if and only if the value is 0.
- Any character string value (**STRING**, **CHAR**, **VARCHAR**) follows the next rules:
  - If the string starts with a digit, then this conversion evaluates to **FALSE**, if and only if the string to integer conversion returns 0.
  - If the string does not start with a digit, then this conversion evaluates to **FALSE** if and only if the string has a length of 0.

**Note:** Consider using the expression `(LENGTH(string)>0) OR string IS NOT NULL`, to check that a string contains characters, or convert the string to a numeric variable and then test the numeric value.
• **DATE** values can be converted to integers. \( \text{MDY}(12, 31, 1899) = 0 \) and evaluates to **FALSE**. Any other date value is different from zero and evaluates to **TRUE**.

• Any other **data type** produces a conversion error and raises the runtime error **-1260**.

Below a more complex example of boolean expressions:

```
MAIN
  DEFINE r BOOLEAN, c INTEGER
  LET c = 4
  LET r = ( c != 5 ) AND ( c == 2 OR c == 4 )
  IF ( r AND canReadFile("config.txt") ) THEN
    DISPLAY "OK"
  END IF
END MAIN
```

If an expression that returns **NULL** is the operand of the **IS NULL** operator, the value of the boolean expression is **TRUE**:

```
MAIN
  DEFINE r INTEGER
  LET r = NULL
  IF r IS NULL THEN
    DISPLAY "TRUE"
  END IF
END MAIN
```

Boolean expressions in **CASE**, **IF**, or **WHILE** statements evaluate to **FALSE**, if any element of the comparison is **NULL**, except for operands of the **IS NULL** and the **IS NOT NULL** operator.

If you include a boolean expression in a context where the runtime system expects a number, the expression is evaluated, and is then converted to an integer by the rules **TRUE=1** and **FALSE=0**.

```
MAIN
  DEFINE r INTEGER
  LET c = 4
  LET r = 4 + (1 == 0)    -- result is 4.
END MAIN
```

**Related concepts**

**Type conversions** on page 311

Explains primitive data type conversion rules of the language.

**Integer expressions**

This section covers integer expression evaluation rules.

An integer expression evaluates to a whole number.

```
MAIN
  DEFINE r, c INTEGER
  LET c = 4
  LET r = c * ( 2 + c MOD 4 ) / getRowCount("customers")
END MAIN
```

The operands of an integer expression can be:

• An **integer literal**.

• A **variable** or **constant** of type **TINYINT, SMALLINT, INTEGER** or **BIGINT**.

• A **function** returning a single integer value.

• A **boolean expression**.

• The result of a **DATE** subtraction, as a number of days.
If an integer expression includes an operand whose value is not an integer data type, the runtime system attempts to convert the value to an integer following the data conversion rules.

If an element of an integer expression is NULL, the expression is evaluated to NULL.

Related concepts
Type conversions on page 311
Explains primitive data type conversion rules of the language.

Numeric expressions
This section covers numeric expression evaluation rules.

A numeric expression evaluates to a decimal value.

```
MAIN
  DEFINE r, c DECIMAL(10,2)
  LET c = 456.22
  LET r = c * 2 + ( c / 4.55 )
END MAIN
```

The operands of a numeric expression can be one of:

- An integer literal.
- A decimal literal.
- A variable or constant of numeric data type.
- A function returning a single numeric value.
- A boolean expression.
- The result of a DATE on page 295 subtraction, as a number of days.

If a number expression includes an operand whose value is not a numeric data type, the runtime system attempts to convert the value to a number following the data conversion rules.

If an element of a number expression is NULL, the expression is evaluated to NULL.

Related concepts
Type conversions on page 311
Explains primitive data type conversion rules of the language.

String expressions
This section covers string expression evaluation rules.

A string expression includes at least one character string value and evaluates to a string data type value.

```
MAIN
  DEFINE r, c VARCHAR(100)
  LET c = "abcdef"
  LET r = c[1,3] || ": " || TODAY USING "YYYY-MM-DD" || " " || length(c)
END MAIN
```

The data type of string expression result is STRING.

At least one of the operands in a string expression must be one of:

- A character string literal.
- A variable or constant of CHAR, VARCHAR, STRING or TEXT data type.
- A function returning a single character value.

Other operands whose values are not character string data types are converted to strings by following the data conversion rules.

If an element of a string expression is NULL, the expression is evaluated to NULL.
An empty string ("") is equivalent to NULL.

Related concepts
Type conversions on page 311
Explains primitive data type conversion rules of the language.

Date expressions
This section covers date expression evaluation rules.

A date expression evaluates to a DATE data type value.

```plaintext
MAIN
  DEFINE r, c DATE
  LET c = TODAY + 4
  LET r = ( c - 2 )
END MAIN
```

The operands of a date expression can be one of:

- A character string literal that can be evaluated to a date based on the DBDATE environment variable.
- A variable or constant of type DATE.
- A function returning a single date value.
- A unary + or - sign associated with an integer expression representing a number of days.
- The TODAY constant.
- A CURRENT expression with YEAR TO DAY qualifiers.
- An EXTEND expression with YEAR TO DAY qualifiers.

If a date expression includes an operand whose value is not a date data type, the runtime system attempts to convert the value to a date value following the data conversion rules.

If an element of an date expression is NULL, the expression is evaluated to NULL.

Related concepts
Type conversions on page 311
Explains primitive data type conversion rules of the language.

Datetime expressions
This section covers date-time expression evaluation rules.

A datetime expression evaluates to a DATETIME data type.

```plaintext
MAIN
  DEFINE r, c DATETIME YEAR TO SECOND
  LET c = CURRENT YEAR TO SECOND
  LET r = c + INTERVAL( 234-02 ) YEAR TO MONTH
END MAIN
```

The operands of a datetime expression can be one of:

- A datetime literal.
- A character string literal representing a datetime with the format YYYY-MM-DD hh:mm:ss.fffff.
- A variable or constant of DATETIME type.
- A function returning a single datetime value.
- A unary + or - sign associated with an interval expression.
- A CURRENT expression.
- An EXTEND expression.

If a datetime expression includes an operand whose value is not a datetime data type, the runtime system attempts to convert the value to a datetime value following the data conversion rules.
If an element of an integer expression is **NULL**, the expression is evaluated to **NULL**.

**Related concepts**
- [Type conversions](#) on page 311
  Explains primitive data type conversion rules of the language.

### Interval expressions

This section covers interval expression evaluation rules.

An interval evaluates to an **INTERVAL** data type.

```
MAIN
  DEFINE r, c INTERVAL HOUR TO MINUTE
  LET c = "12:45"
  LET r = c + ( DATETIME(14:02) HOUR TO MINUTE - DATETIME(10:43) HOUR TO MINUTE )
END MAIN
```

The operands of an interval expression must be one of:

- An **interval literal**.
- A **character string literal** representing an Interval with the format `YYYY-MM-DD hh:mm:ss.ffffff`.
- An integer expression using the **UNITS** operator.
- A **variable** or **constant** of **INTERVAL** type.
- A **function** returning a single interval value.
- The result of a **DATETIME** subtraction.

If an interval expression includes an operand whose value is not an interval data type, the runtime system attempts to convert the value to an interval value following the data conversion rules.

If an element of an integer expression is **NULL**, the expression is evaluated to **NULL**.

**Related concepts**
- [Type conversions](#) on page 311
  Explains primitive data type conversion rules of the language.

### Operators

This section describes basic syntax elements that can appear in expressions.

There are different sort of basic syntax elements such as operators for arithmetic, string and comparison, predefined variables and registers like **SQLSTATE**, and utility operators like **SFMT()** or **TODAY**.

Elements of an expressions are evaluated by following precedence rules, from highest to lowest, as described in the order of precedence list. Use () parentheses to instruct the runtime system to evaluate the expression in a different way than the default order of precedence.

**Related concepts**
- [Expressions](#) on page 330
  Shows the possible expressions supported in the language.
- [Order of precedence](#) on page 337
  The order of precedence defines in which order the elements of an expression are evaluated.
- [Parentheses: ()](#) on page 361
Parentheses ( ) force the evaluation of an expression before other operators.

**Order of precedence**

The order of precedence defines in which order the elements of an expression are evaluated.

The following list describes the precedence order of expression elements.

For example, the \texttt{MOD} operator has a higher precedence as the \texttt{*} operator. When computing an expression like \( (33 \mod 2 \times 5) \), the runtime system first evaluates \((33 \mod 2) = 1\) and then evaluates \((1 \times 5) = 5\). The order of evaluation can be changed this by using parentheses: \((33 \mod (2 \times 5)) = 3\).
<table>
<thead>
<tr>
<th>P</th>
<th>Syntax Element</th>
<th>A</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>CAST(var AS class)</td>
<td>N</td>
<td>Type casting</td>
<td>CAST(var AS java.lang.Number)</td>
</tr>
<tr>
<td>14</td>
<td>var INSTANCEOF class</td>
<td>L</td>
<td>Type checking</td>
<td>var INSTANCEOF java.lang.Boolean</td>
</tr>
<tr>
<td>13</td>
<td>UNITS</td>
<td>L</td>
<td>Single-qualifier interval</td>
<td>(12) UNITS DAY</td>
</tr>
<tr>
<td>12</td>
<td>+ (unary)</td>
<td>R</td>
<td>Unary plus</td>
<td>+324</td>
</tr>
<tr>
<td>12</td>
<td>− (unary)</td>
<td>R</td>
<td>Unary minus</td>
<td>−324</td>
</tr>
<tr>
<td>11</td>
<td>**</td>
<td>L</td>
<td>Exponentiation</td>
<td>x ** 5</td>
</tr>
<tr>
<td>11</td>
<td>MOD</td>
<td>L</td>
<td>Modulus</td>
<td>x MOD 2</td>
</tr>
<tr>
<td>10</td>
<td>*</td>
<td>L</td>
<td>Multiplication</td>
<td>x * y</td>
</tr>
<tr>
<td>10</td>
<td>/</td>
<td>L</td>
<td>Division</td>
<td>x / y</td>
</tr>
<tr>
<td>9</td>
<td>+</td>
<td>L</td>
<td>Addition</td>
<td>x + y</td>
</tr>
<tr>
<td>9</td>
<td>−</td>
<td>L</td>
<td>Subtraction</td>
<td>x − y</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td>L</td>
</tr>
<tr>
<td>7</td>
<td>LIKE</td>
<td>R</td>
<td>String comparison</td>
<td>mystring LIKE &quot;A%&quot;</td>
</tr>
<tr>
<td>7</td>
<td>MATCHES</td>
<td>R</td>
<td>String comparison</td>
<td>mystring MATCHES &quot;A*&quot;</td>
</tr>
<tr>
<td>6</td>
<td>&lt;</td>
<td>L</td>
<td>Less than</td>
<td>var &lt; 100</td>
</tr>
<tr>
<td>6</td>
<td>&lt;=</td>
<td>L</td>
<td>Less then or equal to</td>
<td>var &lt;= 100</td>
</tr>
<tr>
<td>6</td>
<td>&gt;</td>
<td>L</td>
<td>Greater than</td>
<td>var &gt; 100</td>
</tr>
<tr>
<td>6</td>
<td>&gt;=</td>
<td>L</td>
<td>Greater than or equal to</td>
<td>var &gt;= 100</td>
</tr>
<tr>
<td>6</td>
<td>==</td>
<td>L</td>
<td>Equals</td>
<td>var == 100</td>
</tr>
</tbody>
</table>
In this table, the $P$ column defines the precedence, from highest (14) to lowest (1). Note that some operators have the same precedence (i.e. are equivalent in evaluation order). The $A$ column defines the direction of association ($L=$Left, $R=$Right, $N=$None).

**Operator usage context**
Some operators are specific to a context.

**Pure SQL syntax elements**
The following are related to SQL syntax and not part of the language:
- BETWEEN $expr$ AND $expr$
- IN ($expr$ [ , ..' ])

**Report Routine syntax elements**
The following are only available in the FORMAT section of report routines:
- PAGENO
- WORDWRAP

See Report Definition for more details.

**List of expression elements**
This topic is the reference for language expressions.

**Comparison operators**
Comparison operators allow you to compare two values, to include the greater than, less than and equal to functions.

**Table 131: Comparison operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS NULL on page 340</td>
<td>The IS NULL operator checks for NULL values.</td>
</tr>
<tr>
<td>LIKE on page 340</td>
<td>The LIKE operator returns TRUE if a string matches a given mask.</td>
</tr>
<tr>
<td>MATCHES on page 341</td>
<td>The MATCHES operator returns TRUE if a string matches a given mask.</td>
</tr>
<tr>
<td>Equal to ($=$) on page 343</td>
<td>The $==$ operator checks for equality of two expressions or for two record variables.</td>
</tr>
<tr>
<td>Different from ($!=$) on page 344</td>
<td>The $!=$ operator checks for non-equality of two expressions or for two record variables.</td>
</tr>
<tr>
<td>Lower ($&lt;$) on page 345</td>
<td>The $&lt;$ operator is provided to test whether a value or expression is lower than another.</td>
</tr>
<tr>
<td>Lower or equal ($&lt;=$) on page 345</td>
<td>The $&lt;=$ operator is provided to test whether a value or expression is lower than or equal to another.</td>
</tr>
<tr>
<td>Greater ($&gt;$) on page 346</td>
<td>The $&gt;$ operator is provided to test whether a value or expression is greater than another.</td>
</tr>
<tr>
<td>Greater or equal ($&lt;=$) on page 346</td>
<td>The $&lt;=$ operator is provided to test whether a value or expression is greater than or equal to another.</td>
</tr>
<tr>
<td>NVL() on page 347</td>
<td>The NVL () operator returns the second parameter if the first argument evaluates to NULL.</td>
</tr>
</tbody>
</table>
### IS NULL
The IS NULL operator checks for NULL values.

#### Syntax

```plaintext
expr IS [NOT] NULL
```

1. `expr` can be any expression supported by the language.
2. The NOT keyword negates the comparison.

#### Usage

The IS NULL operator can be used to test whether the left-hand expression is null, while IS NOT NULL operator can be used to test for non-null values.

This operator applies to most data types, except complex types like `BYTE` and `TEXT`.

#### Example

```plaintext
MAIN
  DEFINE n INTEGER
  LET n = NULL
  IF n IS NULL THEN
    DISPLAY "The variable is NULL."
  END IF
END MAIN
```

### Related concepts

- **NULL** on page 555
  The NULL constant is provided as the "nil" value.
- **Expressions** on page 330
  Shows the possible expressions supported in the language.
- **NVL()** on page 347
  The NVL() operator returns the second parameter if the first argument evaluates to NULL.

### LIKE
The LIKE operator returns TRUE if a string matches a given mask.

#### Syntax

```plaintext
expr [NOT] LIKE mask [ESCAPE "char"]
```

1. `expr` is any character string expression.
2. `mask` is a character string expression defining the filter.
3. `char` is a single char specifying the escape symbol.
4. The NOT keyword negates the comparison.

#### Usage

The `mask` can be any combination of characters, including the % and _ wildcards:
• The % percent character matches any string of zero or more characters.
• The _ underscore character matches any single character.

The ESCAPE clause can be used to define an escape character different from the default backslash. It must be enclosed in single or double quotes.

A backslash (or the escape character specified by the ESCAPE clause) makes the operator treat the next character as a literal character, even if it is one of the special symbols in the mask list. This allows you to search for %, _ or \ characters.

Do not confuse with the LIKE clause of the DEFINE instruction.

Note: LIKE is also an SQL operator evaluated by the database server. Such expression may have a different behavior than the LIKE operator of the language.

If you need to escape a wildcard character, keep in mind that a string constant must also escape the backslash character. As a result, if you want to pass a backslash to the LIKE operator (by using backslash as default escape character), you need to write four backslashes in the original string constant.

The next table shows some examples of string constants used in the source code and their equivalent LIKE pattern:

### Table 132: Examples of string constants used in the source code and their equivalent LIKE pattern

<table>
<thead>
<tr>
<th>Original String Constant</th>
<th>Equivalent LIKE pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;%&quot;</td>
<td>%</td>
<td>Matches any character in a non-empty string.</td>
</tr>
<tr>
<td>&quot;_&quot;</td>
<td>_</td>
<td>Matches a single character.</td>
</tr>
<tr>
<td>&quot;abc%&quot;</td>
<td>abc%</td>
<td>Starts with abc.</td>
</tr>
<tr>
<td>&quot;#abc&quot;</td>
<td>%abc</td>
<td>Ends with abc.</td>
</tr>
<tr>
<td>&quot;%abc%&quot;</td>
<td>%abc%</td>
<td>Contains abc.</td>
</tr>
<tr>
<td>&quot;abc__&quot;</td>
<td>abc__</td>
<td>Strings equals abc followed by two additional characters.</td>
</tr>
<tr>
<td>&quot;\&quot;</td>
<td>\</td>
<td>Contains a single star character (the % wildcard is escaped)</td>
</tr>
<tr>
<td>&quot;%abc\\def%&quot;</td>
<td>abc\def%</td>
<td>Contains abc followed by a backslash followed by def (the backslash is escaped)</td>
</tr>
</tbody>
</table>

**Example**

```plaintext
MAIN
  IF "abcdef" LIKE "a%e_" THEN
    DISPLAY "The value matches."
  END IF
END MAIN
```

**Related concepts**

**Expressions** on page 330
Shows the possible expressions supported in the language.

**MATCHES**
The MATCHES operator returns TRUE if a string matches a given mask.

**Syntax**

```plaintext
expr [NOT] MATCHES mask [ ESCAPE "char" ]
```

1. `expr` is any character string expression.
2. *mask* is a character string expression defining the filter.
3. *char* is a single char specifying the escape symbol (default is backslash, don't forget to double it!).
4. The **NOT** keyword negates the comparison.

**Usage**

The *mask* can be any combination of characters, including the *, ?, [ ], – and ^ wildcards:

- The * star character matches any string of zero or more characters.
- The ? question mark matches any single character.
- The [ ] brackets match any enclosed character.
- Inside [ ], the – (hyphen) between characters means a range of characters.
- Inside [ ], the ^ An initial caret matches any character that is not listed.

The **ESCAPE** clause can be used to define an escape character different from the default backslash. It must be enclosed in single or double quotes.

A backslash (or the escape character specified by the **ESCAPE** clause) makes the operator treat the next character as a literal character, even if it is one of the special symbols in the *mask* list. This allows you to search for wildcard charachers such as *, ?, [, ] or \.

If you need to escape a wildcard character, keep in mind that a string constant in the source code must also escape the backslash character. As a result, if you want to pass a backslash to the **MATCHES** operator (by using backslash as default escape character), you need to write four backslashes in the original string constant.

To include the hyphen (–) or caret (^) characters in a [ ] set of characters in the pattern, put the hyphen at the beginning of the list and the caret after the first position, there is no need to escape these characters inside [ ].

The next table shows some examples of string constants used in the source code and their equivalent **MATCHES** pattern:

**Table 133: String constants used in the source code and their equivalent **MATCHES** pattern**

<table>
<thead>
<tr>
<th>String constant in source code</th>
<th><strong>MATCHES</strong> pattern in memory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;*&quot;</td>
<td>*</td>
<td>Matches any character in a non-empty string.</td>
</tr>
<tr>
<td>&quot;?&quot;</td>
<td>?</td>
<td>Matches a single character.</td>
</tr>
<tr>
<td>&quot;abc*&quot;</td>
<td>abc*</td>
<td>Starts with abc.</td>
</tr>
<tr>
<td>&quot;*abc&quot;</td>
<td>*abc</td>
<td>Ends with abc.</td>
</tr>
<tr>
<td>&quot;<em>abc</em>&quot;</td>
<td><em>abc</em></td>
<td>Contains abc.</td>
</tr>
<tr>
<td>&quot;abc??&quot;</td>
<td>abc??</td>
<td>Starts with abc, followed by two additional characters.</td>
</tr>
<tr>
<td>&quot;[a-z]*&quot;</td>
<td>[a-z]*</td>
<td>Starts with a letter in the range a to z.</td>
</tr>
<tr>
<td>&quot;[^0-9]*&quot;</td>
<td>[^0-9]*</td>
<td>Must not start with a digit.</td>
</tr>
<tr>
<td>&quot;*&quot;</td>
<td>*</td>
<td>Contains a single star character (the * wildcard is escaped)</td>
</tr>
<tr>
<td>&quot;<em>abc\\def</em>&quot;</td>
<td><em>abc\def</em></td>
<td>Contains abc followed by a backslash followed by def (the backslash is escaped)</td>
</tr>
<tr>
<td>&quot;* [-^] *&quot;</td>
<td>* [-^]*</td>
<td>Contains hyphen or caret characters.</td>
</tr>
</tbody>
</table>
String constant in source code | MATCHES pattern in memory | Description
---|---|---
`*[ -/:@[-^\{\~]*` | `*[ -/:@[-^\{\~]*` | Contains characters that are not valid for an identifier (to be used with NOT MATCHES). Note that the pattern is using character ranges with – hyphen:
- `-` (space to slash)
- `:` (colon to minus)
- `[` (open sb to caret)
- `` (backquote)
- `{` (open cb to tilde)

**Example**

```plaintext
MAIN
  IF "55f-plot" MATCHES "55[a-z]-*" THEN
    DISPLAY "Item reference format is correct."
  END IF
END MAIN
```

**Related concepts**

Expressions on page 330
Shows the possible expressions supported in the language.

**Equal to (==)**
The `==` operator checks for equality of two expressions or for two record variables.

**Syntax 1: Expression comparison**

```plaintext
expr == expr
```

1. `expr` can be any expression supported by the language.

**Syntax 2: Record comparison**

```plaintext
record1.* == record2.*
```

1. `record1` and `record2` are records with the same structure.

**Usage**

The `==` operator evaluates whether two expressions or two records are identical.

A single equal sign (=`) can be used as an alias for the `==` operator.

The syntax comparing two expressions applies to most data types, except complex types like `BYTE` and `TEXT`.

When comparing simple expressions (`expr == expr`), the result of the operator is `NULL` when one of the operands is `NULL`.

When comparing two `records` using the second syntax, the runtime system compares all corresponding members of the records. If a pair of members are different, the result of the operator is `FALSE`. When two corresponding members are `NULL`, they are considered as equal. This second syntax allows you to compare all members of records, but records must have the same structure.

**Example**

```plaintext
MAIN
```
DEFINE n INTEGER
    LET n=512
    IF n==512 THEN
        DISPLAY "The variable equals 512."
    END IF
END MAIN

Related concepts
Expressions on page 330
Shows the possible expressions supported in the language.

Different from (!=)
The != operator checks for non-equality of two expressions or for two record variables.

Syntax 1: Expression comparison

\[ expr /= expr \]

1. *expr* can be any expression supported by the language.

Syntax 2: Record comparison

\[ record1.* /= record2.* \]

1. *record1* and *record2* are records with the same structure.

Usage
The != operator evaluates whether two expressions or two records are different.

Note: \(<\text{>}\) is a synonym for !=.

A less-than sign followed by a greater-than sign (<>\text{<}) can be used as an alias for the != operator.

The syntax comparing two expressions applies to most data types, except complex types like BYTE and TEXT.

When comparing simple expressions (\(expr /= expr\)), the result of the operator is NULL when one of the operands is NULL.

When comparing two records with the second syntax, the runtime system compares all corresponding members of the records. If one pair of members are different, the result of the operator is TRUE. When two corresponding members are NULL, they are considered as equal. This second syntax allows you to compare all members of records, but records must have the same structure.

Example

MAIN
    DEFINE n INTEGER
    LET n==512
    IF n!=32 THEN
        DISPLAY "The variable is not equal to 32."
    END IF
END MAIN

Related concepts
Expressions on page 330
Shows the possible expressions supported in the language.

**Lower (<)**
The `<` operator is provided to test whether a value or expression is lower than another.

**Syntax**

```
expr < expr
```

**Usage**

Applies to most data types, except complex types such as `BYTE` and `TEXT`. If one of the operands is `NULL`, the comparison expression evaluates to `NULL`.

**Example**

```
MAIN
   DEFINE n INT
   LET n = 45
   IF n < 100 THEN
      DISPLAY "The variable is lower than 100."
   END IF
END MAIN
```

**Related concepts**

*Expressions* on page 330
Shows the possible expressions supported in the language.

**Lower or equal (<=)**
The `<=` operator is provided to test whether a value or expression is lower than or equal to another.

**Syntax**

```
expr <= expr
```

**Usage**

Applies to most data types, except complex types such as `BYTE` and `TEXT`. If one of the operands is `NULL`, the comparison expression evaluates to `NULL`.

**Example**

```
MAIN
   DEFINE n INT
   LET n = 100
   IF n <= 100 THEN
      DISPLAY "The variable is lower than or equal to 100."
   END IF
END MAIN
```

**Related concepts**

*Expressions* on page 330
Shows the possible expressions supported in the language.

**Greater (>)**
The `>` operator is provided to test whether a value or expression is greater than another.

**Syntax**

```
expr > expr
```

**Usage**

Applies to most data types, except complex types such as `BYTE` and `TEXT`.

If one of the operands is `NULL`, the comparison expression evaluates to `NULL`.

**Example**

```plaintext
MAIN
    DEFINE n INT
    LET n = 200
    IF n > 100 THEN
    END IF
    DISPLAY "The variable is greater than 100."
END MAIN
```

**Related concepts**

[Expressions](#) on page 330
Shows the possible expressions supported in the language.

**Greater or equal (>=)**
The `>=` operator is provided to test whether a value or expression is greater than or equal to another.

**Syntax**

```
expr >= expr
```

**Usage**

Applies to most data types, except complex types such as `BYTE` and `TEXT`.

If one of the operands is `NULL`, the comparison expression evaluates to `NULL`.

**Example**

```plaintext
MAIN
    DEFINE n INT
    LET n = 100
    IF n >= 100 THEN
    END IF
    DISPLAY "The variable is greater than or equal to 100."
END MAIN
```

**Related concepts**

[Expressions](#) on page 330
NVL()
The `NVL()` operator returns the second parameter if the first argument evaluates to `NULL`.

Syntax

```
NVL(main-expr, subst-expr)
```

1. `main-expr` and `subst-expr` are any expression supported by the language.

Usage

The `NVL()` operator evaluates the first argument, and returns the result if the value is not null, otherwise it returns the second argument.

This allows you to write the equivalent of the following `IF` statement, in a simple scalar expression:

```
IF main-expr IS NOT NULL THEN
  RETURN main-expr
ELSE
  RETURN subst-expr
END IF
```

Example

```
MAIN
  DEFINE var VARCHAR(100)
  LET var = arg_val(1)
  DISPLAY "The argument value is: ", NVL(var, "NULL")
END MAIN
```

Related concepts

Expressions on page 330
Shows the possible expressions supported in the language.

IIF()
The `IIF()` operator returns the second or third parameter depending on the boolean expression given as first argument.

Syntax

```
IIF(bool-expr, true-expr, false-expr)
```

1. `bool-expr` is a boolean expression.
2. `true-expr` and `false-expr` are language expressions.

Usage:

The `IIF()` operator evaluates the first argument, the returns the second argument if the first argument is true, otherwise it returns the third argument.

This allows you to write the equivalent of the following `IF` statement, in a simple scalar expression:

```
IF bool-expr THEN
  RETURN true-expr
ELSE
  RETURN false-expr
END IF
```
Example

```
MAIN
    DEFINE var VARCHAR(10)
    LET var = arg_val(1)
    DISPLAY IIF(var == "A", "Accepted", "Rejected")
END MAIN
```

Related concepts

**Boolean expressions** on page 331
This section covers boolean expression evaluation rules.

**Boolean operators**
Boolean operators include NOT, AND and OR.

### Table 134: Logical operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT on page 348</td>
<td>The <strong>NOT</strong> operator performs a logical negation to invert a boolean expression.</td>
</tr>
<tr>
<td>AND on page 348</td>
<td>The <strong>AND</strong> operator is the logical intersection operator.</td>
</tr>
<tr>
<td>OR on page 349</td>
<td>The <strong>OR</strong> operator is the logical union operator.</td>
</tr>
</tbody>
</table>

**NOT**
The **NOT** operator performs a logical negation to invert a boolean expression.

**Syntax**

```
NOT bool-expr
```

1. **bool-expr** is a boolean expression.

**Usage**
The **NOT** operator is typically used to invert the value of a boolean expression.

If the operand is **NULL**, the negation expression evaluates to **NULL**.

**Example**

```
MAIN
    IF NOT ( 256 == 257 ) THEN
        DISPLAY "This line should display"
    END IF
END MAIN
```

**Related concepts**

**Boolean expressions** on page 331
This section covers boolean expression evaluation rules.

**AND**
The **AND** operator is the logical intersection operator.

**Syntax**

```
bool-expr AND bool-expr
```
1. *bool-expr* is a boolean expression.

**Usage**

The **AND** operator is used to perform a logical conjunction on two boolean expressions.

Possible values of the operands are **TRUE**, **FALSE** and **NULL**. If one of the operands is **NULL**, the logical AND expression evaluates to **NULL** or **FALSE**, depending on the value of the second operand:

- The result of **AND** is **TRUE**, if both operands are **TRUE**.
- The result of **AND** is **FALSE**, if one of the operands is **FALSE** and the other operand is **TRUE**, **FALSE** or **NULL**.
- The result of **AND** is **NULL**, if one of the operands is **NULL** and the other operand is **NULL** or **TRUE**.

By default, the runtime system evaluates both operands on the left and right side of the **AND** keyword. This is the traditional behavior of the Genero language, but in fact the right operand does not need to be evaluated if the first operand evaluates to **FALSE**. This method is called *short-circuit evaluation*, and can be enabled by adding the `OPTIONS SHORT CIRCUIT` clause at the beginning of the module.

**Example**

```text
MAIN
  IF 256!=257 AND 257==257 THEN
    DISPLAY "This line should display"
  END IF
END MAIN
```

**Related concepts**

[**Boolean expressions**](#) on page 331

This section covers boolean expression evaluation rules.

**OR**

The **OR** operator is the logical union operator.

**Syntax**

```
bool-expr OR bool-expr
```

1. *bool-expr* is a boolean expression.

**Usage**

The **OR** operator is used to perform a logical disjunction on two boolean expressions.

Possible values of the operands are **TRUE**, **FALSE** and **NULL**. If one of the operands is **NULL**, the logical **OR** expression evaluates to **NULL** or **TRUE**, depending on the value of the second operand:

- The result of **OR** is **FALSE**, if both operands are **FALSE**.
- The result of **OR** is **TRUE**, if one of the operands is **TRUE** and the other operand is **TRUE**, **FALSE** or **NULL**.
- The result of **OR** is **NULL**, if one of the operands is **NULL** and the other operand is **NULL** or **FALSE**.

By default, the runtime system evaluates both operands on the left and right side of the **OR** keyword. This is the traditional behavior of the Genero language, but in fact the right operand does not need to be evaluated, if the first operand evaluates to **TRUE**. This method is called *short-circuit evaluation*, and can be enabled by adding the `OPTIONS SHORT CIRCUIT` clause at the beginning of the module.

**Example**

```text
MAIN
  IF TRUE OR FALSE THEN
    DISPLAY "This line should display"
  END IF
END MAIN
```
Related concepts
Boolean expressions on page 331
This section covers boolean expression evaluation rules.

Arithmetic operators
Arithmetic operators allow you to complete numeric operations, such as addition and subtraction.

Table 137: Arithmetic operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition (+) on page 350</td>
<td>The + operator adds a number to another.</td>
</tr>
<tr>
<td>Subtraction (-) on page 350</td>
<td>The – operator subtracts a number from another.</td>
</tr>
<tr>
<td>Multiplication (*) on page 351</td>
<td>The * operator multiplies a number with another.</td>
</tr>
<tr>
<td>Division (/) on page 351</td>
<td>The / operator divides a number by another.</td>
</tr>
<tr>
<td>Exponentiation (***) on page 352</td>
<td>The ** operator calculates an exponentiation.</td>
</tr>
<tr>
<td>MOD on page 352</td>
<td>The MOD operator calculates the modulus.</td>
</tr>
</tbody>
</table>

Addition (+)
The + operator adds a number to another.

Syntax

\[\text{num-expr} + \text{num-expr}\]

1. \(\text{num-expr}\) is a numeric expression.

Usage
Use the + operator to add two numeric values.

If one of the operands is NULL, the arithmetic expression evaluates to NULL.

Example

```plaintext
MAIN
  DISPLAY 100 + 200
END MAIN
```

Related concepts
Numeric expressions on page 334
This section covers numeric expression evaluation rules.

Subtraction (-)
The – operator subtracts a number from another.

Syntax

\[\text{num-expr} - \text{num-expr}\]

1. \(\text{num-expr}\) is a numeric expression.
Usage
Use the - operator to subtract a numeric value from another numeric value.
If one of the operands is NULL, the arithmetic expression evaluates to NULL.

Example

```plaintext
MAIN
  DISPLAY 100 - 200
END MAIN
```

Related concepts
Numeric expressions on page 334
This section covers numeric expression evaluation rules.

Multiplication (*)
The * operator multiplies a number with another.

Syntax

```plaintext
num-expr * num-expr
```

1. `num-expr` is a numeric expression.

Usage
Use the * operator to multiply a numeric value to another numeric value.
If one of the operands is NULL, the arithmetic expression evaluates to NULL.

Example

```plaintext
MAIN
  DISPLAY 100 * 200
END MAIN
```

Related concepts
Numeric expressions on page 334
This section covers numeric expression evaluation rules.

Division (/)
The / operator divides a number by another.

Syntax

```plaintext
num-expr / num-expr
```

1. `num-expr` is a numeric expression.

Usage
Use the / operator to divide a numeric value by another numeric value.
If one of the operands is NULL, the arithmetic expression evaluates to NULL.

Example

```plaintext
MAIN
  DISPLAY 100 / 200
```
**Related concepts**

**Numeric expressions** on page 334
This section covers numeric expression evaluation rules.

**Exponentiation (**)**
The ** operator calculates an exponentiation.

**Syntax**

\[ \text{num-expr} \text{ ** int-expr} \]

1. \textit{num-expr} is a numeric expression (real number).
2. \textit{int-expr} is the exponent, as a whole number.

**Usage**
The ** operator returns a value calculated by raising the left-hand operand to a power corresponding to the integer part of the right-hand operand.

\[ n \text{ ** e} = n \times n \times n \ldots \text{ (e times)} \]

If the right operand (the exponent) is a number with a decimal part, it is rounded to a whole integer before computing the exponentiation.

If one of the operands is \texttt{NULL}, the arithmetic expression evaluates to \texttt{NULL}.

**Tip:** The ** operator takes an integer as exponent. If you need to raise a number to an exponent that is a real number, use the \texttt{fgl_decimal_power()} utility function.

**Example**

```plaintext
MAIN
  DISPLAY 2 ** 8
  DISPLAY 10 ** 4
END MAIN
```

**Related concepts**

**Numeric expressions** on page 334
This section covers numeric expression evaluation rules.

\texttt{fgl_decimal_power()} on page 2277
Raises decimal to the power of the real exponent.

**MOD**
The MOD operator calculates the modulus.

**Syntax**

\[ \text{int-expr} \text{ MOD int-expr} \]

1. \textit{int-expr} is an integer expression.

**Usage**
The MOD operator returns the remainder, as an integer, from the division of the integer part of two numbers.

If the right operand is a number with a decimal part, it is rounded to a whole integer before computing the modulus.
If one of the operands is NULL, the arithmetic expression evaluates to NULL.

Example

```
MAIN
  DISPLAY 256 MOD 16
  DISPLAY 26 MOD 2
  DISPLAY 27 MOD 2
END MAIN
```

Related concepts

**Numeric expressions** on page 334
This section covers numeric expression evaluation rules.

**Character string operators**
Character string operators allow you to work with and manipulate character strings.

**Table 138: Character string operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII() on page 353</td>
<td>The ASCII() operator produces an ASCII character.</td>
</tr>
<tr>
<td>COLUMN on page 354</td>
<td>The COLUMN operator generates blanks.</td>
</tr>
<tr>
<td>Concatenate ([][]) on page 355</td>
<td>The</td>
</tr>
<tr>
<td>Append (,) on page 356</td>
<td>The , (comma) appends an expression to a string.</td>
</tr>
<tr>
<td>Substring ([s,e]) on page 357</td>
<td>The [ ] (square brackets) extract a substring.</td>
</tr>
<tr>
<td>USING on page 357</td>
<td>The USING operator converts date and numeric values to a string based on a formatting mask.</td>
</tr>
<tr>
<td>CLIPPED on page 358</td>
<td>The CLIPPED operator removes trailing blank spaces (ASCII 32) of a string expression.</td>
</tr>
<tr>
<td>ORD() on page 359</td>
<td>The ORD() operator returns the code point of a character in the current locale.</td>
</tr>
<tr>
<td>SPACES on page 359</td>
<td>The SPACES operator returns a character string with blanks.</td>
</tr>
<tr>
<td>LSTR() on page 360</td>
<td>The LSTR() operator returns a localized string.</td>
</tr>
<tr>
<td>SFMT() on page 360</td>
<td>The SFMT() operator replaces place holders in a string with values.</td>
</tr>
</tbody>
</table>

**ASCII()**
The ASCII() operator produces an ASCII character.

**Syntax**

```
ASCII ( int-exp )
```

1. *int-exp* is an integer expression. The range of possible values depends on the current application locale / character set:
   - For single byte encodings (like ISO8859-1), the argument must be in the range of 0 to 255.
   - For UTF-8, using char length semantics, the argument must be any valid 16bit code point (in the range 0-65535).
   - For any other locale setting (any multibyte character set, or UTF-8 using byte length semantics), the argument must be in the range 0 to 127.
Usage

The `ASCII()` operator returns the character corresponding to the ASCII code passed as a parameter, in the current encoding of the application locale.

The `ASCII()` function can be also used to produce special characters such as escape (`ASCII(27)`), newline (`ASCII(10)`), horizontal tab (`ASCII(9)`).

When the argument is zero, `ASCII()` has a different behavior, depending on the context:

- `ASCII(0)` only displays the NULL character within the `PRINT` statement.
- If you specify `ASCII(0)` in other contexts, it returns a blank space.

Example

```plaintext
MAIN
  DISPLAY ASCII(65), ASCII(66), ASCII(7)
END MAIN
```

Related concepts

- **Defining the application locale** on page 519
  This section describes the settings defining the application locale, changing the behavior of the compilers and runtime system.

- **ORD()** on page 359
  The `ORD()` operator returns the code point of a character in the current locale.

**COLUMN**

The `COLUMN` operator generates blanks.

Syntax

```
COLUMN position
```

1. `position` is the column position (starts at 1).

Usage

The `COLUMN` operator is typically used in report routines to align data in `PRINT` statements and move the character position forward within the current line. This operator makes sense when used in an expression with the comma append operator: Spaces will be generated depending on the number of characters that have been used in the expression, before the `COLUMN` operator.

The `COLUMN` operator can be used outside report routines, in order to align data to be displayed with a proportional font, typically in a TUI context. For example, the next lines will always `DISPLAY` the content of the `lastname` variable starting from column 30 of the terminal, no matters the number of characters contained in the `firstname` variable. The example defines `VARCHAR` variables, since `CHAR` variables are blank-padded, we would need to use the `CLIPPED` operator:

```plaintext
DEFINE firstname, lastname VARCHAR(50)
DISPLAY firstname, COLUMN(30), lastname
```

The `position` operand must be a non-negative integer that specifies a character position offset (from the left margin) no greater than the line width (that is, no greater than the difference (right margin - left margin). This designation moves the character position to a left-offset, where 1 is the first position after the left margin. If current position is greater than the operand, the `COLUMN` specification is ignored.

Example

```plaintext
PAGE HEADER
  PRINT "Number", COLUMN 12,"Name", COLUMN 35,"Location"
```
ON EVERY ROW
    PRINT customer_num, COLUMN 12, fname, COLUMN 35, city

Related concepts
CLIPPED on page 358
The CLIPPED operator removes trailing blank spaces (ASCII 32) of a string expression.

Concatenate (||)
The || operator makes a string concatenation.

Syntax

```
expr || expr
```

1. expr can be a character, numeric or date/time expression.

Usage
The || operator is the concatenation operator that produces a string expression from the expression elements on both sides of the operator.

If any of the members of a concatenation expression is NULL, the result string will be NULL.

The || operator is typically used to concatenate strings to build a value passed as parameters to a function.

Note: The || concatenation operator has different formatting rules as the comma concatenation operator. The comma will do fixed-size formatting for numeric and date/time values, while || will trim the values.

The || operator has a lower precedence than arithmetic operators; For example, a || b + c is equivalent to (a || (b+c)).

However, the || operator has a higher precedence than LIKE, MATCHES and USING operators; For example, v USING "&&" || "&" is equivalent to v USING ("&&" || "&") which is the same as v USING "&&&".

Example
Building strings to be passed as parameters to functions:

```
MAIN
    CALL print_message( 5, "Current date: " || TODAY )
    DISPLAY "Length: ", length( "ab" || "cdef" )
END MAIN

FUNCTION print_message(l TINYINT, s STRING)
    DISPLAY "Level: ", l, " ", s
END FUNCTION
```

Output:

```
Level: 5 Current date: 12/22/2018
Length: 6
```

Order of precedence test:

```
MAIN
    DISPLAY 78 || 1 + 8             -- 789
    DISPLAY 78 || 3 * 3             -- 789
    DISPLAY 789 USING "&&" || "&"   -- 789
    DISPLAY (34 USING "&&") || "&" -- 34&
    DISPLAY "abc" MATCHES "ab" || "c" -- 1 (TRUE)
    DISPLAY ("ab" MATCHES "ab") || "c" -- 1c
    DISPLAY "a" MATCHES "a" || "&&"  -- 0 (FALSE)
```
FORMAT 23 USING "a" MATCHES "a" || "&&"  -- * (format = "0")
DISPLAY 23 USING ("a" MATCHES "a") || "&&"  -- 123
END MAIN

Formatting differences between || and , (comma):

MAIN
DISPLAY "Date: ", TODAY, " day num: ", DAY(TODAY), " Pi=", 3.1415
DISPLAY "Date: 
TODAY
day num: 
DAY(TODAY)
Pi=3.1415
END MAIN

Output:

Date: 12/22/2018 day num: 22 Pi=3.1415

Related concepts

Append (,) on page 356
The , (comma) appends an expression to a string.

Syntax

expr, expr

1. expr can be a character, numeric or date/time expression.

Usage

The comma operator formats and concatenates expressions together.

This operator can only be used in some instructions such as LET, PRINT, MESSAGE, ERROR and DISPLAY instructions.

Use the comma concatenation operator when data needs to be formatted for printing and display. To concatenate elements in a compressed manner, use the || concatenation operator.

Example

MAIN
DEFINE s STRING
LET s = "A", -999; DISPLAY s
LET s = 123, 456; DISPLAY s
DISPLAY "Today: ", TODAY, " and a number: ", 12345.67
END MAIN

Output:

A -999
   123 456
Today: 12/22/2018 and a number: 12345.67

Related concepts

Formatting data on page 318
Explains data to string conversion options of the language.

**Substring ([s,e])**
The [ ] (square brackets) extract a substring.

**Syntax**

```
char-variable [ start \_ end \_ ]
```

1. `char-variable` must be a character data type variable.
2. `start` defines the position of the first character of the substring to be extracted.
3. `end` defines the position of the last character of the substring to be extracted.
4. If `end` is not specified, only one character is extracted.

**Usage**
The [ ] (square brackets) notation following a `CHAR(size)` on page 293 or `VARCHAR(size)` on page 309 variable extracts a substring from that character variable.

The `start` and `end` arguments can be expressed in bytes or characters, depending on the length semantics used in your programs.

**Important:** Substring expressions in SQL statements are evaluated by the database server. This may have a different behavior than the substring operator of the language.

**Example**

```
MAIN
  DEFINE s CHAR(10)
  LET s = "abcdef"
  DISPLAY s[3,4]
END MAIN
```

**Related concepts**

- **Defining the application locale** on page 519
  This section describes the settings defining the application locale, changing the behavior of the compilers and runtime system.

- **USING**
  The USING operator converts date and numeric values to a string based on a formatting mask.

**Syntax**

```
expr USING format
```

1. `expr` is a language expression.
2. `format` is a string expression that defines the formatting mask to be used.

**Usage**

The USING operator applies a formatting string to the left operand.

The left operand must be a valid date, integer or decimal number.

The format string can be any valid string expression using formatting characters as described in Formatting numeric values on page 318 and Formatting DATE values on page 321.

**Note:** `DATETIME` and `INTERVAL` expressions cannot be formatted with the USING operator. Use the `base.Datetime` and `base.Interval` methods instead. For more details, see Formatting DATETIME values on page 322 and Formatting INTERVAL values on page 324.
The **USING** operator has a low order of precedence: if you use operators with a higher precedence, the resulting string might not be what you are expecting.

For example, the `||` concatenation operator is evaluated before **USING**. As a result:

```plaintext
LET x = a || b USING "format"
```

will first concatenate `a` and `b`, then apply the **USING** format.

To solve this issue, use parentheses around the **USING** expression:

```plaintext
LET x = a || (b USING "format")
```

**Example**

```plaintext
MAIN
  DEFINE d DECIMAL(12,2)
  LET d = -12345678.91
  DISPLAY d USING "$-##,###,##&.&&"
  DISPLAY TODAY USING "yyyy-mm-dd"
END MAIN
```

**Related concepts**

- [Date, numeric and monetary formats](#) on page 532
  This section describes how Genero BDL handles date, time, numeric and monetary formats.
- [Using the Ming Guo date format](#) on page 532
  Genero BDL can be configured to use the The Ming Guo calendar.
- [Concatenate (||)](#) on page 355
  The `||` operator makes a string concatenation.

**CLIPPED**

The **CLIPPED** operator removes trailing blank spaces (ASCII 32) of a string expression.

**Syntax**

```plaintext
expr CLIPPED
```

1. `expr` is a language expression.

**Usage**

This operator removes all trailing blank space (ASCII 32) of a string expression.

The **CLIPPED** operator is typically used to remove the trailing blanks of a **CHAR** value, which would be printed otherwise.

**Example**

```plaintext
MAIN
  DISPLAY "Some text   " CLIPPED
END MAIN
```

**Related concepts**

- [String expressions](#) on page 334
This section covers string expression evaluation rules.

**ORD()**
The ORD() operator returns the code point of a character in the current locale.

**Syntax**

```plaintext
ORD( source STRING )
```

1. `source` is a string expression.

**Usage**
The value returned by ORD() is the code point in the current locale of the character passed as argument.

Only the first character of the argument is evaluated.

When using UTF-8 with character length semantics, the ORD() operator returns the UNICODE code point of the character.

ORD() returns NULL if the argument passed is not valid.

**Related concepts**
- Defining the application locale on page 519
  This section describes the settings defining the application locale, changing the behavior of the compilers and runtime system.
- fgl_keyval() on page 2289
  Returns the key code of a logical or physical key.
- ASCII() on page 353
  The ASCII() operator produces an ASCII character.

**SPACES**
The SPACES operator returns a character string with blanks.

**Syntax**

```plaintext
int-expr SPACES
```

1. `int-expr` is an integer expression.
2. SPACE (without S) is an alias for this operator.

**Usage**
The SPACE operator is typically used in reports to print spaces to align data in the report output.

**Example**

```plaintext
MAIN
  DISPLAY 20 SPACES || "xxx"
END MAIN
```

**Related concepts**
- Reports on page 2026
- Integer expressions on page 333
This section covers integer expression evaluation rules.

**LSTR()**
The `LSTR()` operator returns a localized string.

**Syntax**

```plaintext
LSTR(str-expr)
```

1. `str-expr` is a string expression.

**Usage**

The `LSTR()` operator returns a localized string corresponding to the identifier passed as parameter.

Normally localized strings are automatically replaced when using the `%"ident"` notation in the source code. When the localized string identifier is not known at compile time, use the `LSTR()` function.

**Example**

```plaintext
MAIN
  DISPLAY LSTR ("str"||123) -- loads string 'str123'
END MAIN
```

**Related concepts**

Localized strings on page 538

Localized strings provide a means of writing applications in which the text of strings can be customized on site.

**SFMT()**
The `SFMT()` operator replaces place holders in a string with values.

**Syntax**

```plaintext
SFMT( str-expr , param [, param [ ... ] ]
```

1. `str-expr` is a string expression.
2. `param` is any valid expression used to replace parameter place holders (%n).
3. At least one parameter is required.

**Usage**

The `SFMT()` operator can be used with parameters that will be automatically set in the string at the position defined by parameter placeholders. The parameters used with the `SFMT()` operator can be any valid expression. Numeric and date/time expressions are evaluated to strings depending on the current format settings (DBDATE, DBFORMAT).

A placeholder is a special marker in the string, that is defined by the percent character followed by the parameter number. For example, %4 represents the parameter #4. You are allowed to use the same parameter placeholder several times in the string. If you want to use the percent sign in the string, you must escape it with `%%`.

Predefined placeholders can be used to insert information about last runtime system error that occurred. Note that these are only available in the context of a runtime error trapped with a `WHENEVER ERROR GOTO / CALL` handler:

**Table 139: Predefined placeholders for runtime system error information**

<table>
<thead>
<tr>
<th>Predefined parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>% (ERRORFILE)</td>
<td>Name of the module where last runtime error occurred.</td>
</tr>
</tbody>
</table>
Predefined parameter | Description
---|---
%(ERRORLINE) | Line number in the module where last runtime error occurred.
%(ERRNO) | Last operating system error number.
%(STRERROR) | Last operating system error text.

Example

```
MAIN
  DEFINE n INTEGER
  LET n = 234
  DISPLAY SFMT("Order %1 has been %2.",n,"deleted")
END MAIN
```

In this example, %1 is replaced by the value of the variable n, while %2 is replaced by the string "deleted", resulting in: Order #234 has been deleted.

Related concepts

- **String expressions** on page 334
  This section covers string expression evaluation rules.

- **Associative syntax operators**
  Associative syntax operators allow you to group together objects.

Table 140: Associative syntax operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parentheses: () on page 361</td>
<td>Parentheses ( () ) force the evaluation of an expression before other operators.</td>
</tr>
<tr>
<td>Membership (object.member) on page 362</td>
<td>Separator for object members.</td>
</tr>
<tr>
<td>Variable parameter list: [ ] on page 362</td>
<td>Variable parameter list delimiters.</td>
</tr>
</tbody>
</table>

**Parentheses: ()**

Parentheses ( () ) force the evaluation of an expression before other operators.

**Syntax**

```
(expr [ ... ])
```

1. `expr` is a language expression.

**Usage**

Parentheses can be used to change the order in which expression elements are evaluated, to bypass the precedence of operators.

Parentheses can also be used to ease the readability of the code in a complex expression.

**Example**

```
MAIN
  DEFINE n INTEGER
  LET n = ( ( 3 + 2 ) * 2 )
  IF n=10 AND ( n<=0 OR n>=20 ) THEN
    DISPLAY "OK"
  END IF
```
Related concepts

Order of precedence on page 337
The order of precedence defines in which order the elements of an expression are evaluated.

Expressions on page 330
Shows the possible expressions supported in the language.

Membership (object.member)
Separator for object members.

Syntax

```
setname.element
```

Usage

The period expression element specifies that its right-hand operand is a member of the set whose name is its left-hand operand.

This notation is used to reference RECORD members, object and class methods, as well as module elements.

Example

```
IMPORT FGL customer_module
...
MAIN
  DEFINE rec RECORD
      n INTEGER,
      c CHAR(10)
  END RECORD
DEFINE form ui.Form
LET rec.n = 12345
LET rec.c = "abcdef"
...
CALL form.setElementHidden("page1")
...
CALL customer_module.check(345)
...
END MAIN
```

Related concepts

Built-in packages on page 2378
These topics cover the built-in classes provided by the Genero Business Development Language.

Variable parameter list: [ ]
Variable parameter list delimiters.

Syntax

```
[ variable [, ...] ]
```

Usage

The square brace notation in function parameters defines a variable list of arguments for a built-in function or a built-in class method.
The elements of a variable parameter list are program variables which are passed by reference. As result, the called function can modify the content of the passed variables, to return values in output parameters.

It is not possible to define user functions with variable parameter lists.

For real usage examples, see the read and write methods of the `base.Channel` class.

**Example**

```plaintext
MAIN
    DEFINE id INTEGER, name STRING,
        count INTEGER, stat INTEGER
    LET id = 12345
    LET name = "Forman"
    -- Warning: This is a fake call, the function does not exist!
    -- Here, id and name are passed as input values, while count
    -- and stat are used as output parameters...
    CALL built_in_function( [id, name], [count, stat] )
END MAIN
```

**Related concepts**

- [Read and write record data](#) on page 2425
- [SQL related operators](#)

**SQL related operators**

SQL related operators allow you to retrieve the SQL state and the SQL error message.

**Table 141: SQL related operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLSTATE</td>
<td>The <code>SQLSTATE</code> variable returns the code corresponding to the last SQL error.</td>
</tr>
<tr>
<td>SQLERRMESSAGE</td>
<td>The <code>SQLERRMESSAGE</code> variable holds the error message corresponding to the last SQL error.</td>
</tr>
</tbody>
</table>

**SQLSTATE**

The `SQLSTATE` variable returns the code corresponding to the last SQL error.

**Syntax**

```plaintext
SQLSTATE
```

**Usage**

The `SQLSTATE` predefined variable returns the ANSI/ISO SQLSTATE code when an SQL error occurred.

The `SQLSTATE` error code is a standard ANSI specification, but not all database engines support this feature. Check the database server documentation for more details.

The variable is `NULL` if the last SQL statement was successful.

**Example**

```plaintext
MAIN
    DATABASE stores
    WHENEVER ERROR CONTINUE
    SELECT foo FROM bar
    DISPLAY SQLSTATE
END MAIN
```
**Related concepts**

**SQL execution diagnostics** on page 591

If an SQL statement execution fails, error description can be found in the `SQLCA.SQLCODE`, `SQLSTATE`, `STATUS` and `SQLERRMESSAGE` predefined registers.

**SQLERRMESSAGE**

The `SQLERRMESSAGE` variable holds the error message corresponding to the last SQL error.

**Syntax**

```
SQLERRMESSAGE
```

**Usage**

The `SQLERRMESSAGE` predefined variable returns the error message if an SQL error occurred.

The variable is `NULL` if the last SQL statement was successful.

**Example**

```
MAIN
    DATABASE stores
    WHENEVER ERROR CONTINUE
    SELECT foo FROM bar
    DISPLAY SQLERRMESSAGE
END MAIN
```

**Related concepts**

**SQL execution diagnostics** on page 591

If an SQL statement execution fails, error description can be found in the `SQLCA.SQLCODE`, `SQLSTATE`, `STATUS` and `SQLERRMESSAGE` predefined registers.

**Data type operators**

Data type operators allow you cast a data type or create an instance of a data type.

**Table 142: Data type operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CAST</code> on page 364</td>
<td>The <code>CAST</code> operator converts a Java object to the user-defined type or Java class specified.</td>
</tr>
<tr>
<td><code>INSTANCEOF</code> on page 365</td>
<td>The <code>INSTANCEOF</code> checks the class of an object.</td>
</tr>
</tbody>
</table>

**CAST**

The `CAST` operator converts a Java object to the user-defined type or Java class specified.

**Syntax**

```
CAST( variable AS type )
```

1. `variable` is a variable referencing a Java object.
2. `type` is a user-defined type or a Java class.

**Usage**

The `CAST()` operator is required when you want to assign an object reference to variable defined with a user-defined type or Java class which requires narrowing reference conversion.
In the next code example, when assigning a `java.lang.StringBuffer` reference to a `java.lang.Object` variable, widening reference conversion occurs and no `CAST()` operator is needed, but when assigning an `java.lang.Object` reference to a `java.lang.StringBuffer` variable, you must cast the object reference to a `java.lang.StringBuffer`:

```java
IMPORT JAVA java.lang.Object
IMPORT JAVA java.lang.StringBuffer
MAIN
  DEFINE sb1, sb2 java.lang.StringBuffer
  DEFINE o java.lang.Object
  LET sb1 = StringBuffer.create()
  LET o = sb1 -- Widening Reference Conversion does not need CAST()
  LET sb2 = CAST( o AS java.lang.StringBuffer ) -- Narrowing
     -- Reference Conversion needs CAST()
END MAIN

In order to cast an `fgl.FglRecord` object to a regular `RECORD`, you need to specify a user-defined type (`TYPE` definition):

```java
IMPORT JAVA com.fourjs.fgl.lang.FglRecord
TYPE mytype RECORD f1, f2 INTEGER END RECORD
MAIN
  DEFINE r mytype
  DEFINE jr FglRecord
  LET jr = r
  LET r = CAST(jr AS mytype)
     -- This is denied:
     --   CAST(jr AS RECORD f1, f2 INTEGER END RECORD)
END MAIN
```

**Related concepts**
The `INSTANCEOF` operator on page 2218
The Java interface on page 2192

The *Java interface* allows you to import Java classes and instantiate Java objects in your programs.

**INSTANCEOF**
The `INSTANCEOF` checks the class of an object.

**Syntax**

```
variable INSTANCEOF class
```

1. `variable` is a variable referencing a Java object.
2. `class` is a Java class.

**Usage**
The `INSTANCEOF` operator evaluates to `TRUE` if the object reference is of the specified class.

**Example**

```java
IMPORT JAVA java.lang.Object
IMPORT JAVA java.lang.StringBuffer
IMPORT JAVA java.lang.Number
MAIN
  DEFINE o java.lang.Object
  DEFINE sb java.lang.StringBuffer
  LET sb = StringBuffer.create()
  LET o = sb
  DISPLAY sb INSTANCEOF java.lang.StringBuffer  -- shows 1
```
Related concepts
The CAST operator on page 2218
The Java interface on page 2192
The Java interface allows you to import Java classes and instantiate Java objects in your programs.

Assignment operators
An assignment operator allows you to assign a variable with an expression.

Table 143: Assignment operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment (:=)</td>
<td>The := operator assigns a variable with an expression and returns the result.</td>
</tr>
</tbody>
</table>

Assignment (:=)
The := operator assigns a variable with an expression and returns the result.

Syntax

```
variable := expr
```

Usage
The := assignment operator puts a value in the left-hand variable and the resulting value can again be used in an expression. In other programming languages, the := operator is known as the "walrus" operator.

**Note:** Do not confuse the := assignment operator with the LET instruction.
The := assignment operator has the lowest precedence, it can be used at many places and can simplify coding.

Example
In the next example, the := operator is used to increment the array index before usage:

```
MAIN
  DEFINE arr DYNAMIC ARRAY OF STRING,
     idx INTEGER
  LET idx = 0
  LET arr[idx:=idx+1] = "One"
  LET arr[idx:=idx+1] = "Two"
  LET arr[idx:=idx+1] = "Three"
END MAIN
```

Related concepts
Variables on page 390
Explains how to define program variables.

**Date and time operators**

Date and time operators allow you to work with date and time values.

**Table 144: Date and time operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURRENT on page 367</td>
<td>The CURRENT operator returns the current system date and time.</td>
</tr>
<tr>
<td>EXTEND() on page 368</td>
<td>The EXTEND() operator adjusts a date time value depending on the qualifier.</td>
</tr>
<tr>
<td>DATE() on page 368</td>
<td>The DATE() operator converts an expression to a DATE value.</td>
</tr>
<tr>
<td>TIME() on page 369</td>
<td>The TIME() operator returns a time part of the date time expression.</td>
</tr>
<tr>
<td>TODAY on page 369</td>
<td>The TODAY operator returns the current calendar date.</td>
</tr>
<tr>
<td>YEAR() on page 370</td>
<td>The YEAR() operator extracts the year of a date time expression.</td>
</tr>
<tr>
<td>MONTH() on page 370</td>
<td>The MONTH() operator extracts the month of a date time expression.</td>
</tr>
<tr>
<td>DAY() on page 371</td>
<td>The DAY() operator extracts the day of the month of a date time expression.</td>
</tr>
<tr>
<td>WEEKDAY() on page 371</td>
<td>The WEEKDAY() operator extracts the day of the week of a date time expression.</td>
</tr>
<tr>
<td>MDY() on page 372</td>
<td>The MDY() operator creates a DATE from month, day and year units.</td>
</tr>
<tr>
<td>UNITS on page 373</td>
<td>The UNITS operator converts an integer to an interval.</td>
</tr>
</tbody>
</table>

**CURRENT**

The CURRENT operator returns the current system date and time.

**Syntax**

```
CURRENT qual1 TO qual2 (scale)
```

1. `qual1`, `qual2` and `scale` define the date time qualifier.

**Usage**

The CURRENT operator returns the system date/time in the current local timezone.

This operator can be used to assign the current system date and time to a DATETIME variable.

Use optional datetime qualifiers to specify the precision of the returned value. The possible qualifiers are the same as in a DATETIME data type definition.

If the datetime qualifiers are not specified after the CURRENT keyword, the precision defaults to YEAR TO FRACTION(3) precision.

**Example**

```
MAIN
```
Related concepts

**EXTEND()** on page 368

The `EXTEND()` operator adjusts a date time value depending on the qualifier.

**Syntax**

```
EXTEND ( dt-expr, qual1 TO qual2 \(\text{scale}\) \)
```

1. `dt-expr` is a date / time expression.
2. `qual1`, `qual2` and `scale` define the date time qualifier.

**Usage**

The `EXTEND()` operator is used to convert a date time expression to a **DATETIME** value with a different precision. The default qualifier is **YEAR TO DAY**. The possible qualifiers are the same as in a **DATETIME** data type definition.

The expressions passed as first parameter must be a valid datetime value. If it is a character string, it must consist of valid and unambiguous time-unit values and separators, but with these restrictions:

- It cannot be a character string in date format, such as "12/12/99".
- It cannot be an ambiguous numeric datetime value, such as "05:06" or "05".
- It cannot be a time expression that returns an **INTERVAL** value.

**Example**

```plaintext
MAIN
  DISPLAY EXTEND ( TODAY, YEAR TO FRACTION(4) )
END MAIN
```

Related concepts

**CURRENT** on page 367

The `CURRENT` operator returns the current system date and time.

**DATE()**

The `DATE()` operator converts an expression to a **DATE** value.

**Syntax**

```
DATE \(\text{expr}\)\)
```

1. `expr` is the expression to be converted to a date.

**Usage**

`DATE()` converts a character string, an integer or datetime expression to a **DATE** value.

When `expr` is a character string expression, it must properly formatted by following the datetime format settings like **DBDATE**.
If $expr$ is an integer expression, it is used as the number of days since December 31, 1899.

If you supply no operand, it returns a character representation of the current date in the format "weekday month day year".

**Example**

```main
    main
        display date ( 34000 )
        display date ( "12/04/1978" )
        display date ( current )
        display date
    end main
```

**Related concepts**

- [Expressions](#) on page 330
  Shows the possible expressions supported in the language.

**TIME()**

The `TIME()` operator returns a time part of the date time expression.

**Syntax**

```time
    time [datetime-expr]
```

1. `datetime-expr` is a date time expression.

**Usage**

`TIME()` converts the time-of-day portion of its date time operand to a character string.

This operator converts a date time expression to a character string representing the time-of-day part of its operand.

The format of the returned string is always "hh:mm:ss".

If you supply no operand, it returns a character representation of the current time. You can use the `CURRENT` operator to get a date time result of the current system time.

**Example**

```main
    main
        display time ( current )
    end main
```

**Related concepts**

- [Datetime expressions](#) on page 335
  This section covers date-time expression evaluation rules.

**TODAY**

The `TODAY` operator returns the current calendar date.

**Syntax**

```today
    today
```

**Usage**

`TODAY` returns the current system date as a `DATE` value, in the current local timezone.

This operator can be used to assign the current system date to a `DATE` variable.
The `TODAY` operator is the `DATE` equivalent for the `CURRENT` operator used for `DATETIME`.

**Example**

```
MAIN
  DISPLAY TODAY
END MAIN
```

**Related concepts**

- [Date expressions](page 335)
  
  This section covers date expression evaluation rules.

**YEAR()**

The `YEAR()` operator extracts the year of a date time expression.

**Syntax**

```
YEAR ( expr )
```

1. `expr` is a date / time expression.

**Usage**

Returns an integer corresponding to the year portion of its operand.

**Example**

```
MAIN
  DISPLAY YEAR ( TODAY )
  DISPLAY YEAR ( CURRENT )
END MAIN
```

**Related concepts**

- [MONTH()](page 370)
  
  The `MONTH()` operator extracts the month of a date time expression.

- [DAY()](page 371)
  
  The `DAY()` operator extracts the day of the month of a date time expression.

- [WEEKDAY()](page 371)
  
  The `WEEKDAY()` operator extracts the day of the week of a date time expression.

- [MDY()](page 372)
  
  The `MDY()` operator creates a `DATE` from month, day and year units.

**MONTH()**

The `MONTH()` operator extracts the month of a date time expression.

**Syntax**

```
MONTH ( expr )
```

1. `expr` is a date / time expression.

**Usage**

Returns a positive whole number between 1 and 12 corresponding to the month of its operand.
Example

```plaintext
MAIN
  DISPLAY MONTH ( TODAY )
  DISPLAY MONTH ( CURRENT )
END MAIN
```

Related concepts

**YEAR()** on page 370
The **YEAR()** operator extracts the year of a date time expression.

**DAY()** on page 371
The **DAY()** operator extracts the day of the month of a date time expression.

**WEEKDAY()** on page 371
The **WEEKDAY()** operator extracts the day of the week of a date time expression.

**MDY()** on page 372
The **MDY()** operator creates a **DATE** from month, day and year units.

**DAY()**
The **DAY()** operator extracts the day of the month of a date time expression.

Syntax

```plaintext
DAY ( expr )
```

1. **expr** is a date / time expression.

Usage

Returns a positive whole number between 1 and 31 corresponding to the day of the month of its operand.

Example

```plaintext
MAIN
  DISPLAY DAY ( TODAY )
  DISPLAY DAY ( CURRENT )
END MAIN
```

Related concepts

**YEAR()** on page 370
The **YEAR()** operator extracts the year of a date time expression.

**MONTH()** on page 370
The **MONTH()** operator extracts the month of a date time expression.

**WEEKDAY()** on page 371
The **WEEKDAY()** operator extracts the day of the week of a date time expression.

**MDY()** on page 372
The **MDY()** operator creates a **DATE** from month, day and year units.

**WEEKDAY()**
The **WEEKDAY()** operator extracts the day of the week of a date time expression.

Syntax

```plaintext
WEEKDAY ( expr )
```

1. **expr** is a date / time expression.
Usage
Returns a positive whole number between 0 and 6 corresponding to the day of the week implied by its operand.
The integer 0 (Zero) represents Sunday.

Example

```
MAIN
  DISPLAY WEEKDAY( TODAY )
  DISPLAY WEEKDAY( CURRENT )
END MAIN
```

Related concepts
- **YEAR()** on page 370
  The `YEAR()` operator extracts the year of a date time expression.
- **MONTH()** on page 370
  The `MONTH()` operator extracts the month of a date time expression.
- **DAY()** on page 371
  The `DAY()` operator extracts the day of the month of a date time expression.
- **MDY()** on page 372
  The `MDY()` operator creates a `DATE` from month, day and year units.

MDY()
The `MDY()` operator creates a `DATE` from month, day and year units.

Syntax

```
MDY ( expr1, expr2, expr3 )
```

1. `expr1` is an integer representing the month (from 1 to 12).
2. `expr2` is an integer representing the day (from 1 to 28, 29, 30 or 31 depending on the month).
3. `expr3` is an integer representing the year (four digits).

Usage
The `MDY()` operator builds a date value with 3 integers representing the month, day and year. The result is a `DATE` value.
This function is sensitive to the `C1` modifier of the `DBDATE` environment variable, defining a Ming Guo date format.

Example

```
MAIN
  DISPLAY MDY ( 12, 3+2, 1998 )
END MAIN
```

Related concepts
- **YEAR()** on page 370
  The `YEAR()` operator extracts the year of a date time expression.
- **MONTH()** on page 370
  The `MONTH()` operator extracts the month of a date time expression.
- **DAY()** on page 371
  The `DAY()` operator extracts the day of the month of a date time expression.
- **WEEKDAY()** on page 371
The `WEEKDAY()` operator extracts the day of the week of a date time expression.

Using the Ming Guo date format on page 532
Genero BDL can be configured to use the The Ming Guo calendar.

**UNITS**
The `UNITS` operator converts an integer to an interval.

**Syntax**

```plaintext
expr UNITS qual\[scale]\]
```

where `qual` can be one of:

- YEAR
- MONTH
- DAY
- HOUR
- MINUTE
- SECOND
- FRACTION(1–6)

1. `expr` is an integer expression.

**Usage**

The `UNITS` operator converts an integer expression to an `INTERVAL` value expressed in a single unit of time that you specify after the `UNITS` keyword.

For the qualifiers YEAR, MONTH, DAY, HOUR and SECOND, if the left-hand expression evaluates to a decimal number, any fractional part is discarded before the `UNITS` operator is applied. However, when using `UNITS` FRACTION, the expression can be a decimal number where the integer part is interpreted as a number of seconds, and the decimal part as the fraction of a second:

```plaintext
MAIN
    DEFINE iv INTERVAL SECOND(9) TO FRACTION(5)
    LET iv = 76242.77999 UNITS FRACTION
    DISPLAY iv -- Displays " 76242.77999"
END MAIN
```

The `UNITS` operator can be used to compare `INTERVAL` values. For example, to check if an `INTERVAL` SECOND(9) TO SECOND is negative:

```plaintext
FUNCTION is_negative( iv INTERVAL SECOND(9) TO SECOND )
    RETURN (iv < 0 UNITS SECOND )
END FUNCTION
```

`UNITS` has a higher precedence than any arithmetic or boolean operator. As a result, a left-hand arithmetic expression that uses a `UNITS` operator must be enclosed in parentheses. For example, `10 + 20 UNITS MINUTES` will be evaluated as `10 + (20 UNITS MINUTES)` and give a conversion error. It must be written `(10 + 20) UNITS MINUTES` to get the expected result.

Because the difference between two `DATE` values is an integer count of days rather than an `INTERVAL` data type, you might want to use the `UNITS` operator to convert such differences explicitly to `INTERVAL` values:

```plaintext
MAIN
    DEFINE d DATE
    LET d = TODAY + 200
    DISPLAY (d - TODAY) UNITS DAY
END MAIN
```
Related concepts

**DATETIME qual1 TO qual2 on page 296**
The DATETIME data type stores date and time data with time units from the year to fractions of a second.

**EXTEND() on page 368**
The EXTEND() operator adjusts a date time value depending on the qualifier.

**Dialog handling operators**
Dialog handling operators allow you to handle variables in a DIALOG statement.

**Table 145: Dialog handling operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET_FLDBUF() on page 374</td>
<td>The GET_FLDBUF() operator returns as character strings the current values of the specified fields.</td>
</tr>
<tr>
<td>INFIELD() on page 375</td>
<td>The INFIELD() operator checks for the current screen field.</td>
</tr>
<tr>
<td>FIELD_TOUCHED() on page 375</td>
<td>The FIELD_TOUCHED() operator checks if fields were modified during the dialog execution.</td>
</tr>
</tbody>
</table>

**GET_FLDBUF()**
The GET_FLDBUF() operator returns as character strings the current values of the specified fields.

**Syntax**

```plaintext
GET_FLDBUF ( [group. ]field [, ...] )
```

1. *group* can be a table name, a screen record, a screen array or FORMONLY.
2. *field* is the name of the screen field.

**Usage**
The GET_FLDBUF() operator is used to get the value of a screen field before the input buffer is copied into the associated variable.

**Note:** Use of this operator is recommended only in dialogs allowing field input (INPUT, INPUT ARRAY, CONSTRUCT). The behavior is undefined when used in DISPLAY ARRAY.

The GET_FLDBUF() operator takes the field names as identifiers, not as string expressions:

```plaintext
LET v = GET_FLDBUF( customer.custname )
```

If multiple fields are specified between parentheses, use the RETURNING clause:

```plaintext
CALL GET_FLDBUF( customer.* ) RETURNING rec_customer.*
```

When used in a INPUT ARRAY instruction, the runtime system assumes that you are referring to the current row.

The values returned by this operator are context dependent; it must be used carefully. If possible, use the variable associated to the input field instead.

When using the UNBUFFERED mode, program variables are automatically assigned, and the GET_FLDBUF operator is not required in most cases.

**Example**

```plaintext
INPUT BY NAME ...
...```
ON KEY (CONTROL-Z)
    LET v = GET_FLDBUF( customer.custname )
    IF check_synonyms(v) THEN
        ...

Related concepts
The buffered and unbuffered modes on page 1720
The buffered and unbuffered mode control the synchronization of program variables and form fields.

Editable record list (INPUT ARRAY) on page 1531
The INPUT ARRAY instruction provides always-editable record list handling in an application form.

ui.Dialog.getFieldBuffer on page 2532
Returns the input buffer of the specified field.

INFIELD()
The INFIELD() operator checks for the current screen field.

Syntax

INFIELD ( _group_.field )

1. group can be a table name, a screen record, a screen array or FORMONLY.
2. field is the name of the field in the form.

Usage

INFIELD checks for the current field in a CONSTRUCT, INPUT or INPUT ARRAY dialog.

When used in an INPUT ARRAY instruction, the runtime system assumes that you are referring to the current row.

For a generic coding equivalent, use the DIALOG.getCurrentItem() method.

Example

INPUT ...
    IF INFIELD ( customer.custname ) THEN
        MESSAGE "The current field is customer's name."
    ...

Related concepts
Which form item has the focus? on page 1730
Identify what element of the current form has the focus.

Query by example (CONSTRUCT) on page 1565
The CONSTRUCT instruction implements database query criteria input in an application form.

Record input (INPUT) on page 1482
The INPUT instruction provides single record input control in an application form.

Editable record list (INPUT ARRAY) on page 1531
The INPUT ARRAY instruction provides always-editable record list handling in an application form.

FIELD_TOUCHED()
The FIELD_TOUCHED() operator checks if fields were modified during the dialog execution.

Syntax

FIELD_TOUCHED ( _group_.field.*
               _group_.*
               * )
1. `group` can be a table name, a screen record, a screen array or FORMONLY.
2. `field` is the name of the field in the form.

**Usage**

FIELD_TOUCHED returns TRUE if the value of a screen field (or multiple fields) has changed since the beginning of the interactive instruction.

The operator accepts a list of explicit field names, and supports the `\{group.\}*` notation in order to check multiple fields in a single evaluation. When passing a simple asterisk (*) to the operator, the runtime system will check all fields used by the current dialog.

When used in an INPUT ARRAY instruction, the runtime system assumes that you are referring to the current row.

The FIELD_TOUCHED operator can only be used inside an INPUT, INPUT ARRAY and CONSTRUCT interaction block.

For more details about the FIELD_TOUCHED operator usage and the understand the "touched flag" concept, refer to the definition of the DIALOG instruction.

Do not confuse the FIELD_TOUCHED operator with FGL_BUFFERTOUCHED built-in function; which checks a different field modification flag, that is reset when entering the field. The global touched flag controlled by FIELD_TOUCHED is reset when the dialog starts or when DIALOG.setFieldTouched() is used.

**Example**

```plaintext
INPUT ...
...
AFTER FIELD custname
   IF FIELD_TOUCHED( customer.custname ) THEN
      MESSAGE "Customer name was changed."
   END IF
...
AFTER INPUT
   IF FIELD_TOUCHED( customer.* ) THEN
      MESSAGE "Customer record was changed."
   END IF
...
```

**Related concepts**

- Input field modification flag on page 1723
  Each input field controlled by a dialog instruction has a modification flag.
- ui.Dialog.getFieldTouched on page 2532
  Returns the modification flag for a field.
- ui.Dialog.setFieldTouched on page 2550
Sets the modification flag of the specified field.

## Flow control

Definition of language elements and instructions that control the flow of a program.

### CALL

The CALL instruction invokes a specified function or method.

### Syntax

```plaintext

( [ parameter-name : parameter-value ] )

RETURNING [ variable ]
```

1. `built-in-function` is a built-in function.
2. `built-in-class` is a built-in class.
3. `class-method` is a method invoked by class name.
4. `module` is an imported module.
5. `variable-of-built-in-class` is a variable defined from a built-in class.
6. `object-method` is a method invoked by object reference.
7. `extension-class` is a class from an extension package.
8. `variable-of-extension-class` is a variable defined from an extension-class.
9. `cext-module` is an imported C extension module.
10. `cext-function` is a C function defined in a cext-module.
11. `java-class` is a imported Java class.
12. `java-class-method` is a method of a java-class invoked by class name.
13. `variable-of-java-class` is a variable defined from a java-class.
14. `java-object-method` is a method of a java-class invoked by object reference.
15. `user-function` is a function defined in one of the modules of the program.
16. `function-reference` is a variable referencing a function.
17. `variable-of-user-type` is a variable defined from a user-type that is associated to a user-defined method.
18. `user-method` is a method associated to a user type.
19. `parameter-name` is the name of a function parameter, as specified in the function definition. The parameter name is optional.
20. `parameter-value` can be any valid expression, including object references of built-in classes or Java classes.
21. `variable` is a variable receiving a value returned by the function.
Usage

The **CALL** instruction invokes the function or class/object method specified and passes the program flow control to that function/method. After the called function is executed, the flow control goes back to the caller, the runtime system executing the next statement that appears after the **CALL** instruction.

Function arguments can be any expression supported by the language. Use a double-pipe operator `||` to pass the concatenation of character string expressions as a parameter.

```plaintext
CALL my_function( TODAY, 20*0.5435, 'abc'||'def'||var1 )
```

The **RETURNING** clause assigns values returned by the function to variables in the calling routine. The **RETURNING** clause is only needed when the function returns parameters.

```plaintext
MAIN
  DEFINE var1 CHAR(15)
  DEFINE var2 CHAR(15)
  CALL foo() RETURNING var1, var2
  DISPLAY var1, var2
END MAIN

FUNCTION foo()
  DEFINE r1 CHAR(15)
  DEFINE r2 CHAR(15)
  LET r1 = "return value 1"
  LET r2 = "return value 2"
  RETURN r1, r2
END FUNCTION
```

If the function returns a unique parameter, the function can be used in an expression and can be directly assigned to a variable with **LET var = function(...)** statement.

```plaintext
MAIN
  DEFINE var1 CHAR(10)
  DEFINE var2 CHAR(2)
  LET var1 = foo()
  DISPLAY "var1 = " || var1
  CALL foo() RETURNING var2
  DISPLAY "var2 = " || var2
END MAIN

FUNCTION foo()
  RETURN "Hello"
END FUNCTION
```

The value of a receiving variable may be different from the value returned by the function, following the data conversion rules.

```plaintext
MAIN
  DEFINE s STRING
  LET s = div(10,2)
END MAIN

FUNCTION div(x,y)
  DEFINE x,y INTEGER
  RETURN x / y
END FUNCTION
```
If the `IMPORT FGL` instruction is used to import a module, function can be prefixed with the name of the module followed by a dot (i.e. `module.function`). The module prefix is required to fully-qualify the function in case of conflicts (i.e. when functions with the same name are defined in several modules).

```plaintext
-- main.4gl
IMPORT FGL module1
IMPORT FGL module2
MAIN
   CALL module1.show("aaa")
   CALL module2.show("aaa")
END MAIN
```

```plaintext
-- module1.4gl
FUNCTION show(s)
   DEFINE s STRING
   DISPLAY s
END FUNCTION
```

```plaintext
-- module2.4gl
FUNCTION show(s)
   DEFINE s STRING
   DISPLAY s
END FUNCTION
```

The symbol following the `CALL` keyword can be a program variable that references a function. This allows you to invoke functions dynamically, when the actual function to be called is only known at runtime:

```plaintext
TYPE t_func_ref FUNCTION (p1 INT, p2 INT) RETURNS INT
DEFINE fr t_func_ref,
   r INT
LET fr = FUNCTION add -- Function with the same signature as t_func_ref
CALL fr(100,200) RETURNING r
LET fr = FUNCTION sub -- Function with the same signature as t_func_ref
CALL fr(100,200) RETURNING r
```

To improve code readability, function parameters can be qualified with the name specified in the function definition:

```plaintext
CALL cleanup( mode: "full", verbose: TRUE )
```

For more details, see Calling functions on page 442.

**Related concepts**

- **Expressions** on page 330
  Shows the possible expressions supported in the language.

- **Type conversions** on page 311
  Explains primitive data type conversion rules of the language.

- **Importing modules** on page 480
Use the IMPORT ... instruction to import BDL, C or Java external modules in the current module.

RETURN

The RETURN instruction gives the control of execution back to the caller, optionally returning values on the stack.

Syntax

\[
\text{RETURN} \ [ \text{value} \ [, \ldots] \ ]
\]

1. value can be any valid expression, an object reference or complex type reference such as a dynamic array reference.

Usage

The RETURN instruction transfers the control back from a function with optional return values. Record members can be returned with the .* or THRU notation. Each member is returned as an independent variable.

Consider using the fully typed function definition syntax, with a RETURNS clause in the function header, to get better compilation control of your code. When using the RETURNS clause, the compiler will check that the function body contains RETURN instructions that match the number of return values as specified in the function definition.

A function may have several RETURN points (not recommended in structured programming) but they must all return the same number of values.

The number of returned values must correspond to the number of variables listed in the RETURNING clause of the CALL statement invoking this function.

A function cannot return a static array, but can return the reference of a dynamic array or dictionary.

Example

```plaintext
MAIN
  DEFINE fname, lname VARCHAR(30)
  LET lname = lastname(943)
  DISPLAY lname
  DEFINE fname, lname VARCHAR(30)
  LET lname = lastname(943)
  DISPLAY lname
  CALL fullname(235) RETURNING fname, lname
  DISPLAY fname, lname
END MAIN

FUNCTION lastname(id INTEGER) RETURNS STRING
  CASE id
    WHEN 943
      RETURN "McTiger"
    OTHERWISE
      RETURN NULL
  END CASE
END FUNCTION

FUNCTION fullname(id INTEGER) RETURNS(STRING, STRING)
  CASE id
    WHEN 235
      RETURN "Lee", "Park"
    OTHERWISE
      RETURN NULL, NULL
  END CASE
END FUNCTION
```
Related concepts

**Returning values** on page 446
A function can return values with the `RETURN` instruction.

**CALL** on page 377
The `CALL` instruction invokes a specified function or method.

**Expressions** on page 330
Shows the possible expressions supported in the language.

**Type conversions** on page 311
Explains primitive data type conversion rules of the language.

## CASE

The `CASE` instruction specifies statement blocks that must be executed conditionally.

### Syntax 1

```plaintext
CASE expression-1
    WHEN expression-2
        ↓ statement ↓ EXIT CASE ↓
    ↓ OTHERWISE ↓ EXIT CASE ↓
END CASE
```

1. `expression-1` is any expression supported by the language.
2. `expression-2` is an expression that is tested against `expression-1`.
3. `expression-1` and `expression-2` should have the same data type.
4. `statement` is any instruction supported by the language.

### Syntax 2

```plaintext
CASE
    WHEN boolean-expression
        ↓ statement ↓ EXIT CASE ↓
    ↓ OTHERWISE ↓ EXIT CASE ↓
END CASE
```

1. `boolean-expression` is any boolean expression supported by the language.
2. `statement` is any instruction supported by the language.

### Usage

In a `CASE` flow control block, the first matching `WHEN` block is executed. If there is no matching `WHEN` block, then the `OTHERWISE` block is executed. If there is no matching `WHEN` block and no `OTHERWISE` block, the program execution continues with the next statement following the `END CASE` keyword.

The `EXIT CASE` statement transfers the program control to the statement following the `END CASE` keyword. There is an implicit `EXIT CASE` statement at the end of each `WHEN` block and at the end of the `OTHERWISE` block. The `OTHERWISE` block must be the last block of the `CASE` instruction.

A null expression is considered as false: When doing a `CASE expr ... WHEN [NOT] NULL` using the syntax 1, it always evaluates to `FALSE`. Use syntax 2 as `CASE ... WHEN expr IS NULL` to test if an expression is null.
Make sure that $expression_2$ is not a boolean expression when using the first syntax. The compiler will not raise an error in this case, but you might get unexpected results at runtime.

If there is more than one $expression_2$ matching $expression_1$ (syntax 1), or if two boolean expressions (syntax 2) are true, only the first matching WHEN block will be executed.

**Example**

```plaintext
MAIN
  DEFINE v CHAR(10)
  LET v = "C1"
  -- CASE Syntax 1
  CASE v
    WHEN "C1"
      DISPLAY "Value is C1"
      WHEN "C2"
        DISPLAY "Value is C2"
        WHEN "C3"
          DISPLAY "Value is C3"
          OTHERWISE
            DISPLAY "Unexpected value"
    END CASE
  END CASE
  -- CASE Syntax 2
  CASE
    WHEN ( v="C1" OR v="C2" )
      DISPLAY "Value is either C1 or C2"
    WHEN ( v="C3" OR v="C4" )
      DISPLAY "Value is either C3 or C4"
    OTHERWISE
      DISPLAY "Unexpected value"
  END CASE
END MAIN
```

**Related concepts**

- **Expressions** on page 330
  Shows the possible expressions supported in the language.

- **Boolean expressions** on page 331
  This section covers boolean expression evaluation rules.

**CONTINUE block-name**

The `CONTINUE block-name` instruction resumes execution of a loop or dialog statement.

**Syntax**

```
CONTINUE
  FOR
  FOREACH
  WHILE
  MENU
  CONSTRUCT
  INPUT
  DIALOG
```

**Usage**

The `CONTINUE block-name` instruction transfers the program execution from a statement block to another location in the compound statement that is currently being executed.
CONTINUE block-name can only be used within the statement block specified by block-name. For example, CONTINUE FOR can only be used within a FOR ... END FOR statement block.

The CONTINUE FOR, CONTINUE FOREACH, or CONTINUE WHILE keywords cause the current FOR, FOREACH, or WHILE loop (respectively) to begin a new cycle immediately. If conditions do not permit a new cycle, however, the looping statement terminates.

The CONTINUE MENU, CONTINUE CONSTRUCT, CONTINUE INPUT and CONTINUE DIALOG statements cause the program to skip all subsequent statements in the current control block of a dialog. The screen cursor returns to the most recently occupied field in the current form, giving the user another chance to enter data in that field.

Note: CONTINUE INPUT is valid in INPUT and INPUT ARRAY statements.

Example

```
MAIN
  DEFINE i INTEGER
  LET i = 0
  WHILE i < 5
    LET i = i + 1
    DISPLAY "i=" || i
    CONTINUE WHILE
    DISPLAY "This will never be displayed!"
  END WHILE
END MAIN
```

Related concepts

Boolean expressions on page 331
This section covers boolean expression evaluation rules.

EXIT block-name

The EXIT block-name instruction transfers control out of the current program block.

Syntax

```
EXIT
  CASE
  FOR
  FOREACH
  WHILE
  MENU
  CONSTRUCT
  REPORT
  DISPLAY
  INPUT
  DIALOG ⊥
```

Usage

The EXIT block-name instruction transfers control out of a control structure (a block, a loop, a CASE statement, or an interface instruction).

The EXIT block-name instruction must be used inside the control structure specified by block-name. For example, EXIT FOR can only appear inside a FOR ... END FOR iteration block.

EXIT DISPLAY exits the DISPLAY ARRAY instruction and EXIT INPUT exits an INPUT or an INPUT ARRAY block. EXIT CONSTRUCT exits current CONSTRUCT block. EXIT DIALOG exits current DIALOG block.

To exit a function, use the RETURN instruction. To terminate a program, use the EXIT PROGRAM instruction.
Example

MAIN
   DEFINE i INTEGER
   LET i = 0
   WHILE TRUE
      DISPLAY "This is an infinite loop. How would you get out of here?"
      LET i = i + 1
      IF i = 100 THEN
         EXIT WHILE
      END IF
   END WHILE
   DISPLAY "Done."
END MAIN

Related concepts

RETURN on page 380
The RETURN instruction gives the control of execution back to the caller, optionally returning values on the stack.

EXIT PROGRAM on page 492
The EXIT PROGRAM instruction terminates the execution of the program.

FOR
The FOR instruction executes a statement block a specified number of times.

Syntax

FOR counter = start TO finish [ STEP value ]
   | statement
   | EXIT FOR
   | CONTINUE FOR
   [...]
END FOR

1. counter is the loop counter and must be an integer variable.
2. start is an integer expression used to set an initial counter value.
3. finish is any valid integer expression used to specify an upper limit for counter.
4. value is any valid integer expression whose value is added to counter after each iteration of the statement block.
5. When the STEP keyword is not given, counter increments by 1.
6. statement is any instruction supported by the language.
7. If value is less than 0, counter is decreased. In this case, start should be higher than finish.

Usage
The FOR instruction block executes the statements up to the END FOR keyword a specified number of times, or until EXIT FOR terminates the FOR statement. The CONTINUE FOR instruction skips the next statements and continues with the next iteration.

On the first iteration through the loop, the counter is set to the initial expression at the left of the TO keyword. For all further iterations, the value of the increment expression in the STEP clause specification (1 by default) is added to the counter in each pass through the block of statements. When the sign of the difference between the values of counter and the finish expression at the right of the TO keyword changes, the runtime system exits from the FOR loop.

The FOR loop terminates after the iteration for which the left- and right-hand expressions are equal. Execution resumes at the statement following the END FOR keywords. If either expression returns NULL, the loop cannot terminate, because the boolean expression "left = right" cannot become TRUE.

A value that equals 0 causes an unending loop unless there is an adequate EXIT FOR statement.
Note: Using NULL for start, finish or value is treated as 0. There is no way to catch this as an error. It is not recommended to use NULL as FOR loop values.

If statement modifies the value of counter, you might get unexpected results at runtime. In this case, it is recommended that you use a WHILE loop instead.

It is highly recommended that you ensure that statement does not modify the values of start, finish or value.

Example

```
MAIN
  DEFINE i, i_min, i_max INTEGER
  LET i_min = 1
  LET i_max = 10
  DISPLAY "Count from " || i_min || " to " || i_max
  DISPLAY "Counting forwards..."
  FOR i = i_min TO i_max
    DISPLAY i
  END FOR
  DISPLAY "... and backwards."
  FOR i = i_max TO i_min STEP -1
    DISPLAY i
  END FOR
END MAIN
```

Related concepts
Numeric expressions on page 334
This section covers numeric expression evaluation rules.

GOTO
The GOTO instruction transfers program control to a labeled line within the same program block.

Syntax

```
GOTO \[ label-id \]
```

1. label-id is the name of the LABEL statement to jump to.

Usage

A GOTO statement continues program execution in the line following the LABEL instruction using the label-id identifier specified in the GOTO instruction.

The LABEL jump point can be defined before or after the GOTO statement.

The LABEL and GOTO statements must use the label-id within a single MAIN, FUNCTION, or REPORT program block.

Note: The : colon after the GOTO keyword is optional.

GOTO statements can reduce the readability of your program source and result in infinite loops. Use FOR, WHILE and CASE statements instead.

The GOTO statement can be used in a WHENEVER statement to handle exceptions.

Example

```
MAIN
  DEFINE exit_code INTEGER
  DEFINE l_status INTEGER
  WHENEVER ANY ERROR GOTO _error
```
DISPLAY 1/0
GOTO _noerror

LABEL _error:
  LET l_status = STATUS
  DISPLAY "The error number ", l_status, " has occurred."
  DISPLAY "Description: ", err_get(l_status)
  LET exit_code = -1
  GOTO _exit

LABEL _noerror:
  LET exit_code = 0
  GOTO _exit

LABEL _exit:
  EXIT PROGRAM exit_code
END MAIN

Related concepts

WHENEVER instruction on page 503
Use the WHENEVER instruction to define how exceptions must be handled for the rest of the module.

IF

The IF instruction executes a group of statements conditionally.

Syntax

IF condition THEN
  statement
  [....]
  ELSE
  statement
  [....]
END IF

1.  condition is a boolean expression.
2.  statement is any instruction supported by the language.

Usage

By default, the runtime system evaluates all part of the condition. The semantics of boolean expressions can be controlled by the OPTIONS SHORT CIRCUIT compiler directive, to reduce expression evaluation when using AND / OR operators.

If condition is TRUE, the runtime system executes the block of statements following the THEN keyword, until it reaches either the ELSE keyword or the END IF keywords and resumes execution after the END IF keywords.

If condition is FALSE, the runtime system executes the block of statements between the ELSE keyword and the END IF keywords. If ELSE is absent, it resumes execution after the END IF keywords.

A NULL expression is considered as FALSE.

Important: Use the IS NULL or IS NOT NULL keywords to test if an expression is null or not null.

The following code example displays FALSE:

DEFINE var STRING
LET var = NULL
IF var THEN -- Bad practice!
DISPLAY "TRUE"
ELSE
  DISPLAY "FALSE"
END IF

The next example displays nothing, because both boolean expressions evaluate to null and then we do not enter in the IF block:

DEFINE var STRING
LET var = NULL
IF var THEN        -- Bad practice!
  DISPLAY "TRUE"
END IF
IF NOT var THEN    -- Bad practice!
  DISPLAY "FALSE"
END IF

In this example, using the "" string literal in the comparison expression results to NULL and is also a bad practice:

DEFINE var STRING
LET var = NULL
IF var == "" THEN  -- Bad practice!
  DISPLAY "You will not see this"
END IF

Consider using the IS NULL operator or, for strings that can contain blanks, check if the LENGTH() function returns zero:

DEFINE var STRING
LET var = NULL
IF var IS NULL THEN
  DISPLAY "NULL string"
END IF
LET var = "   
IF LENGTH(var)==0 THEN
  DISPLAY "String in empty, NULL or contains only blanks"
END IF

Example

MAIN
  DEFINE name CHAR(20)
  LET name = "John Smith"
  IF name MATCHES "John*" THEN
    DISPLAY "The name starts with [John]!"
  ELSE
    DISPLAY "The name is " || name || "."
  END IF
END MAIN

Related concepts

Boolean expressions on page 331
This section covers boolean expression evaluation rules.

**LABEL**

The **LABEL** instruction declares a jump point that can be reached by a **GOTO**.

**Syntax**

```plaintext
LABEL label-id:
```

1. `label-id` is a unique identifier in a **MAIN**, **REPORT**, or **FUNCTION** program block.
2. The `label-id` must be followed by a colon (`:`).

**Usage**

The **LABEL** instruction declares a statement label, making the next statement one to which a **GOTO** statement can transfer program control.

**Example**

```
MAIN
  DISPLAY "Line 2"
  GOTO line5
  DISPLAY "Line 4"
  LABEL line5:
  DISPLAY "Line 6"
END MAIN
```

**SLEEP**

The **SLEEP** instruction causes the program to pause for the specified number of seconds.

**Syntax**

```plaintext
SLEEP seconds
```

1. `seconds` must be an integer expression.

**Usage**

The **SLEEP** instruction is typically invoked to let the end user read a message displayed on a character terminal.

For example:

```
MAIN
  DISPLAY "Please wait 5 seconds..."
  SLEEP 5
  DISPLAY "Thank you."
END MAIN
```

With graphical applications, the **SLEEP** command is seldom used and should be avoided.

When `seconds` is lower than zero or is null, the program continues immediately with the next statement.

The **SLEEP** instruction can be interrupted by a process signal like **SIGINT**. If your program uses **DEFER INTERRUPT** to prevent program termination on **SIGINT**, the **SLEEP** instruction will return immediately when the **SIGINT** signal is caught by the process. The program will then continue after **SLEEP**, without waiting for the number of seconds specified.
To prevent interruption of the SLEEP command, put a SLEEP 1 (second) instruction in a WHILE loop and continue until the requested time has expired, as shown in the following example:

```
MAIN
    DEFER INTERRUPT
    CALL mysleep(3)
END MAIN

FUNCTION mysleep(secs SMALLINT)
    DEFINE te DATETIME YEAR TO FRACTION
    IF secs <= 0 OR secs IS NULL THEN
        RETURN
    END IF
    LET te = CURRENT + secs UNITS SECOND
    WHILE secs > 0
        SLEEP secs
        IF CURRENT >= te THEN
            EXIT WHILE
        END IF
        LET secs = ((te - CURRENT) / 1 UNITS SECOND) + 0.5
    END WHILE
END FUNCTION
```

Related concepts

**Integer expressions** on page 333
This section covers integer expression evaluation rules.

### WHILE

The **WHILE** statement executes a block of statements until the specified condition becomes false.

#### Syntax

```
WHILE condition
    ▼ statement
    ▼ EXIT WHILE
    ▼ CONTINUE WHILE ▼
    [....]
END WHILE
```

1. *condition* must be a boolean expression.
2. *statement* is any instruction supported by the language.

#### Usage

As long as the *condition* specified after a **WHILE** keyword is **TRUE**, all statements inside the **WHILE** ... **END WHILE** block are executed. After executing the last statement of the block, the runtime system again evaluates the condition, and if it is still **TRUE**, continues with the first statement in the block.

The loop stops when the condition becomes **FALSE** or when an **EXIT WHILE** is reached.

Use the **CONTINUE WHILE** instruction to skip the next statements and continue with the loop.

To avoid unending loops, make sure that the condition will become **FALSE** at some point, or that an **EXIT WHILE** statement will be executed.

#### Example

```
MAIN
    DEFINE cnt INTEGER
    LET cnt = 1
    WHILE cnt <= 100
```

DISPLAY "Iter: " || cnt
LET cnt = cnt + 1
IF int_flag THEN
   EXIT WHILE
END IF
END WHILE
END MAIN

Related concepts
Boolean expressions on page 331
This section covers boolean expression evaluation rules.

Variables
Expects how to define program variables.

Understanding variables
This is an introduction to variables.

A variable is a program element that can hold volatile data. The following list summarizes variables usage:

- Variables are declared in programs with the `DEFINE` instruction.
- In the definition of a variable, you can specify a value as `initializer`, for simple and structured types.
- After a definition without initializer, variables get default values specific to their type.
- The scope of a variable can be global, local to a module, or local to a function.
- When defined at the module level, a variable can be declare it as PRIVATE or PUBLIC.
- You can define structured variables with records, and with arrays.
- Default values (or NULL) can be assigned with the `INITIALIZE` instruction.
- Direct value assignment is done with the `LET` instruction.
- Database validation rules can be applied with the `VALIDATE` instruction.
- Variables can be used as SQL parameters or fetch buffers in SQL statements.
- Interactive instructions (dialogs) use program variables as model to hold the data.
- Variables can be defined with attributes by using the `ATTRIBUTES()` clause.

DEFINE
A variable contains volatile information of a specific data type.

Syntax

```
[PUBLIC|PRIVATE]
DEFINE identifier [,...] type-definition [ = initializer ]
[,...]
```

where `type-definition` is:

```
[ data-type
   LIKE [dbname:]tabname.colname
   ]
   [ attributes-list ]
```

or `type-definition` is a function type definition:

```
FUNCTION function-name (
   parameter-name data-type [ attributes-list ]
   [,...]
)
```
where `attributes-list` is:

```plaintext
ATTRIBUTES ( attribute \"value\", ..., )
```

where `initializer` is a variable initialization specification:

```plaintext
{ scalar-initializer
  | record-initializer
  | array-initializer
}
```

where `scalar-initializer` is:

```plaintext
{ integer-literal
  | decimal-literal
  | mdy-date-literal
  | datetime-literal
  | interval-literal
  | boolean-literal
  | NULL
}
```

where `record-initializer` is:

```plaintext
( identifier : initializer \[,\ldots\] )
```

and `array-initializer` is:

```plaintext
[ initializer \[,\ldots\] ]
```

1. `identifier` is the name of the variable or record member.
2. `data-type` can be a primitive data type, a record definition, an array definition, a user defined type, a built-in class, an imported package class, or a Java class.
3. `dbname` identifies a specific database schema file.
4. `tabname.colname` references a column defined in the database schema file.
5. `attribute` is an attribute to extend the variable definition with properties.
6. `value` is the value for the variable attribute, it is optional for boolean attributes.
7. `mdy-date-literal` is an `MDY(mm, dd, yyyy)` specification.

### Usage

A `variable` is a named location in memory that can store a single value, or an ordered set of values. Variables can be global to the program, module-specific, or local to a function.

Any program variable needs to be declared with a `DEFINE` statement before it is used.

A variable is typically defined with a `data type`.

The variable name must follow the convention of `identifiers`.

By default, module-specific variables are private; They cannot be used by an other module of the program. In order to improve code re-usability by data encapsulation, we recommend you keep module variables private, except if you want to share large data (like arrays) between modules. To make a module variable public, add the `PUBLIC` keyword before `DEFINE`. When a module variable is declared as public, it can be referenced in another module by using the `IMPORT FGL` instruction.
When defining variables with the **LIKE** clause, the data types are taken from the database schema file at compile time. Make sure that the schema file of the database schema during development corresponds to the database schema of the production database; otherwise the variables defined in the compiled version of your modules will not match the table structures of the production database. For more details, see [Database column types](#) on page 394.

To write well-structured programs, avoid global variables. If you need persistent data storage during a program's execution, use variables local to the module and give access to them with functions, or make the module variables **PUBLIC** to other modules. For more details, see [Declaration context](#) on page 392.

Variables can be defined with the **ATTRIBUTES()** clause, to specify meta-data information for the variable. For more details, see [Definition attributes](#) on page 398.

Variables can be defined to hold a **function reference**. Best practice is to declare a user-defined type with the function type, then define the variables using the **user type**.

By adding an equal sign followed by a value, you can save an additional **LET** instruction to initialize variables. Initializers are especially useful to set values of structured variables (records, arrays). For more details, see [Variable default values](#) on page 395.

**Related concepts**

- **Records** on page 408
  Records allow structured program variables definitions.
- **Arrays** on page 418
  Arrays (static or dynamic) allow you to handle an ordered collection of elements.

**Declaration context**

A variable can be declared in different contexts, which defines its visibility.

The **DEFINE** on page 390 statement declares the identifier of one or more variables, that will be visible to other program blocks depending on the declaration context of the variables. The scope of reference of a variable defines where it can be referenced in the program. Depending on the location of the variable definition, memory will be allocated when the program starts, or during the program execution.

The context of a variable declaration in the source module determines where a variable can be referenced by other language statements, and when storage is allocated for the variable in memory. The **DEFINE** statement can appear in three contexts:

1. **Within a FUNCTION, MAIN, or REPORT program block**, the **DEFINE** declares local variables, and causes memory to be allocated on the runtime stack when the function is called. These **DEFINE** declarations of local variables must precede any procedural statements within the same program block. The scope of reference of a local variable is restricted to the same program block. The variable is not visible elsewhere. Functions can be called recursively, and each recursive entry creates its own set of local variables. The variable is unique to that invocation of its program block. Each time the block is entered, a new copy of the variable is created.

2. **Outside any FUNCTION, REPORT, or MAIN program block**, the **DEFINE** statement declares module variables. Module variables have a persistent state during program execution. Memory for module variables is allocated when the module is loaded. Module variable declarations (**DEFINE**) must appear before any program blocks. By default, the scope of reference is the whole module (module variables are private to the module), but it can be extended to the whole program when the variable is declared with the **PUBLIC** qualifier.

3. **Inside a GLOBALS block**, the **DEFINE** statement declares global variables that are visible to the whole program. Global variables have a persistent state during program execution. Memory for global variables is allocated when the program starts. Multiple **GLOBALS** blocks can be defined for a given module. Use one module to declare all global variables and reference that module within other modules by using the **GLOBALS "filename.4gl"** statement as the first statement in the module, outside any program block.

A compile-time error occurs if you declare the same name for two variables that have the same scope. You can, however, declare the same name for variables that differ in their scope. For example, you can use the same identifier to reference different local variables in different program blocks.
You can also declare the same name for two or more variables whose scopes of reference are different but overlapping. Within their intersection, the compiler interprets the identifier as referencing the variable whose scope is smaller, and therefore the variable whose scope is a superset of the other is not visible.

If a local variable has the same identifier as a global variable, then the local variable takes precedence inside the program block in which it is declared. Elsewhere in the program, the identifier references the global variable.

A module variable can have the same name as a global variable that is declared in a different module. Within the module where the module variable is declared, the module variable takes precedence over the global variable. Statements in that module cannot reference the global variable.

A module variable cannot have the same name as a global variable that is declared in the same module.

If a local variable has the same identifier as a module variable, then the local identifier takes precedence inside the program block in which it is declared. Elsewhere in the same source-code module, the name references the module variable.

If a variable needs to be persistent during program execution, instead of using global variables, consider defining that variable in the module it belongs to, by specifying the PUBLIC or PRIVATE modifiers, depending on the scope you want to give to your variable, for other modules.

**Related concepts**
- Example 1: Local function variables on page 404
- Example 2: PRIVATE module variables on page 404
- Example 3: PUBLIC module variables on page 404
- Example 4: Global variables on page 404

**Structured variables**

Variables can be declared with a composite data type, based on simple data types.

To declare a structured variable, use a **RECORD definition**.

For example:

```plaintext
MAIN
  DEFINE myrec RECORD
    id INTEGER,
    name VARCHAR(100)
  END RECORD
  DEFINE myarr DYNAMIC ARRAY OF RECORD
    id INTEGER,
    name VARCHAR(100)
  END RECORD
  LET myarr[2].id = 52
END MAIN
```

Consider defining a **user type** to list the record members once and reuse the type in all variable definitions:

```plaintext
TYPE mytype RECORD
  id INTEGER,
  name VARCHAR(100)
END RECORD

MAIN
  DEFINE mv mytype
  CALL func1()
END MAIN

FUNCTION func1()
  DEFINE fv mytype
  ...
```

Database column types

Simple variables and record structures can be defined from database columns types.

Variables defined with the LIKE keyword get the same data type as the table column of a database schema.

Important: Database schema files are generated with the fgldbsch tool. Column data types are read from the schema file during compilation. Make sure that your schema files correspond exactly to the production database.

For example:

```
SCHEMA stores
DEFINE cust_name LIKE customer.cust_name
MAIN
   DEFINE rec_cust RECORD LIKE customer.*
   ...
END MAIN
```

A SCHEMA statement must define the database name identifying the database schema files to be used.

Alternatively, specify the database schema file followed by a colon, before the table name. This allows you to use several database schemas at the same time:

```
DEFINE rec_city RECORD LIKE base:city.*
DEFINE rec_country RECORD LIKE base:country.*
DEFINE rec_customer RECORD LIKE orders:customer.*
DEFINE rec_item RECORD LIKE stock:item.*
```

At runtime, a program typically connects to a single database source. Using multiple database schemas is a programming feature.

The database schema files must exist and must be located in one of the directories specified in the FGLDBPATH environment variable.

For CHAR/VARCHAR types, the size in the .sch file, the size in the .sch file is expressed in character units, and is then interpreted by the compiler as a number of bytes or characters, depending on the FGL_LENGTH_SEMANTICS environment variable as when using directly CHAR(N) or VARCHAR(N). For more details, see Extracting database schemas on page 525.

When using database views, the column cannot be based on an aggregate function like SUM().

If LIKE references a SERIAL column, the variable will be defined with the INTEGER data type. If LIKE references an INT8, SERIAL8 or BIGSERIAL column, the variable will be defined with the BIGINT data type.

The table qualifier must specify owner if table.column is not a unique column identifier within its database, or if the database is ANSI-compliant and any user of your application is not the owner of table.

Related concepts

Primitive Data types on page 289
Selecting the correct data type assists you in the input, storage, and display of your data.

User defined types

User defined types help to centralize the definition of complex structured data types.

A user defined type is created with the TYPE keyword. The next code example defines a type as a dynamic array of a record, with the structure of a database table as defined in the mydbschema.sch schema file:

```
SCHEMA mydbschema
...
TYPE t_custlist DYNAMIC ARRAY OF RECORD LIKE customer.*
```
Variables can then be defined with the "custlist" user defined type:

```
DEFINE cl t_custlist
```

The scope of a type can be global, local to a module or local to a function.

Variables can be defined with a type defined in the same scope, or in a higher level of scope.

A typical usage is to declare user-defined types as PUBLIC in a module, so they can be reused in other modules:

The `myutils.4gl` module defines the user type:

```
PUBLIC TYPE t_item_info RECORD
  id INTEGER,
  description VARCHAR(100),
  creation DATE,
  mandatory BOOLEAN,
  ...
END RECORD
```

In the module importing the utility module, define variables using the type name, with utility module as prefix:

```
IMPORT FGL myutils

DEFINE itemlist DYNAMIC ARRAY OF myutils.t_item_info
```

**Related concepts**

- [Types](#) on page 431
  - Types can be defined by the programmer to centralize the definition of complex/structured variables.

### Variable default values

Variables get a default value when defined.

By default, variables are initialized to a value depending on the variable data type.

For example, when defining an INTEGER variable, it gets zero as default value, while a DECIMAL variable is set to NULL:

```
MAIN
  DEFINE v1 INTEGER, v2 DECIMAL(10,2)
  DISPLAY v1 -- Shows "0"
  DISPLAY NVL(v2,"null") -- Shows "null"
END MAIN
```

The next table lists default initialization values for each **primitive data type**:

**Table 146: Data type specific default values for variables**

<table>
<thead>
<tr>
<th>Data type</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOLEAN</td>
<td>FALSE</td>
</tr>
<tr>
<td>CHAR</td>
<td>NULL</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>NULL</td>
</tr>
<tr>
<td>STRING</td>
<td>NULL</td>
</tr>
<tr>
<td>INTEGER</td>
<td>Zero</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>Zero</td>
</tr>
</tbody>
</table>
### Related concepts

**Variable initializers**

Variables can be initialized with a literal.

The `DEFINE` instruction can be used with an equal sign followed by a literal to initialize the variable:

```sql
DEFINE v INTEGER = -800
```

**Note:** A variable initializer can not use expressions such as a value returned by a function, it must always be a static literal.

The compiler produces the error `-6631`, if the type of the initializer and the type of the variable are incompatible:

```sql
MAIN
  DEFINE v INT = "abc"
  incompatible types, found: 'CHAR', required: 'INTEGER'.
  See error number -6631.
END MAIN
```

**Note:** Local function variables of the first function in a module will be initialized before any `WHENEVER` statement.

Date variables can be initialized with a `MDY()` literal:

```sql
DEFINE d DATE = MDY(12,24,2018)
```

Record variables can be initialized with a list of identifiers followed by a colon and the literal value, surrounded with parentheses:

```sql
DEFINE r1 RECORD
  f1 INTEGER,
  f2 STRING
END RECORD = (f1: 99, f2: "abc")
```

An invalid record initialization value can produce compilation errors such as `-8421`, `-8424`, or `-8425`.

Variables defined with types can also be initialized:

```sql
TYPE Type1 RECORD
  f1 INTEGER,
  f2 STRING
```
Record members can be omitted, or specified in a different order:

```plaintext
DEFINE r1 Type1 = (f2: "abc")           -- f1 omitted
DEFINE r2 Type1 = (f2: "abc", f1: 99)    -- f2 before f1
```

**Note:** When a record member is omitted in the initializer, it will get the default value according to its type, as described in Variable default values on page 395.

Variables defined from database schema columns can be initialized by specifying the initializer after the LIKE clause:

```plaintext
SCHEMA stores
DEFINE customer RECORD LIKE customer.* = (fname: 'Peter', lname: 'Mango')
```

Initializers for nested records must specify the sub-record member name followed by parentheses including the values for the members of the sub-record:

```plaintext
DEFINE r1 RECORD
    f1 INTEGER,
    f2 RECORD
        f21 STRING,
        f22 STRING
    END RECORD
END RECORD = ( f1: 99,
    f2: (f21:"abc", f22:"def") )
```

Arrays can be initialized by specifying a list of elements separated by a comma, surrounded by square brackets:

```plaintext
DEFINE arr DYNAMIC ARRAY OF INT = [ 1, 2, 3, 5, 8, 13 ]
```

An invalid array initialization value can produce compilation errors such as -8422.

Structured arrays can be initialized by combining the initializer syntax for arrays and records:

```plaintext
DEFINE pls DYNAMIC ARRAY OF RECORD
    id INTEGER,
    name VARCHAR(50)
END RECORD = [
    ( id: 501, name: "Baxter" ),
    ( id: 502, name: "Folkap" ),
    ( id: 503, name: "Kirtshof" )
]
```

See the DEFINE syntax topic for a complete description of the variable initializer syntax.

**Related concepts**

Literals on page 325
Describes the syntax of literals (constant values) to be used in sources.

**Definition attributes**

Variables can be defined with meta-data information.

**Basics**

To specify metadata information when defining a variable, use the `ATTRIBUTES` clause right after the type:

```
DEFINE myvar INTEGER ATTRIBUTES( ... )
```

**Note:** When defining attributes for complex types such as arrays, dictionaries or records, the `ATTRIBUTES` clause must be specified right after the complex type keywords:

```
DEFINE uuids DYNAMIC ARRAY ATTRIBUTES(json_name="uuid-list") OF STRING
DEFINE customer RECORD ATTRIBUTES(json_name = "customer-record")
... END RECORD
```

**JSON serialization attributes**

To define JSON serialization options, use variable definition attributes such as `json_null` and `json_name`:

```
DEFINE rec RECORD
  cust_id INTEGER ATTRIBUTES(json_null="null"),
  cust_name INTEGER ATTRIBUTES(json_name="Customer Name"),
  ...
  orderlist DYNAMIC ARRAY ATTRIBUTES(json_null="undefined") OF RECORD
    ...
    END RECORD,
  ...
END RECORD
```

List of supported JSON attributes for variable definitions:

- `json_null` (values can be "null" or "undefined")
- `json_name`

For more details about JSON serialization attributes see [BDL names and JSON element names](#) on page 577.

**XML serialization attributes**

Variable attributes are also used when defining variables for XML-based Web Services:

```
DEFINE data RECORD ATTRIBUTES(XMLName="Demo"),
  val1 INTEGER ATTRIBUTES(XMLName= "Value1"),
  val2 STRING ATTRIBUTES(XMLName= "Value2")
  attr INTEGER ATTRIBUTES(XMLAttribute,XMLName= "MyAttr")
END RECORD
```

For more details about XML attributes, see [Attributes to customize XML serialization](#) on page 3523.

**Related concepts**

[DEFINE](#) on page 390
A variable contains volatile information of a specific data type.

**INITIALIZE**

The INITIALIZE instruction initializes program variables with NULL or default values.

### Syntax

```
INITIALIZE target [, ...]  
  ↓  TO NULL  
  ↓  LIKE [table.* | table.column]  
```

1. `target` is the name of the variable to be initialized.
2. `table.column` can be any column reference defined in the database schema files.

### Usage

The INITIALIZE instruction assigns NULL or default values to variables.

The argument of the INITIALIZE instruction can be a simple variable, a record (with .* notation), a record member, a range of record members specified with the THRU keyword, an array or an array element.

The TO NULL clause initializes the variable to NULL.

When initializing a static array TO NULL, all elements will be initialized to null. When initializing a dynamic array TO NULL, all elements will be removed (i.e. the dynamic array is cleared).

The LIKE clause initializes the variable to the default value defined in the database schema validation file. This clause works only by specifying the `table.column` schema entry corresponding to the variable.

To initialize a complete RECORD, you can use the star to reference all members:

```
INITIALIZE record.* LIKE table.*
```

You cannot initialize variables defined with a complex data type (like TEXT or BYTE) to a non-NULL value.

### Example

```plaintext
SCHEMA stores  
MAIN  
  DEFINE cr RECORD LIKE customer.*  
  DEFINE a1 ARRAY[100] OF INTEGER  
  INITIALIZE cr.cust_name TO NULL  
  INITIALIZE cr.cust_name THRU cr.cust_address TO NULL  
  INITIALIZE cr.* LIKE customer.*  
  INITIALIZE a1 TO NULL  
  INITIALIZE a1[10] TO NULL  
END MAIN
```

### Related concepts

- [DEFINE](#) on page 390
- A variable contains volatile information of a specific data type.
- [Variable default values](#) on page 395
Variables get a default value when defined.

LOCATE (for TEXT/BYTE)

The LOCATE statement specifies where to store data of TEXT and BYTE variables.

Syntax 1: Locate in memory

LOCATE target IN MEMORY

Syntax 2: Locate in a specific file

LOCATE target IN FILE filename

Syntax 3: Locate in a temporary file

LOCATE target IN FILE

1. target is the name of a TEXT or BYTE variable to be located.
2. filename is a string expression defining the name of a file.

Usage

Before using TEXT and BYTE large objects, the data storage location must be specified with the LOCATE instruction. After defining the data storage, the variable can be used as input parameter or as a fetch buffer in SQL statements, as well as in interaction statements and reports.

The first syntax using the IN MEMORY clause specifies that the large object data must be located in memory.

The second syntax using the IN FILE filename clause specifies that the large object data must be located in a specific file.

The third syntax using the IN FILE clause specifies that the large object data must be located in a temporary file. The location of the temporary file can be defined with the DBTEMP environment variable. If DBTEMP is not defined, the default temporary directory dependents from the platform are used.

The FREE instruction can be used to free the resources allocated to the large object variable.

Note: When the TEXT or BYTE variable is already located, a new LOCATE will free the allocated resource: If the prior LOCATE was using the IN FILE clause, the temporary file is dropped, if the prior LOCATE was using IN MEMORY, the memory is freed.

Example

The following code example defines two TEXT variables. The first located in memory and the second located in a named file. The variables are then used in SQL statements:

```
MAIN
    DEFINE ctext1, ctext2 TEXT
    DATABASE stock
    LOCATE ctext1 IN MEMORY
    LOCATE ctext2 IN FILE "/tmp/data1.txt"
    CREATE TABLE lobtab ( key INTEGER, col1 TEXT, col2 TEXT )
    INSERT INTO lobtab VALUES ( 123, ctext1, ctext2 )
END MAIN
```

The next code example illustrates the storage semantics of BYTE and TEXT, by fetching large objects from the database into an array. Each member of the array needs to get an individual storage location, before the data is
actually fetched into the LOB handler of the array element. By using LOCATE IN FILE, a temporary file will be created for each large object:

```sql
TYPE t_arr DYNAMIC ARRAY OF RECORD
  id INTEGER,
  cmt TEXT
END RECORD

MAIN
  DEFINE arr t_arr,
  t TEXT
  DATABASE test1
  LOCATE t IN MEMORY
  CREATE TEMP TABLE tt1 ( id INTEGER, cmt TEXT )
  LET t = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
  INSERT INTO tt1 VALUES ( 1, t )
  LET t = "bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb";
  INSERT INTO tt1 VALUES ( 2, t )
  CALL fill_array(arr)
END MAIN

FUNCTION fill_array(arr)
  DEFINE arr t_arr,
  i INTEGER
  CALL arr.clear()
  DECLARE c1 CURSOR FOR SELECT * FROM tt1
  LET i=1
  LOCATE arr[i].cmt IN FILE
  FOREACH c1 INTO arr[i].*
      LOCATE arr[i:=i+1].cmt IN FILE
  END FOREACH
  CALL arr.deleteElement(i)
  FOR i=1 TO arr.getLength()
      DISPLAY arr[i].*
  END FOR
END FUNCTION
```

**Related concepts**

**FREE (for TEXT/BYTE)** on page 401

The `FREE` statement releases resources allocated to the specified variable.

**FREE (for TEXT/BYTE)**

The `FREE` statement releases resources allocated to the specified variable.

**Syntax**

```
FREE target
```

1. *target* is the name of a TEXT or BYTE variable to be freed.
Usage
When followed by a variable name, the FREE statement releases resources allocated to store the data of TEXT and BYTE variables.

If the TEXT/BYTE variable was located in memory, the runtime system releases the memory. If the variable was located in a file, the runtime system deletes the file.

For variables declared in a local scope of reference, the resources are automatically freed by the runtime system when returning from the function or MAIN block.

After freeing a TEXT or BYTE variable, it must be re-configured with a new LOCATE call.

Temporary files of large object are automatically deleted when the program ends.

Example

```
MAIN
  DEFINE ctext TEXT
  DATABASE stock
  LOCATE ctext IN FILE "/tmp/data1.txt"
  SELECT col1 INTO ctext FROM lobtab WHERE key=123
  FREE ctext
END MAIN
```

Related concepts
FREE (result set cursor) on page 723
Releases SQL cursor resources allocated by the DECLARE instruction.

LET
The LET statement assigns values to variables.

Syntax

```
LET target = expr [, ...]
```

1. target is the name of the variable to be assigned.
2. expr is any valid expression supported by the language.

Usage
The LET statement assigns a value to a variable, or a set of values to all members of a RECORD by using the . * notation.

The runtime system applies data type conversion rules if the data type of expr does not correspond to the data type of target.

When assigning a numeric or date/time value to a character string variable, the values are formatted for display (for example, the numeric data is right-aligned).

When specifying a comma-separated list of expressions for the right operand, the LET statement concatenates all expressions together. Unlike the || operator, if an expression in the comma-separated list evaluates to NULL, the concatenation result will not be null, except if all expressions to the right of the equal sign are null.

The target variable can be a record followed by dot- star (record.*), to reference all record members of the record. In this case, the right operand must also be a record using this notation, and all members will be assigned individually.

Variables defined with a complex data type (like TEXT or BYTE) can only be assigned to NULL.

Example

```
SCHEMA stores
MAIN
```
DEFINE c1, c2 RECORD LIKE customer.*
-- Single variable assignment
LET c1.customer_num = 123
-- Complete RECORD assignment
LET c1.* = c2.*
END MAIN

Related concepts
Variable default values on page 395
Variables get a default value when defined.

Expressions on page 330
Shows the possible expressions supported in the language.

Type conversions on page 311
Explains primitive data type conversion rules of the language.

VALIDATE
The VALIDATE instruction checks a variable value based on database schema validation rules.

Syntax

```
VALIDATE target [,..] LIKE
  ↓
  table.*
  ↓
  table.column
```

1. `target` is the name of the variable to be validated.
2. If `target` is a record, you can use the star notation to validate all members in the record.
3. `table.column` can be any column reference defined in the database schema.

Usage
The VALIDATE statement tests whether the value of the specified variable is within the range of values for a corresponding column in `.val database schema file` referenced by a SCHEMA clause. If the value does not match any value defined in the INCLUDE attribute of the corresponding column, the runtime system raises error -1321.

The argument of the VALIDATE instruction can be a simple variable, a record, or an array element. If the target is a record, you can use the dot-star (`.*`) notation to reference all record members in the validation, or specify a range of record members with the THRU clause.

Example

```
SCHEMA stores
MAIN
  DEFINE cname LIKE customer.cust_name
  LET cname = "aaa"
  VALIDATE cname LIKE customer.cust_name
END MAIN
```

Related concepts
Form-level validation rules on page 1727
Form-level validation rules can be defined for each field controlled by a dialog.

Examples

Variable definition usage examples.

Example 1: Local function variables

This example shows how to define local function variables.

```
FUNCTION myfunc()
    DEFINE i INTEGER
    FOR i=1 TO 10
        DISPLAY i
    END FOR
END FUNCTION
```

Example 2: PRIVATE module variables

This example shows how to define a private module variable.

```
PRIVATE DEFINE s VARCHAR(100)
FUNCTION myfunc()
    DEFINE i INTEGER
    FOR i=1 TO 10
        LET s = "item #" || i
    END FOR
END FUNCTION
```

Example 3: PUBLIC module variables

This example shows how to declare public and private module variables.

The module "mydebug.4gl":

```
PUBLIC DEFINE level INTEGER,
    logfile STRING
PRIVATE DEFINE count INTEGER
FUNCTION message(m)
    DEFINE m STRING
    IF level THEN
        -- Write message to debug_logfile
        DISPLAY m
    END IF
    LET count = count + 1
END FUNCTION
```

The main module:

```
IMPORT FGL mydebug
MAIN
    LET mydebug.level = 4
    LET mydebug.logfile = "myfile.log"
    CALL mydebug.message("Some debug info...")
END MAIN
```

Example 4: Global variables

This example shows how to define and use global variables.
The module "myglobs.4gl":

```4gl
GLOBALS
  DEFINE userid CHAR(20)
  DEFINE extime DATETIME YEAR TO SECOND
END GLOBALS
```

The module "mylib.4gl":

```4gl
GLOBALS "myglobs.4gl"
DEFINE s VARCHAR(100)
FUNCTION myfunc()
  DEFINE i INTEGER
  DISPLAY "User Id = " || userid
  FOR i=1 TO 10
    LET s = "item #" || i
  END FOR
END FUNCTION
```

The main module:

```4gl
GLOBALS "myglobs.4gl"
MAIN
  LET userid = fgl_getenv("LOGNAME")
  LET extime = CURRENT YEAR TO SECOND
  CALL myfunc()
END MAIN
```

**Constants**

The definition of constants allows to centralize common static values.

**Understanding constants**

This is an introduction to constant definition.

A constant defines a read-only value identified by a name. A constant is similar to a variable, except that its value cannot be modified by program code.

A constant is defined with an identifier, and optional data type and a literal value.

Constants are typically used to define common invariable values that will be used at several places in a program:

```4gl
CONSTANT PI DECIMAL(12,10) = 3.1415926,
   MAX_SIZE INT = 10000,
   ERRMSG = "PROGRAM ERROR: %1" -- type defaults to STRING
```

A good practice is to group constants that belong to the same domain in a given .4gl module. Defining these constants as PUBLIC, and import the module where the constants are needed.

**Related concepts**

Importing modules on page 480
Use the IMPORT ... instruction to import BDL, C or Java external modules in the current module.

**CONSTANT**

The CONSTANT instruction defines a program constant.

**Syntax**

\[
\text{[PRIVATE|PUBLIC]} \text{ CONSTANT } \text{constant-definition } [, \ldots]
\]

where \text{constant-definition} is:

\[
\text{identifier } [\text{ data-type }] = \text{literal}
\]

1. \text{identifier} is the name of the constant to be defined.
2. \text{data-type} can be any primitive data type, except complex types like \text{TEXT} or \text{BYTE}.
3. \text{literal} must be an integer, decimal, string, or date/time, interval literal, or an \text{MDY()} expression.
4. \text{literal} cannot be \text{NULL}.

**Usage:**

Define constants to name static values that do not change during the program execution.

Constants are similar to variables, except that its value cannot be modified.

Constants can be defined with global, module, or function scope.

By default, module constants are private; they cannot be used by an other module of the program. To make a module constant public, add the PUBLIC keyword before CONSTANT. When a module constant is declared as public, it can be referenced by another module by using the IMPORT instruction.

When declaring a constant, the data type specification can be omitted. The literal value automatically defines the data type:

\[
\begin{align*}
\text{CONSTANT c1 = "Drink"} & \quad \text{-- Declares a STRING constant} \\
\text{CONSTANT c2 = 4711} & \quad \text{-- Declares an INTEGER constant}
\end{align*}
\]

However, in some cases, you may need to specify the exact data type for a constant:

\[
\begin{align*}
\text{CONSTANT c1 SMALLINT = 12000} & \quad \text{-- Would be an INTEGER by default}
\end{align*}
\]

Constants can be used in variable, records, and array definitions:

\[
\begin{align*}
\text{CONSTANT n = 10} \\
\text{DEFINE a ARRAY[n] OF INTEGER}
\end{align*}
\]

Constants can be used at any place in the language where you normally use literals:

\[
\begin{align*}
\text{CONSTANT n = 10} \\
\text{FOR i=1 TO n} \\
\ldots
\end{align*}
\]

Constants can be passed as function parameters, and returned from functions.

Define public constants in a module to be imported by others:

\[
\begin{align*}
\text{PUBLIC CONSTANT pi = 3.14159265}
\end{align*}
\]
For date time constants, the value must be specified as an `MDY()` literal, `DATETIME literal` or `INTERVAL literal`:

```
CONSTANT my_date DATE = MDY(12,24,2011)
CONSTANT my_datetime DATETIME YEAR TO SECOND = DATETIME(2011-12-24 11:22:33) YEAR TO SECOND
CONSTANT my_interval INTERVAL HOUR(5) TO FRACTION(3) = INTERVAL(-54351:50:24.234) HOUR(5) TO FRACTION(3)
```

A constant cannot be used in the `ORDER BY` clause of a static `SELECT` statement, because the compiler considers identifiers after `ORDER BY` as part of the SQL statement (i.e. column names), not as constants:

```
CONSTANT pos = 3
-- Next line will produce an error at runtime
SELECT * FROM customers ORDER BY pos
```

Automatic data type conversion can take place in some cases:

```
CONSTANT c1 CHAR(10) = "123"
CONSTANT c2 CHAR(10) = "abc"
DEFINE i INTEGER
FOR i = 1 TO c1 -- Constant "123" is converted to 123 integer
...
FOR i = 1 TO c2 -- Constant "abc" is converted to zero!
...
```

Character constants defined with a string literal that is longer than the length of the data type are truncated:

```
CONSTANT s CHAR(3) = "abcdef"
DISPLAY s -- Displays "abc"
```

The compiler throws an error when an undefined symbol is used in a constant declaration:

```
CONSTANT s CHAR(c) = "abc"
-- Compiler error: c is not defined.
```

The compiler throws an error when a variable is used in a constant declaration:

```
DEFINE c INTEGER
CONSTANT s CHAR(c) = "abc"
-- Compiler error: c is a variable, not a constant.
```

The compiler throws an error when you try to assign a value to a constant:

```
CONSTANT c INTEGER = 123
LET c = 345
-- Runtime error: c is a constant.
```

The compiler throws an error when the symbol used is not defined as an integer constant:

```
CONSTANT c CHAR(10) = "123"
DEFINE s CHAR(c)
-- Compiler error: c is a not an integer constant.
```

You typically define common special characters with constants:

```
CONSTANT c_esc  = '\x1b'
CONSTANT c_tab  = '\t'
CONSTANT c_cr   = '\r'
CONSTANT c_lf   = '\n'
```
CONSTANT c_crlf = '
'

Related concepts
Importing modules on page 480
Use the IMPORT ... instruction to import BDL, C or Java external modules in the current module.

Examples
CONSTANT usage examples.

Example 1: Defining and using constants
This example shows how to define constants.

CONSTANT
c1 ="Drink", # Declares a STRING constant
c2 = 4711, # Declares an INTEGER constant
n = 10, # Declares an INTEGER constant
x SMALLINT=1 # Declares a SMALLINT constant
DEFINE a ARRAY[n] OF INTEGER
MAIN
CONSTANT c1 = "Hello"
DEFINE i INTEGER
FOR i=1 TO n
...
END FOR
DISPLAY c1 || c2 # Displays "Hello4711"
END MAIN

Records
Records allow structured program variables definitions.

Understanding records
This is an introduction to records.

A record defines a structured variable, where each member can be defined with a specific data type.

Records are used in interactive instructions like INPUT to control forms, and record are also used in INSERT and UPDATE SQL instructions, to update the database table.

Records can contain sub-record structures, and list types like arrays and dictionaries:

DEFINE reader RECORD
    id INTEGER,
    name VARCHAR(100),
    birth DATE,
    address RECORD
        num VARCHAR(20),
        street VARCHAR(200),
        city_id INTEGER,
        state_id VARCHAR(5)
    END RECORD,
    book_ids DYNAMIC ARRAY OF INTEGER
END RECORD
Records are typically used to store the values of a database row. Records can be based on the column types of a database table, which is defined in a database schema (SCHEMA):

```
SCHEMA stores
DEFINE cust RECORD customer.*
   -- cust is defined with the column of the customer table
```

The following list summarizes record usage:

- Record variables are defined with the RECORD syntax block, or with a TYPE defined as RECORD.
- Records can be defined with attributes.
- Records can be initialized in their definitions with record initializers.
- To access a record member, use the dot notation (`record-name.member-name`).
- Records can be copied to other records, if they are defined with the same type, by using the `LET rec1 = rec2` assignment operation.
- It is possible to compare all members of records of the same type with the `rec1.* == rec2.*` comparison expression.
- Records can be passed as function parameters by value or by reference (INOUT).

**RECORD**

The RECORD keyword defines a structured type or variable.

**Syntax 1 (explicit record definition)**

```
RECORD [ attributes-list ]
   member [ data-type ]
   \| LIKE [ dbname: ]tabname.colname
\| attributes-list \[
\] END RECORD
```

where attributes-list is:

```
ATTRIBUTES ( attribute = "value" \| \|, \..., )
```

1. `member` is an identifier for a record member / field.
2. `data-type` is a data type, a record definition, a user defined type, an array definition, a built-in class, an imported package class, or a Java class.
3. `dbname` identifies a specific database schema file.
4. `tabname.colname` references a column defined in the database schema file.
5. `attribute` is an attribute to extend the record or record member definition with properties.
6. `value` is the value for the record definition attribute, it is optional for boolean attributes.

**Syntax 2 (database column based record)**

```
```

where attributes-list is:

```
ATTRIBUTES ( attribute = "value" \| \|, \..., )
```

1. `dbname` identifies a specific database schema file.
2. `tabname.*` references the structure of a complete table defined in the database schema file.
3. `attribute` is an attribute to extend the record definition with properties.
4. *value* is the value for the record definition attribute, it is optional for boolean attributes.

**Usage**

A record defines an ordered set of variables called members. Each record member is defined with a specific type or in turn, structured type.

**Tip:** Consider defining a user type for records, to avoid to repeat the record definition for each variable.

Records whose members correspond in number, order, and data type compatibility to a database table can be useful for transferring data from the database to the screen, to reports, or to functions.

In the first form (Syntax 1), record members are defined explicitly:

```define rec record
  cust_id int,
  cust_name varchar(50),
  cust_address varchar(100),
  ...
end record
```

In the second form (Syntax 2), record members are created implicitly from the table definition found in the database schema file specified by the `SCHEMA` instruction:

```schema stock
... define rec record like customer.*
```

**Important:** When using the `LIKE` clause, the data types are taken from the database schema file during compilation. Make sure that the database schema file of the development database corresponds to the production database, otherwise the records defined in the compiled version of your programs will not match the table structures of the production database. Statements like `SELECT * INTO record.* FROM table` would fail.

**Record initializers**

Records can be initialized in the `DEFINE` instruction.

To initialize a record variable in its definition, use the equal sign followed by a record initializer. A record initializer is specified with parentheses, where record member values must be specified as a `key: value` pair, separated by a comma:

```define rec record
  cust_id int,
  cust_name varchar(50)
end record = ( cust_id: 101, cust_name = "Mike Torn" )
```

Below is a complex record example, using a sub-record and a dynamic array:

```define reader record
  id integer,
  name varchar(100),
  birth date,
  address record
    num varchar(20),
    street varchar(200),
    city_id integer,
    state_id varchar(5)
  end record,
  book_ids dynamic array of integer
end record = ( id : 123,
  name : "Scott Spinley",
```
birth : MDY(12, 24, 1998),
address : {
    num : "2A",
    street : "Sunset Bld",
    city_id : 9834,
    state_id : "CA"
},
book_ids : [ 234, 34, 458 ]

For more details, see Variable initializers on page 396.

**Record definition attributes**

Records can be defined with attributes, to complete the type description.

Records can be defined with the `ATTRIBUTES()` clause, to specify meta-data information for the record.

In the next code example, the `t_cust` record type gets attributes to define the corresponding field names for JSON serialization:

```plaintext
IMPORT util

TYPE t_cust RECORD ATTRIBUTES(json_name="customer")
    cust_id INTEGER ATTRIBUTES(json_name="id"),
    cust_name VARCHAR(50)
END RECORD

TYPE t_group RECORD
    cust1 t_cust,
    cust2 t_cust
END RECORD

MAIN
    DEFINE g t_group

    LET g.cust1.cust_id = 998
    LET g.cust1.cust_name = "Scott Finley"
    LET g.cust2.cust_id = 999
    LET g.cust2.cust_name = "Mike Fergusson"

    DISPLAY util.JSON.format(util.JSON.stringify(g))

END MAIN

Output:

```javascript
{
    "customer": {
        "id": 998,
        "cust_name": "Scott Finley"
    },
    "customer": {
        "id": 999,
        "cust_name": "Mike Fergusson"
    }
}
```

Record attributes are part of the attributes specification for variables.

**Note:** This feature is especially used when defining records for XML-based Web Services. For more details about XML attributes, see Attributes to customize XML serialization on page 3523.
Accessing record members

Record members are accessed by the dot notation.

The notation `record-name.member-name` refers to an individual member of a record:

```
DISPLAY rec.cust_name TO FORMONLY.f_name -- Single record member
```

The notation `record-name.*` refers to the entire list of record members.

```
DISPLAY BY NAME rec.* -- All record members
```

The notation `record-name.start-member THRU record-name.end-member` refers to a consecutive set of members. (THROUGH is a synonym for THRU):

```
DISPLAY rec.cust_id THRU rec.cust_address -- Members from cust_id to cust_address
```

For complex record structures, sub-records can be accessed by chaining the dot-member notation:

```
DEFINE reader RECORD
  id INTEGER,
  name VARCHAR(100)
  address RECORD
    num VARCHAR(20),
    street VARCHAR(200),
    city_id INTEGER
  END RECORD
END RECORD

MAIN
  LET reader.name = "Mike Finley"
  LET reader.address.num = "2A"
  LET reader.address.street = "Sunset Bld"
END MAIN
```

Related concepts

`LET` on page 402
The `LET` statement assigns values to variables.

Copying records

Records can be assigned to each other with the `=` operator.

It is possible to copy all members of a record to the members of another record, as long as the types match exactly, by using the assignment operator:

```
LET rec2 = rec1
```

**Important:** Record members of type `DYNAMIC ARRAY`, `DICTIONARY` and the LOB data of `TEXT/BYTE` members are not cloned when assigning records. A full copy of such members must be done by hand, otherwise both records will reference the same dynamic array, dictionary or LOB data.

When assigning records, both variables must have been defined with the same `RECORD` specification, or with the same `user-type`. If the types do not match exactly, the compiler will produce the error `4325`:

```
DEFINE rec1 RECORD
  id INTEGER,
  name VARCHAR(50)
END RECORD
```
DEFINE rec2 RECORD
  id INTEGER,
  name VARCHAR(100)  -- Note the difference with rec1
END RECORD

MAIN
  LET rec2 = rec1
  The source and destination records in this record assignment statement
  are not compatible in types and/or length.
  See error number -4325.
END MAIN

Note: The .* (dot star) notation LET rec2.* = rec1.* is supported for backward compatibility and is
equivalent to LET rec2 = rec1. When copying records, avoid the .* after the record names: The result at
runtime is the same, but your code will benefit from more compiler verifications.

If the record structure contains sub-records on several levels, the sub-members are copied recursively:

TYPE t_person RECORD
  ...
  address RECORD
  ...
  END RECORD
END RECORD
...
DEFINE r1, r2 t_person...
...
LET r1 = r2  -- address sub-record is copied as well...

If the record structure contains dynamic array or dictionary members, the runtime system will copy the reference
of such members. Similarly, if the record uses BYTE/TEXT members, the (locator) handler of the large object will
be copied, but the actual LOB data is not cloned. As result, after the copy, both records will point to the same array,
dictionary or TEXT/BYTE data.

The next code example implements a copyTo() method for the t_reader record type, which makes a proper
clone of the book_ids dynamic array:

IMPORT util

TYPE t_reader RECORD
  id INTEGER,
  name VARCHAR(100),
  book_ids DYNAMIC ARRAY OF INTEGER
END RECORD

MAIN
  DEFINE r1, r2 t_reader
  LET r1.id = 999
  LET r1.name = "Mike Rutberg"
  LET r1.book_ids[1] = 98458
  DISPLAY "1: r1 books: ", util.JSON.stringify(r1.book_ids)
  CALL r1.copyTo( r2 )
  DISPLAY "2: r2 books: ", util.JSON.stringify(r2.book_ids)
  DISPLAY "3: r1 books: ", util.JSON.stringify(r1.book_ids)
  DISPLAY "4: r2 books: ", util.JSON.stringify(r2.book_ids)
FUNCTION (r t_reader) copyTo(dst t_reader INOUT)
  DEFINE bl DYNAMIC ARRAY OF INTEGER
  LET dst = r -- makes a copy of the reference of book_ids
  CALL r.book_ids.copyWith(bl) -- clone the array
  LET dst.book_ids = bl -- copy the reference to the new array
  -- Don't do this:
  -- CALL r.book_ids.copyWith(dst.book_ids)
  -- dst.book_ids and r.book_ids reference the same array!
END FUNCTION

Output:

1: r1 books: [98458,98111]
2: r2 books: [98458,98111]
3: r1 books: [98458,98111,18234]
4: r2 books: [98458,98111]

Unlike dynamic array members, static array members are fully copied, as when assigning static arrays directly.

For more details about copying TEXT/BYTE variables, see the TEXT data type.

Related concepts

Equal to (==) on page 343
The == operator checks for equality of two expressions or for two record variables.

Different from (!=) on page 344
The != operator checks for non-equality of two expressions or for two record variables.

Comparing records

Records can be compared with the == comparison operator and the .* notation.

It is possible to compare records by using the .* notation and the == or != operators:

```
IF recl.* == rec2.* THEN
  ...
END IF
```

All members will be compared individually.

Important: If the record contains members of type ARRAY[n], DYNAMIC ARRAY, DICTIONARY or TEXT/BYTE, the comparison will always evaluate to FALSE, except when TEXT/BYTE members are NULL in both records. Consider testing each record member individually. Eventually, the size of TEXT/BYTE members can be compared with the LENGTH() function.

If the record contains several levels of sub-records, all sub-records will be processed recursively.

If two record members are NULL, the result of this member comparison is TRUE.

If two corresponding members do not contain the same value, or one of them is NULL, the records are considered as different.

Tip: Record comparison can be used to implement optimistic locking for database updates. For more details, read the SQL programming guide.

The next code example shows how to compare two records defined from the same type:

```
TYPE t_cust RECORD
  id INTEGER,
  name VARCHAR(50),
  address RECORD
```
num VARCHAR(5),
  street VARCHAR(100)
END RECORD
END RECORD

MAIN
DEFINE r1, r2 t_cust
  LET r1.id = 999
  LET r1.name = "Mike Torme"
  LET r1.address.num = "2A"
  LET r1.address.street = "Sunset bld"
  LET r2 = r1
  DISPLAY "1: ", IIF( r1.* == r2.*, "Equals", "Differs" )
  LET r2.name = "Mike Torm"
  DISPLAY "2: ", IIF( r1.* == r2.*, "Equals", "Differs" )
  LET r2.name = NULL
  DISPLAY "3: ", IIF( r1.* == r2.*, "Equals", "Differs" )
END MAIN

Output:
1: Equals
2: Differs
3: Differs

Related concepts
IF on page 386
The IF instruction executes a group of statements conditionally.

Records as function parameters
Records can be passed to function by value or by reference.

Records can be passed as function parameters, and can be returned from functions.

When passing records to functions with the record-name.* notation, the record is expanded, as if each individual member had been passed by value:

```plaintext
TYPE t_cust RECORD
  id INTEGER,
  name VARCHAR(50)
END RECORD

MAIN
  DEFINE r t_cust
  LET r.id = 999
  LET r.name = "Mike Torme"
  CALL func1(r.*) -- Local variable r is not changed
  DISPLAY "1: ", r.id, " ", r.name
  CALL func2() RETURNING r.* -- Local variable r is changed
  DISPLAY "2: ", r.id, " ", r.name
END MAIN

FUNCTION func1(x t_cust)
  LET x.name = "Scott Finley"
END FUNCTION

FUNCTION func2()
```
DEFINE x t_cust
LET x.id = 998
LET x.name = "Scott Finley"
RETURN x.*
END FUNCTION

Output:

1:         999 Mike Torme
2:         998 Scott Finley

Records can be passed to functions by reference, if the function is defined with the INOUT keyword:

TYPE t_cust RECORD
  id INTEGER,
  name VARCHAR(50)
END RECORD

MAIN
  DEFINE r t_cust
  LET r.id = 999
  LET r.name = "Mike Torme"
  CALL func1(r) -- Local variable r is changed
  DISPLAY "1: ", r.id, " ", r.name
END MAIN

FUNCTION func1(x t_cust INOUT)
  LET x.name = "Scott Finley"
END FUNCTION

Records can be passed by value to a method, when you specify only the name of the record:

TYPE t_class RECORD
  id INTEGER
END RECORD

TYPE t_cust RECORD
  id INTEGER,
  name VARCHAR(50)
END RECORD

MAIN
  DEFINE o t_class
  DEFINE r t_cust
  LET r.id = 999
  LET r.name = "Mike Torme"
  CALL o.meth1(r) -- Local variable r is not changed
  DISPLAY "1: ", r.id, " ", r.name
END MAIN

FUNCTION (this t_class) meth1(x t_cust)
  LET this.id = x.id
  LET x.name = "Scott Finley"
END FUNCTION

Output:

1:         999 Mike Torme

For more details, see Passing records as parameter on page 494.
**THRU/THROUGH**

The THRU keyword can be used to specify a set of members of a record.

**Syntax**

```
record.first-member ↓ THRU ↓ THROUGH ↓ record.last-member
```

1. `record` defines the record to be used.
2. `first-member` defines the member of the record starting the group of variables.
3. `last-member` defines the member of the record ending the group of variables.
4. `THROUGH` is a synonym for `THRU`.

**Usage**

The THRU keyword can be used in several instructions such as `DISPLAY`, `PRINT`, `INITIALIZE`, `VALIDATE`, `LOCATE`, to specify a list of record members by using a starting member and an ending member.

**Example**

```sql
SCHEMA stores
MAIN
    DEFINE cust LIKE customer.*
    INITIALIZE cust.cust_name THRU customer.cust_address TO NULL
END MAIN
```

**Related concepts**

`RECORD` on page 409

The `RECORD` keyword defines a structured type or variable.

**Examples**

`RECORD` usage examples.

**Example 1: Defining a record with explicit member types**

This example shows a simple record definition with built-in types.

```sql
MAIN
    DEFINE rec RECORD
        id INTEGER,
        name VARCHAR(100),
        birth DATE
    END RECORD
    LET rec.id = 50
    LET rec.name = 'Scott'
    LET rec.birth = TODAY
    DISPLAY rec.*
END MAIN
```

**Example 2: Defining a record with a database table structure**

This example shows how to define a record with members using the same types as database columns.

```sql
SCHEMA stores
DEFINE cust RECORD LIKE customer.*
MAIN
    DATABASE stores
    SELECT * INTO cust.* FROM customer WHERE customer_num=2
    DISPLAY cust.*
END MAIN
```
Example 3: Assigning an comparing records
This example shows how to use records with comparison and assignment operators.

```
TYPE t_cust RECORD
  cust_id INTEGER,
  cust_name VARCHAR(50)
END RECORD
MAIN
  DEFINE cust1, cust2 t_cust
  LET cust1.cust_id = 999
  LET cust1.cust_name = "Scott Spinley"
  LET cust2 = cust1 -- Legacy syntax: LET cust2.* = cust1.*
  IF cust1.* = cust2.* THEN
    DISPLAY "1: Records are equal."
  END IF
  LET cust1.cust_name = NULL
  IF cust1.* != cust2.* THEN
    DISPLAY "2: Records are different."
  END IF
END MAIN
```

Arrays
Arrays (static or dynamic) allow you to handle an ordered collection of elements.

Understanding arrays
This is an introduction to arrays.

Arrays can store a one-, two- or three-dimensional set of elements.

The language supports three kind of array types:

- Static arrays - introduced in early versions of the language.
- Dynamic arrays - to be used in new developments.
- Java arrays - to define a Java array, to interface with Java classes.

For static and dynamic arrays, elements can be of simple built-in types such as `INTEGER`, `VARCHAR(n)`, `RECORD` structured types, or `user-defined types`. A BDL array can also be defined with built-in classes, imported module classes (from the Web Services extensions for example), or Java classes.

The first element in a BDL array is at position 1, for static, dynamic and Java arrays.

Arrays can be used to define a list of records that will be controlled by dialog instructions such as `DISPLAY ARRAY` and `INPUT ARRAY`.

Related concepts
Dictionaries on page 426
A dictionary holds an unordered collection of elements accessed by a key.

Using Java arrays on page 2215
ARRAY

An array defines a vector variable with a list of elements.

Syntax 1: Static array definition

```
ARRAY [ size [, size ] ] attributes-list OF data-type
```

where attributes-list is:

```
ATTRIBUTES ( attribute = "value" [, ... ] )
```

1. `size` can be an integer literal or an integer constant. The upper limit is 65535.
2. `data-type` can be a data type, a record definition, a user defined type, a built-in class, an imported package class, or a Java class.
3. `attribute` is an attribute to extend the array definition with properties.
4. `value` is the value for the array definition attribute, it is optional for boolean attributes.

Syntax 2: Dynamic array definition

```
DYNAMIC ARRAY attributes-list WITH DIMENSION rank OF data-type
```

where attributes-list is:

```
ATTRIBUTES ( attribute = "value" [, ... ] )
```

1. `rank` an be an integer literal of 1, 2, or 3. Default is 1.
2. `data-type` can be a data type, a record definition, a user defined type, a built-in class, an imported package class, or a Java class.
3. `attribute` is an attribute to extend the array definition with properties.
4. `value` is the value for the array definition attribute, it is optional for boolean attributes.

Syntax 3: Java array definition

```
ARRAY [ ] OF java-type
```

1. `java-type` must be a Java class or a simple data type that has a corresponding primitive type in Java, such as INTEGER (int), FLOAT (double).

Usage

An array defines an ordered set of elements.

Array variables can invoke methods specific to the array types.

The type of the array elements can be of a simple type or structured records.

Consider using dynamic arrays instead of static arrays.

Java-style arrays will only be useful to interface with Java calls.

Static and dynamic arrays can be defined with the ATTRIBUTES() clause, to specify meta-data information for the variable. This feature is especially used when defining variables for XML-based Web Services. For more details about XML attributes, see Attributes to customize XML serialization on page 3523.

Related concepts

Primitive Data types on page 289
Selecting the correct data type assists you in the input, storage, and display of your data.

Using Java arrays on page 2215  
Variables on page 390  
Explains how to define program variables.

Static arrays

Static arrays have a predefined and limited size.

Defining static arrays

Static arrays can store a one-, two- or three-dimensional array of variables, all of the same type. An array member can be any type except another array (ARRAY ... OF ARRAY).

```
MAIN
  DEFINE custlist ARRAY[100] OF RECORD
    id INTEGER,
    name VARCHAR(50)
  END RECORD
  LET custlist[50].id = 12456
  LET custlist[50].name = "Beerlington"
END MAIN
```

Static array starting index

The first element in a static array is at index position 1:

```
DISPLAY arr[1].name
```

Multi-dimensional static arrays

The multi-dimensional array syntax (ARRAY[i][j][k]) specifies static arrays defined with an explicit size for all dimensions. Static arrays have a size limit. The biggest static array size you can define is 65535.

A single array element can be referenced by specifying its coordinates in each dimension of the array.

Avoid using large static arrays; All elements of static arrays are allocated and initialized when the program starts, even if the array is not used.

```
MAIN
  DEFINE a1 ARRAY[100] OF INTEGER
  LET a1[50] = 12456
  LET a1[5000] = 12456  -- Runtime error!
END MAIN
```

Element types

The elements of a static array variable can be defined as a structured record:

```
MAIN
  DEFINE arr ARRAY[50] OF RECORD
    key INTEGER,
    name CHAR(10),
    address VARCHAR(200),
    contacts ARRAY[50] OF VARCHAR(20)
  END RECORD
  LET arr[1].key = 12456
  LET arr[1].name = "Scott"
  LET arr[1].contacts[1] = "Bryan COX"
  LET arr[1].contacts[2] = "Mike FLOWER"
```
Passing static arrays to functions

Static arrays are passed by value to functions. This is not recommended, as all array members will be copied on the stack.

A static array cannot be returned from a function.

Consider using dynamic arrays if you need to pass/return a list of elements to/from functions.

Using array methods

Array methods can be used on static arrays; however, these methods are designed for dynamic arrays and are not appropriate for static arrays.

Related concepts

Array methods on page 424
Native BDL arrays and Java arrays can be used to invoke built-in methods.

Controlling out of bound in static arrays on page 421

Controlling out of bound in static arrays

By default, when an index is lower than 1 or greater as the static array length, fglrun raises error -1326, and this error is not trappable.

Note: When using a dynamic array, new elements are automatically allocated, if the index is greater than the actual array size. The error -1326 will only occur when the index is lower than 1.

Raising an index out of bounds error is normal. However, in some situations, code must execute without error and evaluate expressions using indexes that are greater than the size of the array, especially with boolean expressions in IF statements:

```
IF index <= max_index AND arr[index].name IS NOT NULL THEN
  ...
END IF
```

In this example, by default (when not using the OPTIONS SHORT CIRCUIT instruction), all parts of a boolean expression need to be evaluated, and the runtime system must get the value of the arr[index] element.

If such kind of code cannot be changed and must execute, use the fglrun.arrayIgnoreRangeError FGLPROFILE entry, to control the behavior of the runtime system when an array index is out of bounds:

```
fglrun.arrayIgnoreRangeError = true
```

When this FGLPROFILE entry is set to true, the runtime system will return the first element of the array if the index is out of bounds, and continues with the normal program flow.

Unless existing code is relying on the relaxed behavior, it is better to keep the default settings and get array out of bounds errors when the array index is invalid.

Change the code as follows, to detatch the index checking from its usage:

```
IF index <= max_index THEN
  IF arr[index].name IS NOT NULL THEN
    ...
  END IF
END IF
```
Or, use the `OPTIONS SHORT CIRCUIT` compiler directive, to control boolean expression evaluation. You can then put both conditions in the same `IF` statement. When the first condition is false, the second condition is not evaluated:

```sql
OPTIONS SHORT CIRCUIT
...
IF index <= max_index AND arr[index].name IS NOT NULL THEN
...
END IF
```

**Related concepts**

*Dynamic arrays* on page 422

---

**Dynamic arrays**

**Defining dynamic arrays**

Dynamic arrays are defined with the `DYNAMIC ARRAY` syntax and specify an array with a variable size. Dynamic arrays have no theoretical size limit. The elements of dynamic arrays are allocated automatically by the runtime system, based on the indexes used.

```sql
MAIN
  DEFINE a1 DYNAMIC ARRAY OF INTEGER
  LET a2[5000] = 12456 -- Automatic allocation for element 5000
END MAIN
```

**Dynamic array starting index**

The first element in a dynamic array is at index position 1:

```sql
DISPLAY arr[1].name
```

**Initializing arrays**

Arrays can be initialized in their definitions with variable initializers:

```sql
DEFINE num DYNAMIC ARRAY OF STRING = [ "One", "Two", "Three" ]
```

**Element types**

The elements of a dynamic array variable are typically defined as a structured record:

```sql
MAIN
  DEFINE arr DYNAMIC ARRAY OF RECORD
      key INTEGER,
      name VARCHAR(30),
      address VARCHAR(200),
      contacts DYNAMIC ARRAY OF VARCHAR(20)
    END RECORD
  LET arr[1].key = 12456
  LET arr[1].name = "Scott"
  LET arr[1].contacts[1] = "Bryan COX"
  LET arr[1].contacts[2] = "Mike FLOWER"
END MAIN
```

**Automatic element allocation**

When a dynamic array element does not exist, it is automatically allocated before it is used. For example, when you assign an array element with the `LET` instruction by specifying an array index greater than the current length of the
array, the new element is created automatically before assigning the value. This is also true when using a dynamic array in a FOREACH loop or when dynamic array elements are used as r-values, for example in a DISPLAY.

**Important:**

Pay attention to automatic element allocation in dynamic arrays. The following code example creates an additional element because at each iteration, the runtime system must allocate a new element to fetch the row from the database. As a result, you need to remove the last element of the array after the FOREACH loop:

```plaintext
DEFINE arr DYNAMIC ARRAY OF RECORD
   key INTEGER,
   name VARCHAR(30)
END RECORD,
DECLARE c1 CURSOR FOR SELECT ckey, cname FROM mytable
LET x=1
FOREACH c1 INTO arr[x].*
   LET x=x+1
END FOREACH
CALL arr.deleteElement(x)

-- A more elegant way to fetch rows into an array:
TYPE my_type RECORD LIKE mytable.*
DEFINE arr DYNAMIC ARRAY OF my_type,
   rec my_type,
   x INTEGER
DECLARE c1 CURSOR FOR SELECT * FROM mytable
LET x=1
FOREACH c1 INTO rec.*
   LET arr[x:=x+1].* = rec.*
END FOREACH
```

**Passing and returning dynamic arrays to functions**

Dynamic arrays are passed (or returned) by reference to/from functions.

The dynamic array can be modified inside the called function, and the caller will see the modifications:

```plaintext
MAIN
   DEFINE a DYNAMIC ARRAY OF INTEGER
   CALL fill(a)
   DISPLAY a.getLength() -- shows 2
END MAIN

FUNCTION fill(x)
   DEFINE x DYNAMIC ARRAY OF INTEGER
   CALL x.appendElement()
   CALL x.appendElement()
END FUNCTION
```

See also **Passing dynamic arrays as parameter** on page 497.

The reference to dynamic arrays created inside a function can be returned from the function:

```plaintext
MAIN
   DEFINE a DYNAMIC ARRAY OF INTEGER
```
LET a = create()
    DISPLAY a.getLength() -- shows 2
END MAIN

FUNCTION create()
    DEFINE x DYNAMIC ARRAY OF INTEGER
    CALL x.appendElement()
    CALL x.appendElement()
    RETURN x
END FUNCTION

See also Returning dynamic arrays from functions on page 500.

Using multidimensional dynamic arrays

Multidimensional dynamic arrays can be defined by using the WITH DIMENSION syntax.

Array methods can be used on multidimensional arrays with the brackets notation:

MAIN
    DEFINE a2 DYNAMIC ARRAY WITH DIMENSION 2 OF INTEGER
    DEFINE a3 DYNAMIC ARRAY WITH DIMENSION 3 OF INTEGER
    LET a2[50,100] = 12456
    LET a2[51,1000] = 12456
    DISPLAY a2.getLength() -- shows 51
    DISPLAY a2[50].getLength() -- shows 100
    DISPLAY a2[51].getLength() -- shows 1000
    LET a3[50,100,100] = 12456
    LET a3[51,101,1000] = 12456
    DISPLAY a3.getLength() -- shows 51
    DISPLAY a3[50].getLength() -- shows 100
    DISPLAY a3[51].getLength() -- shows 101
    DISPLAY a3[50,100].getLength() -- shows 100
    DISPLAY a3[51,101].getLength() -- shows 1000
    CALL a3[50].insertElement(10) -- inserts at 50,10
    CALL a3[50,10].insertElement(1) -- inserts at 50,10,1
END MAIN

Related concepts

Array methods on page 424
Native BDL arrays and Java arrays can be used to invoke built-in methods.

Array methods

Native BDL arrays and Java arrays can be used to invoke built-in methods.

For example, to clear a dynamic array, use the clear() method:

DEFINE arr DYNAMIC ARRAY OF ...
CALL arr.clear()

For the list of native array methods, see DYNAMIC ARRAY methods on page 2394.
For the list of Java array methods, see Java Array type methods on page 2404.

Copying and assigning arrays

Arrays can be fully copied or assigned by reference.

Prerequisites to copy and assign arrays

To make a copy of a static or dynamic array, or when assigning a reference of a dynamic array to another variable, the source and destination arrays must have the same structure.
The **copyTo()** method

An array can be copied to another array with the **copyTo()** method. This method makes a full copy of all elements of the source array:

```plaintext
MAIN
    DEFINE left, right DYNAMIC ARRAY OF RECORD
       key INTEGER
    END RECORD
    LET left[1].key = 123
    CALL left.copyTo(right)  -- copies the array
    DISPLAY right[1].key     -- shows 123
    LET right[1].key = 456
    DISPLAY left[1].key      -- shows 123
END MAIN
```

**Note:** The **copyTo()** method can be used with dynamic and static arrays.

**Assigning dynamic array references**

The reference of a dynamic array can be copied to another variable defined with the same type, by specifying the name of the array, as if it was a simple data type.

After assigning a dynamic array to another variable, both variables will reference the same dynamic array object and elements:

```plaintext
MAIN
    DEFINE left, right DYNAMIC ARRAY OF RECORD
       key INTEGER
    END RECORD
    LET left[1].key = 123
    LET right = left      -- copies the reference
    DISPLAY right[1].key  -- shows 123
    LET right[1].key = 456
    DISPLAY left[1].key   -- shows 456
END MAIN
```

**The .* notation with arrays**

For backward compatibility, the compiler allows the .* notation to assign an array to another array with the same structure.

**Important:** Assigning/copying arrays with the .* notation is supported for backward compatibility. Consider using the array names without the .* suffix: LET arr2 = arr1, this syntax has the same effect.

When using the .* notation, static array elements are copied **by value** (except objects and BYTE/TEXT members):

```plaintext
MAIN
    DEFINE left, right ARRAY[10] OF RECORD
       key INTEGER
    END RECORD
    LET left[1].key = 123
    LET right.* = left.*   -- copies the array
    DISPLAY right[1].key   -- shows 123
    LET right[1].key = 456
    DISPLAY left[1].key    -- shows 123
END MAIN
```

**Note:** When using the .* notation with dynamic arrays, the reference to the dynamic array is copied.
Examples

Array usage examples.

**Example 1: Using static and dynamic arrays**
This example illustrates the syntax difference of static and dynamic arrays.

```plaintext
MAIN
    DEFINE a1 DYNAMIC ARRAY OF INTEGER
    DEFINE a2 DYNAMIC ARRAY WITH DIMENSION 2 OF INTEGER
    DEFINE a3 ARRAY[10,20] OF RECORD
        id INTEGER,
        name VARCHAR(100),
        birth DATE
    END RECORD
    LET a1[5000] = 12456
    LET a2[5000,300] = 12456
    LET a3[5,1].id = a1[50]
    LET a3[5,1].name = 'Scott'
    LET a3[5,1].birth = TODAY
END MAIN
```

**Example 2: Filling a dynamic array with rows**
This example shows how to fetch database rows into a dynamic array.

Automatic allocation of dynamic array element in the FOREACH statement creates an additional element that needs to be deleted after the loop:

```plaintext
SCHEMA stores
MAIN
    DEFINE custarr DYNAMIC ARRAY OF RECORD LIKE customer.*
    DEFINE index INTEGER

DATABASE stores

    DECLARE curs CURSOR FOR SELECT * FROM customer
    LET index = 1
    FOREACH curs INTO custarr[index].*
        LET index = index+1
    END FOREACH
    CALL custarr.deleteElement(custarr.getLength())
    DISPLAY "Number of rows found: ", custarr.getLength()
    FOR index=1 TO custarr.getLength()
        DISPLAY custarr[index].*
    END FOR

END MAIN
```

Dictionaries

A dictionary holds an unordered collection of elements accessed by a key.

**Understanding dictionaries**
This is an introduction to dictionaries.

Dictionaries can store an unordered collection of elements, that will be accessed by key, rather than by index as in arrays.
Dictionaries are used to implement a hash map (a.k.a. associative array).

The keys must be character string:

```plaintext
DEFINE dict DICTIONARY OF INTEGER
LET dict["abcdef"] = 999
```

Dictionary elements can be of simple built-in types such as INTEGER, VARCHAR(n), RECORD structured types, or user-defined types. A dictionary can also be defined with built-in classes, imported module classes (from the Web Services extensions for example), or Java classes.

**Related concepts**
- Arrays on page 418
Arrays (static or dynamic) allow you to handle an ordered collection of elements.

**DICTIONARY**

A dictionary defines an associative array (hash-map) of elements.

**Syntax**

```plaintext
DICTIONARY \ attributes-list \ OF data-type
```

where attributes-list is:

```plaintext
ATTRIBUTES ( attribute \ = \ "value" \ [, \ ... ] )
```

1. data-type can be a data type, a record definition, a user defined type, a built-in class, an imported package class, or a Java class.
2. attribute is an attribute to extend the dictionary definition with properties.
3. value is the value for the dictionary definition attribute, it is optional for boolean attributes.

**Usage**

A dictionary defines an associative array of unordered elements, accessed by a key.

Dictionary variables can invoke methods specific to the dictionary types.

The elements of the dictionary can be of a simple type, or structured records.

For example, to define a dictionary of strings:

```plaintext
DEFINE dict DICTIONARY OF STRING
```

The dictionary subscript syntax consists of a character string expression (the key), specified between square brackets.

The result of the subscript syntax can be used as l-value (as target variable in assignments):

```plaintext
LET dict["abcdef"] = "the value"
```

or as r-value (in expressions):

```plaintext
DISPLAY dict["abcdef"]
```

Dictionary elements are automatically created when needed. For more details, see DICTIONARY in action on page 428.

**Related concepts**
- Primitive Data types on page 289
Selecting the correct data type assists you in the input, storage, and display of your data.

**Variables** on page 390
Explains how to define program variables.

**DICTIONARY in action**

**Defining dictionaries**

Dictionaries are defined with the **DICTIONARY** syntax. Dictionaries have no theoretical size limit.

The elements of dictionaries are allocated automatically by the runtime system. Dictionary elements are accessed by key:

```plaintext
MAIN
  DEFINE dict DICTIONARY OF DECIMAL(10,2)
  LET dict["bananas"] = 10540.45
  LET dict["apples"] = 3487.55
  LET dict["oranges"] = 234.10
  DISPLAY dict.getLength() -- Shows 3
END MAIN
```

**Element keys**

In a dictionary, elements are identified by a key. Similar to an array index, a key allows access to the element it references.

A key must be a hashable character string with a given length. For example, "Mike Hurn", "AXF98234".

Obviously, keys must be unique.

Keys are used in dictionary subscripts. The key value must be enclosed in square brackets:

```plaintext
LET dict[ "the key" ].member = value
```

The **DICTIONARY** class provides following methods related to keys:

- To check if a key exists, use the **contains()** method.
- To delete an element, use the **remove()** method.
- To get all keys of a dictionary, use the **getKeys()** method.

**Element types**

The elements of a dictionary variable are typically defined with simple data types, with a structured record, or with a user-defined type:

```plaintext
TYPE t_contact RECORD
  name VARCHAR(30),
  address VARCHAR(100),
  birth DATE
END RECORD
MAIN
  DEFINE contact DICTIONARY OF t_contact
  LET contact["EFC456"].name = "Mike Campbell"
  LET contact["EFC456"].address = "5, Big tree St."
  LET contact["EFC456"].birth = MDY(10,23,1999)
  DISPLAY contact["EFC456"].*
END MAIN
```
Automatic element allocation

When a dictionary element does not exist, it is automatically allocated before it is used.

For example, when you assign a dictionary element with the \texttt{LET} instruction by specifying a key that does not exist yet, the new element is created automatically before assigning the value.

Similarly, when used in an expression, the element will be automatically allocated if it does not exist, and the element value will evaluate to \texttt{NULL}:

```main
DEFINE dict DICTIONARY OF INTEGER
DISPLAY (dict["unexisting"] IS NULL) -- shows 1 (TRUE)
END MAIN
```

Dictionary elements are also automatically created in a \texttt{FOREACH} loop or when dictionary elements are used as r-values, for example in a \texttt{DISPLAY}.

Consider using the \texttt{contains()} method, to check if a given element key is already existing.

```main
DEFINE dict DICTIONARY OF INTEGER
LET dict["id1"] = 33  -- Created automatically
DISPLAY dict["id2"]  -- Created automatically
IF dict.contains("id3") THEN
    DISPLAY dict["id3"]  -- Not displayed as it does not exist
END IF
END MAIN
```

Passing and returning dictionaries to functions

Dictionaries are passed (or returned) by reference to/from functions.

The dictionary can be modified inside the called function, and the caller will see the modifications.

```main
DEFINE dict DICTIONARY OF INTEGER
CALL fill(dict)
DISPLAY dict.getLength() -- shows 2
END MAIN

FUNCTION fill(x)
DEFINE x DICTIONARY OF INTEGER
LET x["ABC"] = 123
LET x["DEF"] = 456
END FUNCTION
```

Comparing dictionaries

\texttt{DICTIONARY} variables are not comparable: Comparing two \texttt{DICTIONARY} variable, or using a dictionary in any expression, will produce the compiler error -4340:

```main
DEFINE d1, d2 DICTIONARY OF INTEGER
LET d1["ABC"] = 123
LET d2["DEF"] = 456
DISPLAY "1:", (d1 == d2)
The variable 'd1' is too complex a type to be used in an expression.
See error number -4340.
The variable 'd2' is too complex a type to be used in an expression.
See error number -4340.
```
When comparing records with `rec1.* == rec2.*` expression, if the record type contains `DICTIONARY` members, the comparison always evaluates to `FALSE`:

```plaintext
MAIN
    DEFINE r1, r2 RECORD
    pkey INTEGER,
    dic DICTIONARY OF STRING
    END RECORD
    LET r1.pkey = 101
    LET r1.dic["aaa"] = "xxx"
    LET r2.pkey = 101
    LET r2.dic["aaa"] = "xxx"
    DISPLAY IIF( r1.* == r2.*, "TRUE", "FALSE" )  -- Shows "FALSE"
END MAIN
```

In order to compare the content of two dictionaries, write a loop to compare all elements individually.

**Related concepts**

[DICTIONARY methods](#) on page 2401

**Dictionary methods**

Dictionary variables can be used to invoke built-in methods.

For example, to clear a dictionary, use the `clear()` method:

```plaintext
DEFINE dict DICTIONARY OF ...
CALL dict.clear()
```

For the list of dictionary methods, see [DICTIONARY methods](#) on page 2401.

**Examples**

Dictionary usage examples.

**Example 1: Simple DICTIONARY usage**

Fill a `DICTIONARY` and show existing elements.

```plaintext
MAIN
    DEFINE dict DICTIONARY OF RECORD
       name VARCHAR(50),
       born DATE
    END RECORD
    DEFINE keys DYNAMIC ARRAY OF STRING
    DEFINE i INT
    INITIALIZE dict TO NULL
    -- 1) put some values into the dictionary
    LET dict["Mike"].name = "Mike"
    LET dict["Mike"].born = mdy(12,23,1998)
    --
    LET dict["Cliff"].name = "Cliff"
    LET dict["Cliff"].born = mdy(02,11,2001)
    -- 2) manipulate an element
    LET dict["Cliff"].born = mdy(4,10,1961)
    -- 3) get key list and display all elements
    LET keys = dict.getKeys()
    FOR i = 1 TO keys.getLength()
```
DISPLAY i, " ", dict[keys[i]].
END FOR

-- 4) check that an element exists
DISPLAY dict.contains("Cliff")

-- 5) removing an element
CALL dict.remove("Cliff")

-- 6) dictionary size
DISPLAY dict.getLength()

END MAIN

Types

Types can be defined by the programmer to centralize the definition of complex/structured variables.

Understanding type definition

This is an introduction to types.

The TYPE instruction declares a user-defined type, which can be based on:

- **Primitive data types** (INTEGER, VARCHAR(n))
- **Records** (RECORD ... END RECORD, RECORD ... LIKE tabname. *)
- **Arrays** (DYNAMIC ARRAY OF ...)
- **Dictionaries** (DICTIONARY OF ...)
- **Function references** (FUNCTION name(type,...) RETURNS ...)
- **Interfaces** (INTERFACE ... END INTERFACE)

Once declared, a type can be referenced in the declaration of program variables, or in other types.

Types are typically defined to avoid the repetition of complex structured types.

**Tip:** User-defined types improve code readability by centralizing data structure definitions. Consider using PUBLIC TYPE definitions to share types across modules with IMPORT FGL.

Types can also be completed with methods to encapsulate the data, and use the concept of polymorphism with interfaces.

What are anonymous types?

While user-defined types are defined with the TYPE instruction and are identified by a name, anonymous types are created automatically by the runtime system when defining variables with a complete type specification (not with a user-defined type).

For example:

```sql
DEFINE name VARCHAR(50)
DEFINE cust_rec RECORD
  cust_id INTEGER,
  cust_name VARCHAR(50)
END RECORD
DEFINE cust_list DYNAMIC ARRAY OF
```

The above

**Related concepts**

[Variables on page 390](#)
Explains how to define program variables.

**TYPE**

Types define a synonym for a base or structured data type.

**Syntax:**

```
{PUBLIC|PRIVATE} TYPE identifier type-definition [, ...]
```

where `type-definition` is:

```
{ data-type
  LIKE [dbname:]tabname.colname
  | attributes-list }
```

or `type-definition` is a function type definition:

```
FUNCTION function-name {
  parameter-name data-type | attributes-list | [, ...]
  | attributes-list }
  RETURNS | data-type | attributes-list | [, ...] }
```

where `attributes-list` is:

```
ATTRIBUTES ( attribute | "value" | [, ...] )
```

1. `identifier` is the name of the type to be defined.
2. `data-type` can be a primitive data type, a record structure, an array definition, a dictionary definition, a built-in class, an imported package class, or a Java class.
3. `dbname` identifies a specific database schema file.
4. `tabname.colname` references a column defined in the database schema file.
5. `attribute` is an attribute to extend the type definition with properties.
6. `value` is the value for the type attribute, it is optional for boolean attributes.

**Usage**

User-defined types enforce reusability and simplify programming, by centralizing data structure definitions at a single place.

When defining types with the `LIKE` clause, the data types are taken from the database schema file at compile time. Make sure that the schema file of the database schema during development corresponds to the database schema of the production database; otherwise the types defined in the compiled version of your modules will not match the table structures of the production database.

Types referencing data types can use the `ATTRIBUTES()` clause, to specify meta-data information for the type member. This feature is especially used when defining types for XML-based Web Services. For more details about XML attributes, see Attributes to customize XML serialization on page 3523.

Function reference types can be used to define program variables, that will hold a function reference. See Function references on page 449 for more details.

**Related concepts**

Primitive Data types on page 289
Selecting the correct data type assists you in the input, storage, and display of your data. 

**Records** on page 408  
Records allow structured program variables definitions.

**Arrays** on page 418  
Arrays (static or dynamic) allow you to handle an ordered collection of elements.

**Using types in programs**
Define a type as a synonym for an existing data type, or as a shortcut for records and array structures.

**Defining and using a type**
After declaring a type, it can be used as a normal data type to define variables.

```plaintext
TYPE t_customer RECORD  
cust_num INTEGER,  
cust_name VARCHAR(50),  
cust_addr VARCHAR(200)  
END RECORD

...  
DEFINE c1 t_customer  
...

DEFINE o1 RECORD  
  order_num INTEGER,  
  customer t_customer,  
...  
END RECORD

...  
DEFINE custlist DYNAMIC ARRAY OF t_customer
```

The scope of a type is the same as for variables and constants. Types can be global, module-specific, or local to a function.

**Using a type defined in another module**
A good practice is to define types that belong to the same domain in a single .4gl module, and import that module in the modules where the types are needed.

By default, module-specific types are private; They cannot be used by an other module of the program. To make a module type public, add the PUBLIC keyword before TYPE. When a module type is declared as public, it can be referenced by another module by using the IMPORT FGL instruction:

```plaintext
-- customers.4gl
PUBLIC TYPE t_ord RECORD  
  ord_id INTEGER,  
  ord_date DATE,  
  ord_total DECIMAL(10,2)  
END RECORD
PUBLIC TYPE t_cust RECORD  
  cust_id INTEGER,  
  cust_name VARCHAR(50),  
  orders DYNAMIC ARRAY OF t_ord,  
...
END RECORD

-- main.4gl
IMPORT FGL customers
MAIN
  DEFINE custlist DYNAMIC ARRAY OF t_cust
```
Types for function references

Types can also be used to declare a function signature, in order to define program variables that reference functions with that signature:

```plaintext
TYPE callback_function FUNCTION(p1 INT, p2 INT) RETURNS INT
DEFINE v callback_function
... 
LET v = FUNCTION add 
...
```

Completing types with methods and interfaces

Types define data structures that can be manipulated from methods, to use the concept of encapsulation and make your code more robust:

```plaintext
PUBLIC TYPE Circle RECORD
  pos_x FLOAT,
  pos_y FLOAT,
  diameter FLOAT
END RECORD

PUBLIC FUNCTION (c Circle) area() RETURNS FLOAT
  RETURN util.Math.pi() * (c.diameter / 2) ** 2
END FUNCTION
```

Furthermore, to manipulate similar but different data structures with the same set of methods, you can define interfaces to indirectly manipulate different instances of types:

```plaintext
PUBLIC TYPE Circle RECORD ... END RECORD
PUBLIC FUNCTION (c Circle) area() RETURNS FLOAT ... END FUNCTION

PUBLIC TYPE Rectangle RECORD ... END RECORD
PUBLIC FUNCTION (r Rectangle) area() RETURNS FLOAT ... END FUNCTION

PUBLIC TYPE Triangle RECORD ... END RECORD
PUBLIC FUNCTION (t Triangle) area() RETURNS FLOAT ... END FUNCTION

PUBLIC TYPE Shape INTERFACE
  area() RETURNS FLOAT
END INTERFACE
```

Related concepts

- **Records** on page 408
  Records allow structured program variables definitions.
- **Arrays** on page 418
  Arrays (static or dynamic) allow you to handle an ordered collection of elements.

Examples

**Example 1: User type with a record structure**

The example shows how to define a user type as a RECORD.

```plaintext
TYPE t_customer RECORD
  cust_num INTEGER,
  cust_name string,
  cust_email email
END RECORD
```
cust_name VARCHAR(50),
cust_addr VARCHAR(200)
END RECORD

MAIN
DEFINE custrec t_customer
DEFINE custarr DYNAMIC ARRAY OF t_customer

LET custrec.cust_num = 123
LET custrec.cust_name = "Mike Pantock"

LET custarr[10].* = custrec.*

END MAIN

Example 2: User type defined in a module

This example shows how to use a type defined in another module.

The module "type_order.4gl":

PUBLIC TYPE rpt_order RECORD
  order_num INTEGER,
  store_num INTEGER,
  order_date DATE,
  cust_num INTEGER,
  fac_code CHAR(3)
END RECORD

The main program:

IMPORT FGL type_order

MAIN
  DEFINE o type_order.rpt_order
  CONNECT TO "custdemo"

  DECLARE order_c CURSOR FOR
    SELECT orders.*
    FROM orders ORDER BY cust_num
  START REPORT order_list
  FOREACH order_c INTO o.*
  OUTPUT TO REPORT order_list(o.*)
  END FOREACH
  FINISH REPORT order_list

END MAIN

REPORT order_list(ro)
  DEFINE ro rpt_order
  FORMAT
    ON EVERY ROW
      PRINT ro.order_num, , ro.order_date
  END REPORT

Example 3: User type for function reference

This example shows how to define a user type from a function type.

TYPE t_func_ref FUNCTION (p1 INT, p2 INT) RETURNS INT
Functions

Describes user defined functions.

Understanding functions

This is an introduction to functions.

Function basics

Functions are named program blocks containing a set of statements to be executed when the function is invoked with a CALL statement, or when the function is used in an expression, or when the function is registered in a callback mechanism like WHENEVER ERROR CALL.

Parameters and return values

Functions can get input parameters and return zero, one or several values.

Scope of functions

A function is defined in a program module, and is by default visible to all modules (it is PUBLIC). A function can be declared as PRIVATE to the module where it is defined. In this case the function is hidden to other modules. This concept is explained in Scope of a function on page 440.

Function references

It is possible to hold a function reference in a program variable. The variable can then be assigned with any function reference that is declared with the same function signature as the type used to define the variable.

Note: Function references apply only to regular functions. It is not possible to use a method reference.

Functions as methods for a type

When declaring a function with a receiver variable and receiver type enclosed in parentheses before the function name, it becomes a method acting on the specified type.

Use methods in conjunction with interfaces to write clear robust code by using encapsulation and polymorphism concepts.

Function attributes

Functions can be declared with function attributes, to define additional information on the function.
**FUNCTION definitions**

A FUNCTION definition defines a named procedure with a set of statements.

**Syntax 1 (legacy function syntax):**

```
[PUBLIC|PRIVATE] FUNCTION function-name ( [ parameter-name |, ... ] )
  [ DEFINE parameter-name data-type |, ... ]
  [ local-declaration |, ... ]
  [ instruction ]
  [ RETURN expression |, ... ]
END FUNCTION
```

1. `function-name` is the function identifier.
2. `parameter-name` is the name of a formal argument of the function.
3. `data-type` can be a primitive data type, a user defined type, a built-in class, an imported package class, or a Java class.
4. `local-declaration` is a DEFINE, CONSTANT or TYPE instruction.
5. `instruction` is a language statement to be executed when the function is invoked.
6. `expression` is a value to be returned by the function.

**Syntax 2 (fully typed function):**

```
[PUBLIC|PRIVATE] FUNCTION function-name ( [ parameter-name data-type |, ... ]
  [ attributes-list ]
  [ record-name record-type INOUT ]
  [ ]
) [ attributes-list ]
  [ RETURNS returns-specification ]
  [ local-declaration |, ... ]
  [ instruction ]
  [ RETURN expression |, ... ]
END FUNCTION
```

where `attributes-list` is:

```
ATTRIBUTES ( attribute = "value" |, ... )
```

where `returns-specification` is:

```
( data-type | attributes-list )
( ( data-type | attributes-list ) |, ... )
( )
```

1. `function-name` is the identifier of the function.
2. `parameter-name` is the name of a formal argument of the function.
3. `data-type` can be a primitive data type, a user defined type, a built-in class, an imported package class, or a Java class.
4. `record-name` is the name of a parameter defined with a record TYPE.
5. `record-type` is a record TYPE.
6. `attributes-list` is a comma-separated list of name = value pairs or name attributes.
7. **attribute** is an attribute name to extend the function definition with properties.
8. **value** is the value for the function attribute, it is optional for boolean attributes.
9. **local-declaration** is a **DEFINE**, **CONSTANT** or **TYPE** instruction.
10. **instruction** is a language statement to be executed when the function is invoked.
11. **expression** is a value to be returned by the function.

**Syntax 3 (methods):**

```plaintext
[PUBLIC|PRIVATE] FUNCTION ( receiver-name receiver-type ) method-name (
  parameter-name data-type
  ↓ record-name record-type INOUT
  ↓ ...
) RETURNS returns-specification 
  ↓ local-declaration [...] ↓ 
  ↓ instruction 
  ↓ [ RETURN expression ↓, ↓ ... ↓ ] ↓ 
  ↓ [...] ↓ 
END FUNCTION
```

where **returns-specification** is:

```plaintext
{ data-type
  ↓ ( data-type ↓, ↓ ... ↓ )
  ↓ ()
}
```

1. A **(receiver-name receiver-type)** clause defines a function as a method for a user-defined type.
2. **receiver-name** is the identifier of the receiver referenced in the method body.
3. **receiver-type** is the user-defined type to be the target of this method.
4. **method-name** is the identifier of the method.
5. **parameter-name** is the name of a formal argument of the method.
6. **data-type** can be a primitive data type, a user defined type, a built-in class, an imported package class, or a Java class.
7. **record-name** is the name of a parameter defined with a record **TYPE**.
8. **record-type** is a record **TYPE**.
9. **local-declaration** is a **DEFINE**, **CONSTANT** or **TYPE** instruction.
10. **instruction** is a language statement to be executed when the method is invoked.
11. **expression** is a value to be returned by the method.

**Function names**

Like other identifiers, function names are case-insensitive. However, always consider using the same naming convention when defining and invoking functions.

**Note:** If the function name is also the name of a built-in function, an error occurs at link time, even if the program does not reference the built-in function.

**Function definition using legacy syntax**

The following example shows a function definition using the legacy syntax, with parameter definition in the function body:

```plaintext
FUNCTION split(str, len)
  DEFINE str STRING, len INT
```
RETURN str.substring(1, len),
        str.substring(len+1, str.getLength())
END FUNCTION

This syntax does not define a function with a complete signature.

**Function definition with complete function type**

By specifying data types in the parameter list, you define a function with a complete function type.

This syntax allows better compilation checking:

```plaintext
FUNCTION split(str STRING, len INT) RETURNS (STRING, STRING)
    RETURN str.substring(1, len),
           str.substring(len+1, str.getLength())
END FUNCTION
```

Braces after the `RETURNS` clause are not required, when the function returns a single value:

```plaintext
FUNCTION count_items(sid INT) RETURNS INT
    DEFINE cnt INT
    SELECT COUNT(*) INTO cnt FROM stock WHERE stock_id = sid
    RETURN cnt
END FUNCTION
```

When the function returns a single value/type, it is also possible to enclose the type in parentheses:

```plaintext
FUNCTION count_items(sid INT) RETURNS (INT)
    ...  
END FUNCTION
```

In the next example, the function has no parameters and does not return values:

```plaintext
FUNCTION clean_debug_log() RETURNS ()
    CALL os.Path.delete("log.txt")
END FUNCTION
```

**Methods**

A `FUNCTION` definition using a receiver identifier and type defines a method for this type. Specify the receiver identifier and type before the method name:

```plaintext
TYPE Rectangle RECORD
    height, width DOUBLE PRECISION
END RECORD

FUNCTION (r Rectangle) area() RETURNS DOUBLE PRECISION
    RETURN r.height * r.width
END FUNCTION
```

For more details see Methods on page 452.

**Function with record passed as reference**

The `INOUT` keyword can be used to pass records by reference:

```plaintext
FUNCTION init_cust(r CustRec INOUT)
    LET r.cust_id = 0
    LET r.cust_name = "<undefined>"
    LET r.cust_addr = "<undefined>"
    LET r.cust_crea = CURRENT
```
Function with attributes

Function attributes can be used to complete the definition of the function itself, for its parameters and return values:

FUNCTION add(
    p1 INTEGER ATTRIBUTES(json_name = "parameter 1"),
    p2 INTEGER ATTRIBUTES(json_name = "parameter 2")
) ATTRIBUTES(json_name = "function add")
    RETURNS INTEGER ATTRIBUTES(json_name = "return value 1")
    DEFINE var1 INTEGER
    LET var1 = (p1 + p2)
    RETURN var1
END FUNCTION

Note: Function attributes are especially useful to implement RESTful Web Services functions.

For more details see Function attributes on page 455.

Related concepts

Variables on page 390
Explains how to define program variables.
Expressions on page 330
Shows the possible expressions supported in the language.

Scope of a function

A functions can be isolated to control its visibility to other modules.

A FUNCTION block cannot appear within the MAIN block, in a REPORT block, or within another FUNCTION block. A function must be declared at the root level in the source code:

```
MAIN
    ...
END MAIN

FUNCTION myfunc( ... )
    ...
END FUNCTION

REPORT myrep( ... )
    ...
END REPORT
```

By default, functions are PUBLIC; They can be called by any other module of the program.

If a function is only used by the current module, you can hide that function to other modules, to make sure that it will not be called by mistake.

To keep a function local to the module, add the PRIVATE keyword before the function header.

Private functions are only hidden to external modules, all function of the current module can still call local private functions.

```
PRIVATE FUNCTION check_number(n)
    ...
END FUNCTION
```
For better code readability, you can use the `PUBLIC` keyword for functions that are global and visible to all modules:

```plaintext
PUBLIC FUNCTION initialize()
... END FUNCTION
```

When not using the module prefix, function symbols are global to all modules and must be unique. However, with the `IMPORT FGL` method, you can use the module prefix when invoking a function and simplify function naming conventions, since different modules can define functions with the same name:

```plaintext
-- Module svgutils.4gl:
PUBLIC FUNCTION initialize()
... END FUNCTION

-- Module sqlutils.4gl:
PUBLIC FUNCTION initialize()
... END FUNCTION

-- Main program:
IMPORT FGL svgutils
IMPORT FGL sqlutils
MAIN
    CALL svgutils.initialize()
    CALL sqlutils.initialize()
... END MAIN
```

**Related concepts**

- [Importing modules](#) on page 480

Use the `IMPORT ...` instruction to import BDL, C or Java external modules in the current module.

### Local symbol definition

Symbols defined inside a function body are only visible to the function.

### Purpose of local symbols

Local function symbols are part of and only to be used in the function body. They are created and initialized when the function is invoked. They are dropped when the function terminates.

### Defining local symbols

Inside the body of a function, you can define language elements that will only be visible for the function code:

- local constants with the `CONSTANT` instruction,
- local user-defined types with the `TYPE` instruction,
- local variables with the `DEFINE` instruction.

```plaintext
FUNCTION check_customer( cust_id INTEGER )
    CONSTANT c_max = 1000 -- local constant
    TYPE t_cust RECORD LIKE customer.* -- local type
    DEFINE found BOOLEAN -- local variable
    ... END FUNCTION
```

Function arguments and local symbols must use different names, it is not possible to define a local variable with the same name as a function parameter:

```plaintext
FUNCTION func_a(x INTEGER)
```
DEFINE x INTEGER
The symbol 'x' has been defined more than once.
See error number -4319.
LET x = 1
END FUNCTION

Scope of local symbols
Local function symbols are not visible in other program blocks. Global or module variable can use the same name as a local variable: The global or module variable is not visible within the function scope of the local variable using the same name.

DEFINE x INTEGER   -- Declares a module variable
FUNCTION func_a()
  DEFINE x INTEGER -- Declares a local variable
  LET x = 123      -- Assigns local variable
END FUNCTION

FUNCTION func_b()
  LET x = 123 -- Changes the module variable
END FUNCTION

To increase code readability, consider using different names for global, module and local function symbols. A common practice is to use prefixes for global (g_) and module (m_) variables.

Related concepts
Variables on page 390
Explains how to define program variables.

Calling functions
Functions can be invoked, to execute the code they define.

How can functions be invoked?
A function can be invoked in different ways:
1. with the CALL instruction,
2. in an expression,
3. in a callback mechanism.

Note: The symbol used to identify the function to be called can be a static function name, or a variable referencing a function.

Typical function invocation with CALL
In most cases, functions are invoked with the CALL instruction:

FUNCTION open_database(dbname STRING)
  ...
END FUNCTION

...  
  CALL open_database("stock")
**Function invoked as part of an expression**

When a function returns a single value, it can be invoked in an *expression*:

```
MAIN
    DEFINE r INTEGER
    LET r = 500 + add(50,5)
END MAIN

FUNCTION add(x,y) RETURNS INTEGER
    DEFINE x, y INTEGER
    RETURN (x + y)
END FUNCTION
```

**Callback mechanisms**

Genero BDL provides some callback mechanisms where functions are invoked when needed, for example:

- the `WHENEVER ERROR CALL` instruction,
- the `INITIALIZER` form field attribute,
- the `ui.Form.setDefaultInitializer()` method,
- ...

**Recursion**

A function can invoke itself recursively:

```
MAIN
    CALL recursive(1)
END MAIN

FUNCTION recursive(x)
    DEFINE x INTEGER
    DISPLAY "x = ", x
    IF x<10 THEN
        CALL recursive(x+1)
    END IF
END FUNCTION
```

**Important:** Each time a function calls itself, parameters are pushed on the *stack*. A deep level of recursion can result in out of memory errors.

**Calling a function from its reference variable**

A function can be referenced by a variable, and be invoked through this variable in a `CALL` instruction, or in an expression:

```
TYPE t_func_ref FUNCTION (p1 INT, p2 INT) RETURNS INT
DEFINE fr t_func_ref
LET fr = FUNCTION add  -- Function with the same signature as t_func_ref
DISPLAY fr(100,200)
```

**Naming parameters in a function call**

To improve code readability, function parameters can be qualified with the name specified in the *function definition*:

```
MAIN
    CALL cleanup( mode: "full", verbose: TRUE )
END MAIN
```
FUNCTION cleanup( mode STRING, verbose BOOLEAN )
    IF verbose THEN
        DISPLAY "Cleanup mode: ", mode
    END IF
END FUNCTION

Built-in functions and class/object methods can also be invoked with named parameters:

DEFINE s STRING
DISPLAY s.subString(startIndex: 1, endIndex: 3)
...
DEFINE io base.Channel
LET io = base.Channel.create()
CALL io.openClientSocket(host: "localhost", port: 4711, mode: "u", timeout: 0)

If the parameter name in the function call does not match the parameter in the function definition, the compiler will produce error -8420.

Note: Named function call parameters does not imply free ordering, nor does it allow you to omit parameters. All parameters must be specified, in the same order as in the function declaration.

In order to check the parameter names, fglcomp needs to know the function definition. When the called function is defined in another module, use IMPORT FGL to let the compiler know the function definition.

Tip: Named function parameters are provided by the source code completer.

Related concepts
Type conversions on page 311
Explains primitive data type conversion rules of the language.

Runtime stack on page 493
The runtime stack is used to pass/return values to/from functions.

Function parameters
Functions can take parameters, to specialize their behavior.

Purpose of function parameters
The function declaration specifies the name of the function and the identifiers of its formal arguments (if any).

Function parameters hold values passed by the caller, to be used in the body of the function.

Function parameter specification with legacy syntax
With the legacy function definition syntax, the data type of each formal argument of the function must be specified by a DEFINE statement that immediately follows the argument list.

FUNCTION check_address(zipcode, street, city)
    DEFINE zipcode CHAR(5),
    street VARCHAR(100),
    city VARCHAR(50)
    DEFINE found BOOLEAN -- local function variable
    ...
END FUNCTION
Function parameter specification with fully-typed syntax

With the fully-typed function definition syntax, the type of the parameters is specified inside the parentheses:

```bdl
FUNCTION check_address(zipcode CHAR(5), street VARCHAR(100), city VARCHAR(50))
  DEFINE found BOOLEAN -- local function variable
  ...
END FUNCTION
```

Function without parameters

If no argument is needed in a function call, an empty argument list must still be supplied, enclosed between the parentheses:

```bdl
FUNCTION begin_work()
  ...
END FUNCTION
```

Passing primitive type values as parameters

Function arguments using simple data types such as `INTEGER` are passed by value (the values are copied on the stack).

In the following code example, the variable `x` defined in the `MAIN` block will not be modified by the function:

```bdl
MAIN
  DEFINE x INTEGER
  LET x = 123
  CALL myfunc(x)
  DISPLAY x   -- displays 123
END MAIN

FUNCTION myfunc(x)
  DEFINE x INTEGER
  LET x = x + 1
END FUNCTION
```

Note: Parameters of type `TEXT` and `BYTE` are also passed by value, but it's only the large object handler which is copied on the stack. The actual data is pointed to by the handler and therefore it is not copied.

Type conversions

The actual argument in a call to the function need not be of the declared data type of the formal argument. The runtime system will do the appropriate data type conversions when needed. If the data type conversion is not possible, a runtime error occurs.

Genero BDL data type conversion is quite permissive by default. The following example will not raise an error, because the string value is silently converted to a `NULL INTEGER`:

```bdl
MAIN
  CALL add("aaaaa")
END MAIN

FUNCTION add(x)
  DEFINE x INTEGER
  DISPLAY "x = ", x
END FUNCTION
```
Passing complex objects as parameters

Complex structures such as dynamic arrays and objects are passed by reference: The reference of the complex object is copied on the stack and can be used as is inside the function:

```
MAIN
  DEFINE ch base.Channel
  LET ch = base.Channel.create()
  CALL ch.openFile(arg_val(1), "r")
  CALL read_lines(ch)
END MAIN

FUNCTION read_lines(ch base.Channel)
  DEFINE s STRING
  WHILE TRUE
    LET s = ch.readLine()
    IF ch.isEOF() THEN EXIT WHILE END IF
    DISPLAY s
  END WHILE
END FUNCTION
```

Passing records by reference

By default, when using the . * (dot star) notation, records are expanded on the stack: Each field of the record is passed by value like a primitive data type:

```
CALL show_cust_info( r_cust.* )
```

Records can be passed by reference, when specifying the INOUT keyword in the function definition:

```
FUNCTION init_cust(r CustRec INOUT)
  LET r.cust_id = 0
  LET r.cust_name = "<undefined>"
  LET r.cust_addr = "<undefined>"
  LET r.cust crea = CURRENT
END FUNCTION
```

**Important:** Passing records by reference with INOUT is only supported when the called function is defined locally, or is known by the compiler because it was imported with IMPORT FGL.

For more details, see Passing records as parameter on page 494.

**Related concepts**

- **Primitive Data types** on page 289
- Selecting the correct data type assists you in the input, storage, and display of your data.
- **Variables** on page 390
  
  Explains how to define program variables.

Returning values

A function can return values with the RETURN instruction.

**Defining the returned types in a function declaration**

Function definitions can specify the list of data types returned by the function with the RETURNS clause.

**Note:** The RETURNS clause in the function header is not mandatory. However, it is recommended, to define the complete function signature, and take advantage of better code checking by the compiler.
If the function returns a single value, specify the data type after the `RETURNS` clause:

```sql
FUNCTION get_count() RETURNS INT
...
END FUNCTION
```

**Note:** When the function returns a single value/type, it is also possible to enclose the type in parentheses:

```sql
FUNCTION get_count() RETURNS (INT)
```

But for convenience, the parentheses are optional when only one return type is used.

If the function returns a several values, you must specify the data types after the `RETURNS` clause inside parentheses:

```sql
FUNCTION get_address() RETURNS ( CHAR(5), VARCHAR(100), INTEGER )
...
END FUNCTION
```

When the function returns no values, enforce compiler verification by using the `RETURNS()` clause in the function head:

```sql
DEFINE the_list DYNAMIC ARRAY OF STRING

FUNCTION clear_list() RETURNS ()
    CALL the_list.clear()
    RETURN 0
| Invalid number of return values.
| See error number -8415.
END FUNCTION
```

The `RETURNS` types specification can use base data types such as `INTEGER`, user-defined types, and classes.

```sql
SCHEMA stores
TYPE t_cust DYNAMIC ARRAY OF RECORD LIKE customer.*
...
FUNCTION get_cust_list() RETURNS t_cust
...
END FUNCTION
```

**Returning from the function**

Use the `RETURN` instruction in the body of the function, to push a list of values on the stack, and return to the caller.

The `RETURN` instruction takes an optional comma-separated list of expressions:

```sql
FUNCTION get_address() RETURNS ( CHAR(5), VARCHAR(100), INTEGER )
    DEFINE zipcode CHAR(5),
    street VARCHAR(100),
    city INTEGER
...
    RETURN zipcode, street, city
END FUNCTION
```

The next example shows a function returning a single value:

```sql
FUNCTION get_count() RETURNS ()
...
    RETURN count
END FUNCTION
```
If a function does not need to return a value, the RETURN instruction can be used without arguments:

```sql
FUNCTION show_notfound() RETURNS ()
    IF SQLCA.SQLCODE==0 THEN
        RETURN
    END IF
    ...
END FUNCTION
```

**Expressions using functions returning a single value**

When a function returns a single value, it can be invoked in expressions:

```sql
MAIN
    DEFINE r INTEGER
    LET r = 500 + add(50,5)
END MAIN

FUNCTION add(x,y) RETURNS INTEGER
    DEFINE x, y INTEGER
    RETURN (x + y)
END FUNCTION
```

**Calling a function returning a list of values**

Functions returning multiple values must be invoked with a CALL instruction using the RETURNING clause. Values specified in RETURN statement must correspond in number and position to the RETURNING clause of the CALL instruction, and must be of the same or of compatible data types, to the variables in the RETURNING clause of the CALL statement. An error results if the list of returned values in the RETURN statement conflicts in number or in data type with the RETURNING clause of the CALL statement that invokes the function.

```sql
MAIN
    DEFINE zipcode CHAR(5),
        street VARCHAR(100),
        city VARCHAR(50)
    CALL get_address() RETURNING zipcode, street, city
END MAIN

FUNCTION get_default_address() RETURNS (STRING,STRING,STRING)
    RETURN "00000", "<undefined>", "<undefined>"
END FUNCTION
```

**Returning complex structures**

When returning simple built-in types like INTEGER, values are copied on the stack and copied to the caller variables. When returning a RECORD structure, all values are expanded on the stack (it is recommended to define a TYPE for the record structure. It can then be used in the RETURNS clause):

```sql
TYPE t_rec RECORD
    pkey INT,
    name VARCHAR(20)
END RECORD

MAIN
    DEFINE r1 t_rec
    CALL get_rec() RETURNING r1.*
    DISPLAY r1.*
END MAIN
```
FUNCTION get_rec() RETURNS (t_rec)
    DEFINE r t_rec
    LET r.pkey = 999
    LET r.name = "Mike"
    RETURN r.*
END FUNCTION

When returning complex types such as objects or dynamic arrays, the reference of the element are copied on the stack (this means that you can create an object inside a function, and return its reference in the RETURN statement):

MAIN
    DEFINE c base.Channel
    LET c = open_file("myfile.txt")
    ...
END MAIN

FUNCTION open_file(filename)
    DEFINE filename STRING
    DEFINE c base.Channel
    TRY
        LET c = base.Channel.create()
        CALL c.openFile(filename, "r")
        RETURN c
    CATCH
        RETURN NULL
    END TRY
END FUNCTION

Related concepts
Type conversions on page 311
Explains primitive data type conversion rules of the language.
Runtime stack on page 493
The runtime stack is used to pass/return values to/from functions.

Function references
Function can be referenced and invoked dynamically in a CALL instruction, or in an expression.

Purpose of function references
A function reference points to a function definition, that can be called at runtime. The actual function is not known at compile time, only the function type (number and type of parameters and return values) is known.

This feature allows you to manipulate functions dynamically, for example to implement generic module, that can be configured with callback functions.

Function references are based on function types. Referenced functions must be defined with the syntax defining parameter types in parentheses (and the RETURNS clause, if the functions return values). For more details, see FUNCTION syntax 2.

Defining function types
A function type identifies the signature of a function from the number, names and types of parameters and return values of that function:

FUNCTION(p1 INT, p2 INT) RETURNS INT

Note: The name of the parameters is part of the function signature. Function types using the same number of parameters/types and return types, but different parameter names are considered as a different function types by the compiler.
A function type can be used as other types, to declare simple variables, members of a structured RECORD, or arrays:

```
DEFINE fx FUNCTION(p1 INT, p2 INT) RETURNS INT
```

To simplify function reference usage, define a user-type with the TYPE instruction, with the function type that will match functions to be called by reference:

```
TYPE callback_function FUNCTION(p1 INT, p2 INT) RETURNS INT
```

For more details about user-defined type definitions, see Types on page 431.

**Variable definition for function references**

When the user-type for the function reference is available, declare a program variable to hold such function reference:

```
DEFINE callback callback_function
```

For more details about variable definitions, see Variables on page 390.

**Get the FUNCTION reference**

To get the reference of a function, use the FUNCTION keyword followed by the name of the function to be referenced. The function must be defined in the current module, or in a module imported with IMPORT FGL.

```
LET callback = FUNCTION add
```

In the above example, the function "add()" must be defined with the same function type as the "callback" variable.

For more details, see FUNCTION func-spec on page 451.

**Invoking a function with the CALL statement**

Functions referenced in a variable can be invoked with the CALL instruction, by using the variable. The referenced function will be called as in a regular function call:

```
CALL callback(100,200) RETURNING result
```

**Using function references in expressions**

Variables referencing functions can be used in expressions, like in this example:

```
LET get_count_func = FUNCTION get_total_items()
LET c = get_count_func()
LET get_count_func = FUNCTION get_total_elements()
LET c = c + get_count_func()
```

**Passing function references as function parameters**

Like other values, function references can be passed as function parameters:

```
CALL process( FUNCTION add, FUNCTION sub, callback )
...FUNCTION process( f1 callback_function,
                      f2 callback_function,
                      f3 callback_function )
    DISPLAY f1(100,200) + f2(200,50) + f3(150,300)
END FUNCTION
```
Related concepts

Flow control on page 377
Definition of language elements and instructions that control the flow of a program.

FUNCTION *func-spec*

The FUNCTION keyword provides the reference to the specified function.

**Syntax**

```
FUNCTION [module-name.]function-name
```

1. *module-name* is the name of an imported module.
2. *function-name* is the name of a function defining in the current module or in an imported module.

**Usage**

Inside a MAIN or FUNCTION block, the FUNCTION keyword instructs the compiler to use the next symbol as the name of a function, rather than the name of a variable (the language allows the declaration of variables and functions with the same name in the same module).

The function specification following the FUNCTION keyword can be a single function name or a function name prefixed by a module name.

**Note:** The return-types of the referenced functions must be known: If the referenced function returns one or more values, the function type and the function itself must be defined with a RETURNS clause. Otherwise, the compiler will produce the error -8419.

A FUNCTION *func-spec* expression is typically used to assign a variable defined with a TYPE referencing a function. It can also be used as parameter in a function call, but it cannot be combined with other expressions.

**Important:** In order to assign a function reference to a variable, the variable must have been defined with a function type that matches the referenced function: The function parameter names and types, as well as the return types must be the same. If the signatures of the function type and function reference do not match, the compiler produces the error -6631.

**Example**

```plaintext
IMPORT FGL mymodule

TYPE callback_function FUNCTION(p1 INT, p2 INT) RETURNS INT
FUNCTION add(p1 INT, p2 INT) RETURNS INT
  RETURN p1 + p2
END FUNCTION

... DEFINE v callback_function
  LET v = FUNCTION add -- Assign function reference to the variable
...
  CALL process( FUNCTION add, ... ) -- Function reference passed as parameter
...
  LET v = FUNCTION mymodule.sub -- Using a module prefix
```

Related concepts

Function references on page 449
Function can be referenced and invoked dynamically in a CALL instruction, or in an expression.

Methods

A function declared with a receiver type defines a method for this type.

Purpose of methods

Methods are functions that perform on a variable with a specific user-defined type, and cannot be used for another type.

Use methods to implement the interface (the access methods) for a type.

Note: The concept of methods allows you to write robust code like in Object-Oriented Programming languages, without the complexity and traps of OOP.

Defining a method

A method is a FUNCTION defined with a receiver argument specified in parentheses before the function name.

The receiver consists of an identifier followed by a user-defined type. The receiver identifier is then referenced in the function body as the target of this method:

```plaintext
TYPE Rectangle RECORD
  height, width FLOAT
END RECORD

FUNCTION (r Rectangle) area() RETURNS FLOAT
  RETURN r.height * r.width
END FUNCTION
```

Method definition rules

The receiver type must be a TYPE defined in the same module as the method, and it must define a structured RECORD type (it cannot be a flat type defined with a primitive type such as INTEGER).

Important: Receiver types and methods for this type must be defined in the same module. If the receiver type is not defined in the same module as the method, the compiler produces the error -8426.

Method names and receiver type field names must be different: It is not legal to define a method with the same name as a field of the receiver type:

```plaintext
PUBLIC TYPE Rectangle RECORD
  height, width FLOAT
END RECORD
...
PUBLIC FUNCTION (r Rectangle) width () RETURNS INTEGER
  Method and field names must be different. Type rectangle.Rectangle has field width.
  See error number -8427.
  RETURN r.width
END FUNCTION
```

The compiler is more strict regarding methods definitions, compared to regular functions. For example, when a method returns values, it must be defined with the RETURNS clause:

```plaintext
PUBLIC TYPE Rectangle RECORD
  height, width FLOAT
END RECORD
...
PUBLIC FUNCTION (r Rectangle) getWidth ()
  RETURN r.width
```
A method without return type (RETURNS) cannot return values.
See error number -8431.
END FUNCTION

**Invoking a method**

A method for type needs a variable defined to reference an instance of that type. To invoke a method, prefix the method name with the variable of the type used as receiver for the method:

```plaintext
DEFINE r1, r2 Rectangle
DISPLAY r1.area()
DISPLAY r2.area()
```

**Polymorphism**

The same function name can be reused for a different receiver type. Using a different receiver type for the same function name implies another method:

```plaintext
FUNCTION (r Rectangle) area() RETURNS FLOAT
    RETURN r.height * r.width
END FUNCTION

FUNCTION (r Triangle) area() RETURNS FLOAT
    RETURN ( r.height * r.base ) / 2
END FUNCTION
```

**Receiver variable is passed by reference**

A method for a type can implicitly modify the receiver (the receiver variable is passed by reference to the method):

```plaintext
FUNCTION (r Rectangle) setDimensions(w FLOAT, h FLOAT)
    CALL myAssert( (w>0 AND h>0), "Invalid dimensions!" ) -- stops program
    LET r.width = w
    LET r.height = h
END FUNCTION
```

**Methods with record parameters**

When defining a method taking a structured record type as parameter, you must call the method with a record variable name. The record will be passed as a copy: The original record cannot be modified, unless the INOUT clause is used.

**Note:** The . * dot star notation (expanding the record fields on the stack) is not allowed when calling a method with a record as parameter. This technique is supported with regular functions for backward compatibility.

```plaintext
TYPE t_rec1 RECORD
    f1 INT
END RECORD

TYPE t_rec2 RECORD
    f1 INT
END RECORD

FUNCTION (r t_rec1) method1(p t_rec2)
    LET p.f1 = 999
END FUNCTION

FUNCTION (r t_rec1) method2(p t_rec2 INOUT)
    LET p.f1 = 999
END FUNCTION
```
FUNCTION main()
    DEFINE r1 t_rec1, r2 t_rec2
    CALL r1.method1( r2 )
    DISPLAY r2.f1 -- not modified
    CALL r1.method2( r2 )
    DISPLAY r2.f1 -- 999
    -- CALL r1.method1( r2.* ) -- invalid!
END FUNCTION

Methods returning a record structure

Methods can return a complete RECORD: The method must be defined with RETURNS type-name and use the RETURN clause by specifying the record variable without the .* notation (this would expand all members of the record on the stack):

```plaintext
TYPE t_cust RECORD
    id INT,
    name VARCHAR(50)
END RECORD

TYPE t_status RECORD
    errcode INT,
    message STRING
END RECORD

FUNCTION (r t_cust) checkData() RETURNS t_status
    DEFINE s t_status
    CASE
        WHEN r.id IS NULL OR r.id < 1
            LET s.errcode = -9
            LET s.message = "Invalid identifier"
        WHEN LENGTH(r.name) = 0
            LET s.errcode = -8
            LET s.message = "Name is empty"
    END CASE
    RETURN s
END FUNCTION
```

FUNCTION main()
    DEFINE r t_cust
    DEFINE s t_status
    LET s = r.checkData()
    DISPLAY s.*
    LET r.id = 101
    LET s = r.checkData()
    DISPLAY s.*
    LET r.name = "John Lambert"
    LET s = r.checkData()
    DISPLAY s.*
END FUNCTION

Related concepts

Runtime stack on page 493
The runtime stack is used to pass/return values to/from functions.

**Function attributes**

Function attributes can be used to add definition information about the function, its parameters and its return values.

**Purpose of function attributes**

Function attributes can be used to define properties for the function itself, for the parameters accepted by the function, and for the returned values/types.

Function attributes are typically used to define RESTful Web services with high-level framework on page 3375.

**Usage**

Function, parameter and return value attributes are specified with the `ATTRIBUTES()` clause. The `ATTRIBUTES()` clause contains a comma-separated list of attributes. An attribute can be a `name=value` pair, or single `name` for a boolean attribute:

```
ATTRIBUTES( WSScope='book.write', WSPUT, WSPut='{id}', ... )
```

The `ATTRIBUTES()` clause can be specified:

- after a type of parameter in the ( ) parameter list, to define parameter attributes.
- after the closing brace of the parameter list, to define function attributes.
- after a type in the `RETURNS` clause, to define return value attributes.

**Example**

```plaintext
TYPE SimpleBookType RECORD ... END RECORD
TYPE BookType RECORD ... END RECORD

PUBLIC FUNCTION UpdateBookById(
    id INTEGER ATTRIBUTES( WSParam ),
    b SimpleBookType ATTRIBUTES( WSMedia='application/json' )
) ATTRIBUTES( WSScope='book.write, book.read',
    WSPUT,
    WSPut='/{id}',
    WSDescription="Update book title and author for given id",
    WSThrows="400:Invalid,404:NotAvailable" )

RETURNS ( BookType ATTRIBUTES( WSMedia='application/json,application/xml' )
)

DEFINE ret BookType
# ... code to update the book ...
RETURN ret.*
END FUNCTION
```

**Related concepts**

REST on page 3274

Representational State Transfer (REST) is a Web standard architecture that provides a method for communication between a Web service and a client over HTTP.

**Related reference**

High-level RESTful Web service attributes on page 3463
Examples

Function definition examples.

Example 1: Function fetching customer number
The example shows a typical function definition returning an INTEGER.

FUNCTION findCustomerNumber(name VARCHAR(50)) RETURNS INTEGER
DEFINE num INTEGER
CONSTANT sqltxt = "SELECT cust_num FROM customer WHERE cust_name = ?"
PREPARE stmt FROM sqltxt
EXECUTE stmt INTO num USING name
IF SQLCA.SQLCODE = 100 THEN
  LET num = -1
END IF
RETURN num
END FUNCTION

Example 2: Private function definition
This example shows how to define a private function.
This function will not be visible to other modules because it is declared as PRIVATE for the module:

PRIVATE FUNCTION checkLength( name VARCHAR(50) )
DEFINE ok BOOLEAN
IF length(name) == 0 THEN
  LET ok = FALSE
ELSE
  LET ok = TRUE
END IF
RETURN ok
END FUNCTION

Example 3: Using function references
This example shows how to invoke functions dynamically by reference.
The module "compute.4gl":

PUBLIC TYPE compute_function FUNCTION(p1 INT, p2 INT) RETURNS INT
PRIVATE DEFINE cf compute_function
PUBLIC FUNCTION set_function( f compute_function )
  LET cf = f
END FUNCTION
PUBLIC FUNCTION compute(i1 INT, i2 INT) RETURNS INT
  DEFINE r INT
  IF cf IS NULL THEN
    DISPLAY "ERROR: Define the function with set_function(FUNCTION <name>)"
    EXIT PROGRAM 1
  END IF
  LET r = cf(i1, i2)
  DISPLAY SFMT("compute(%1, %2) = %3", i1, i2, r)
  RETURN r
END FUNCTION
PUBLIC FUNCTION add(p1 INT, p2 INT) RETURNS INT
  RETURN p1 + p2
END FUNCTION
PUBLIC FUNCTION sub(p1 INT, p2 INT) RETURNS INT
    RETURN p1 - p2
END FUNCTION

The main program:

IMPORT FGL compute

MAIN
    DEFINE r INT
    CALL compute.set_function(FUNCTION compute.add)
    LET r = compute(1, 2)
    CALL compute.set_function(FUNCTION compute.sub)
    LET r = compute(1, 2)
END MAIN

Example 4: Defining methods

This example shows how to define methods for a Rectangle type:

PUBLIC TYPE Rectangle RECORD
    height, width FLOAT
END RECORD

PUBLIC FUNCTION (r Rectangle) area () RETURNS FLOAT
    RETURN r.height * r.width
END FUNCTION

PUBLIC FUNCTION (r Rectangle) kind () RETURNS STRING
    RETURN "Rectangle"
END FUNCTION

PUBLIC FUNCTION (r Rectangle) setDimensions (w FLOAT, h FLOAT) RETURNS ()
    LET r.width = w
    LET r.height = h
END FUNCTION

Interfaces

An interface groups a set of methods acting on a user-defined type.

Understanding interfaces

This is an introduction to interfaces.

Interfaces are a way to achieve polymorphism in Genero BDL.

While user-defined types describe the data structure and storage, interfaces describe the behavior of an object (type), by declaring the list of operations (methods) that can be done on associated types.

The code using a variable declared as an interface can then act on different typed variables for which these methods exist.

INTERFACE

An interface is defined by a list of methods for a type.

Syntax

INTERFACE
### Language basics

1. **method-name** defines the name of a method.
2. **parameter-name** is the name of a formal argument of the method.
3. **data-type** can be a primitive data type, a user defined type, a built-in class, an imported package class, or a Java class.

### Usage

An **INTERFACE** structure defines a list of methods that apply to types.

The interface defines the how, the type defines the what.

An interface is associated to a type through the list of methods defined for that type.

All elements inside an **INTERFACE** must be and can only be methods for a user-defined type, and must be specified in the interface by using the same parameter names, parameter types and return types of the methods it refers to.

An interface is typically defined as a type to simplify its reusage:

```plaintext
TYPE Shape INTERFACE
  area() RETURNS FLOAT,
  kind() RETURNS STRING
END INTERFACE
```

Methods of multiple individual types associated to an interface can be invoked indirectly by declaring a variable with the interface structure:

```plaintext
DEFINE s Shape
CALL s.area() -- Can be the area() method for types Circle, Rectangle, etc.
```

Several interfaces can be defined for a given type. This provides a high level of flexibility:

```plaintext
TYPE Shape INTERFACE
  area() RETURNS FLOAT
END INTERFACE

TYPE Domain INTERFACE
  domainName() RETURNS STRING
END INTERFACE
```

```plaintext
... DEFINE r Rectangle = ( height:10, width:20 ) DEFINE v1 Shape DEFINE v2 Domain

LET v1 = r DISPLAY v1.area()

LET v2 = r DISPLAY v2.domainName()
```
Related concepts
Variables on page 390
Explains how to define program variables.

INTERFACE usage

Defining interfaces
An interface is defined by a group of methods that apply on user-defined types, to define a common usage interface for several individual types.

A method declared in an interface must use the same parameter names, parameter types and return types as the method implementation it refers to.

An interface must be defined as a type with the TYPE declaration:

```plaintext
TYPE Shape INTERFACE
    kind () RETURNS STRING,
    area () RETURNS FLOAT
END INTERFACE
```

A variable defined with an interface type can receive any type related to the interface.

Associating types to an interface
A user-defined type for which methods are defined is implicitly associated to any interface that defines a set of methods for this type.

For example, a type Rectangle gets associated to a method named area():

```plaintext
TYPE Rectangle RECORD
    height, width FLOAT
END RECORD

FUNCTION (r Rectangle) area () RETURNS FLOAT
    RETURN r.height * r.width
END FUNCTION
```

An interface defined with the area() method becomes implicitly an interface for the type Rectangle:

```plaintext
TYPE Shape INTERFACE
    area() RETURNS FLOAT
END INTERFACE
```

If you define a variable v with an INTERFACE structure listing the area() method, you can assign a variable defined as Rectangle to v, and invoke the method with v.area():

```plaintext
FUNCTION main()
    DEFINE r Rectangle = ( height:10, width:20 )
    DEFINE v Shape
    LET v = r
    DISPLAY v.area()
END FUNCTION
```

Implementation tips
For maximum flexibility, consider implementing the types and corresponding methods in individual modules, and implement the interface in another individual module.

In the parent module using the types/methods and interfaces, import each module with the IMPORT FGL instruction.
Related concepts

Methods on page 452

A function declared with a receiver type defines a method for this type.

Examples

Interface usage examples.

Example 1: Simple INTERFACE usage

Defines an INTERFACE to handle shape objects.

This first module called rectangle.4gl defines the Rectangle type and the methods that apply to this type:

```plaintext
PUBLIC TYPE Rectangle RECORD
  height, width FLOAT
END RECORD

PUBLIC FUNCTION (r Rectangle) area() RETURNS FLOAT
  RETURN r.height * r.width
END FUNCTION

PUBLIC FUNCTION (r Rectangle) kind() RETURNS STRING
  RETURN "Rectangle"
END FUNCTION

PUBLIC FUNCTION (r Rectangle) setDimensions(w FLOAT, h FLOAT) RETURNS()
  LET r.width = w
  LET r.height = h
END FUNCTION
```

The circle.4gl module defines the Circle type and methods that apply to this type:

```plaintext
IMPORT util

PUBLIC TYPE Circle RECORD
  diameter FLOAT
END RECORD

PUBLIC FUNCTION (c Circle) area() RETURNS FLOAT
  RETURN util.Math.pi() * (c.diameter / 2) ** 2
END FUNCTION

PUBLIC FUNCTION (c Circle) kind() RETURNS STRING
  RETURN "Circle"
END FUNCTION

PUBLIC FUNCTION (c Circle) setDiameter(d FLOAT) RETURNS()
  LET c.diameter = d
END FUNCTION
```

The "shapes.4gl" module implements the Shape interface, to group the area() and kind() methods, which both apply to the Rectangle and Circle types. The totalArea() functions takes a dynamic array of Shape objects, knowing only about the behavior defined by the Shape interface.

In this example, the module implementing the Shape interface does not need to import the rectangle and circle modules: The interface definition is abstract and is independent from the types definitions: Only the method definitions matters. Other shape types could be implemented, such as Triangle, Trapezoid, without touching the Shapes interface.
Note also that the `Rectangle.setDimensions()` and `Circle.setDiameter()` methods are not part of the Shape interface: These methods are specific to each shape types and therefore cannot be part of a common interface.

```plaintext
PUBLIC TYPE Shape INTERFACE
   area() RETURNS FLOAT,
   kind() RETURNS STRING
END INTERFACE

PUBLIC TYPE ShapesArray DYNAMIC ARRAY OF Shape

PUBLIC FUNCTION totalArea(shapes ShapesArray) RETURNS FLOAT
   DEFINE i INT
   DEFINE area FLOAT
   FOR i = 1 TO shapes.getLength()
      LET area = area + shapes[i].area()
   END FOR
   RETURN area
END FUNCTION

The main module imports the `rectangle`, `circle` and `shapes` modules, defines `rectangle` and `circle` variables, that can be assigned to an dynamic array of `shapes`, that can in turn be used with the abstract interface:

```plaintext
IMPORT FGL rectangle
IMPORT FGL circle
IMPORT FGL shapes

FUNCTION main()
   DEFINE r1 rectangle.Rectangle
   DEFINE c1 circle.Circle
   DEFINE sa shapes.ShapesArray

   CALL r1.setDimensions(10, 20)
   CALL c1.setDiameter(20)

   LET sa[1] = r1
   LET sa[2] = c1

   DISPLAY sa[1].kind(), sa[1].area()
   DISPLAY sa[2].kind(), sa[2].area()
   DISPLAY "Total area:", shapes.totalArea(sa)
END FUNCTION
```
Advanced features

These topics cover advanced features of the Genero Business Development Language

Program modules

Explains program structure basics.

Structure of a program

The structure of a program consists of MAIN and FUNCTION blocks defined in several modules.

The program starts from the MAIN block. From the MAIN block, the code can invoke other blocks of instructions defined as callable routines with FUNCTION / END FUNCTION blocks. The language statements are executed by the runtime system in the order that they appear in the code:

```plaintext
MAIN
  CALL func1()
END MAIN

FUNCTION func1()
  DISPLAY "Hello from func1()!"
END FUNCTION
```

Some instructions can include other instructions. Such instructions are called *compound statements*. Every compound statement of the language supports the END *statement* keyword (where *statement* is the name of the compound statement), to mark the end of the compound statement construct within the source code module. Most compound statements also support the EXIT *statement* keywords, to transfer control of execution to the statement that follows the END *statement* keywords. By definition, every compound statement can contain at least one statement block, a group of one or more consecutive statements. In the syntax diagram of a compound statement, a statement block always includes this element:

```plaintext
MAIN
  INPUT BY NAME rec.*
  ...
  ON ACTION quit
    EXIT INPUT
  END INPUT
END MAIN
```

Related concepts

Structure of a module on page 462
A module defines a set of program elements such as functions, report routines, types, constants and variables.

Importing modules on page 480
Use the IMPORT ... instruction to import BDL, C or Java external modules in the current module.

Structure of a module

A module defines a set of program elements such as functions, report routines, types, constants and variables.

Syntax

The declaration order of elements defined in a program module is constrained. Define module elements in the following way:

```plaintext
[ compiler-options
```
1. **compiler-options** are described in OPTIONS (Compilation) on page 557.
2. **import-statement** imports an external module, see Importing modules on page 480.
3. **schema-statement** defines a database schema for the compilation.
4. **globals-inclusion** includes a globals file.
5. **constant-definition** defines constants.
6. **type-definition** defines user types.
7. **variable-definition** defines variables.
8. **MAIN-block** declares the main block of the program.
9. **dialog-block** declares a declarative dialog.
10. **function-block** declares a function.
11. **report-routine** declares a report routine.

**Usage**

A module defines a set of program elements that can be used by other modules when defined as **PUBLIC**, or to be local to the current module when defined as **PRIVATE**. Program elements are user-defined types, variables, constants, functions, report routines, and declarative dialogs.

A module can import other modules with the **IMPORT FGL** instruction. A module can define functions, reports, module variables, constants and types, as well as declarative dialogs.

Program modules are written as `.4gl` source files and are compiled to `.42m` files. Compiled modules (.42m files) can be linked together to create a program. However, linking is supported for backward compatibility only. The preferred way is to define module dependencies with the **IMPORT FGL** instruction. For better code re-usability, module elements can be shared by each other by qualifying module variables, constants, types, and functions with **PRIVATE** or **PUBLIC** keywords. PUBLIC module elements can be referenced in other modules.

**Example**

```plaintext
OPTIONS SHORT CIRCUIT
IMPORT FGL cust_data
SCHEMA stores

PRIVATE CONSTANT c_title = "Customer data form"
PUBLIC TYPE t_cust RECORD LIKE customer.*
PRIVATE DEFINE cust_arr DYNAMIC ARRAY OF t_cust

MAIN
  ...
END MAIN

DIALOG cust_dlg()
  INPUT BY NAME cust_rec.*
  ...
```
END INPUT
END DIALOG

FUNCTION cust_display()
  ...
END FUNCTION

FUNCTION cust_input()
  ...
END FUNCTION

REPORT cust_rep(row)
  ...
END REPORT

Related concepts
Structure of a procedural DIALOG block on page 1594
Structure of a declarative DIALOG block on page 1655
A declarative DIALOG instruction is made of a single sub-dialog block, with an optional DEFINE clause to declare local variables.
Structure of a program on page 462
The structure of a program consists of MAIN and FUNCTION blocks defined in several modules.

The MAIN block / function
The MAIN block is the starting point of the program.

Syntax 1 (MAIN / END MAIN)

MAIN
  _ local-declaration
  [.....]
  _ instruction
  [.....]
END MAIN

1. local-declaration is a DEFINE, CONSTANT or TYPE instruction.
2. instruction is a language statement.

Syntax 2 (FUNCTION main())

FUNCTION main()
  _ local-declaration
  [.....]
  _ instruction
  [.....]
END FUNCTION

1. local-declaration is a DEFINE, CONSTANT or TYPE instruction.
2. instruction is a language statement.

Usage
A Genero program starts in the MAIN block, to perform the instructions defined in this block.
Important: If a DATABASE instruction was specified (for the compilation DB schema) before the MAIN / END MAIN block, an implicit connection will occur in MAIN. For more details see the SCHEMA on page 468 instruction.

The MAIN block typically consists of:

1. The signal handling instructions DEFER INTERRUPT and DEFER QUIT,
2. The exception handling instruction WHENEVER ERROR CALL,
3. Global runtime configuration settings with the OPTIONS instruction,
4. A database connection with the CONNECT TO or DATABASE instruction,
5. In an interactive program, a call to a function implementing the main dialog instruction controlling the main form.

The MAIN / END MAIN block must appear before any other FUNCTION / END FUNCTION block:

```
IMPORT FGL cust_module
MAIN
  DEFINE uname, upswd STRING
  DEFER INTERRUPT
  DEFER QUIT
  OPTIONS FIELD ORDER FORM, INPUT WRAP,
      SQL INTERRUPT ON, HELP FILE "myhelp"
  CALL get_login() RETURNING uname, upswd
  TRY
    CONNECT TO "stores" USER uname USING upswd
    CATCH
      IF SQLCA.SQLCODE < 0 THEN
        DISPLAY "Error: Could not connect to database."
        EXIT PROGRAM 1
      END IF
  END TRY
  CALL cust_module.customer_input()
END MAIN

FUNCTION show_help()
  DISPLAY "Usage: ...
END FUNCTION

FUNCTION main()
  DEFINE op t_func
  LET op = FUNCTION add
```

FUNCTION main()

The MAIN block can also be defined as a regular function with FUNCTION main() / END FUNCTION.

In fact a MAIN / END MAIN block is equivalent to FUNCTION main() / END FUNCTION (returning no values), except that with a MAIN block, an implicit database connection is performed, if the DATABASE instruction is used before MAIN / END MAIN, to define the compilation database schema (the implicit database connection does not occur, when using the SCHEMA instruction).

The FUNCTION main() is very useful, when the main block must be defined after other functions, for example when using function references where functions must be declared before they are referenced:

```
TYPE t_func FUNCTION (p1 INT, p2 INT) RETURNS INT

FUNCTION add(p1 INT, p2 INT) RETURNS INT
    RETURN p1 + p2
END FUNCTION

FUNCTION sub(p1 INT, p2 INT) RETURNS INT
    RETURN p1 - p2
END FUNCTION

FUNCTION main()
    DEFINE op t_func
    LET op = FUNCTION add
```
Defining MAIN in imported modules

When using IMPORT FGL (that is, when not linking programs), a MAIN block or main() function can be defined in the imported modules.

This allows for example to write unit tests in the same source module.

File math.4gl:

```
MAIN
  DEFINE
    p1 INT = 6,
    p2 INT = 9,
    res INT
  DISPLAY "Unit testing add (", p1, ",", p2, ")"
  LET res = add(p1, p2)
  IF res = 15 THEN
    DISPLAY " PASSED"
  ELSE
    DISPLAY " FAILED"
  END IF
END MAIN

PUBLIC FUNCTION add(p1 INT, p2 INT) RETURNS INT
  RETURN p1 + p2
END FUNCTION
```

File main.4gl:

```
IMPORT FGL math
MAIN
  DISPLAY math.add(5, 4)
END MAIN
```

Compiling and running both modules:

```
$ fglcomp math.4gl
$ fglrun math.42m
Unit testing add (       6 ,           9)
  PASSED
$ fglcomp main.4gl
$ fglrun main.42m
  9
```

Related concepts

Functions on page 436
Describes user defined functions.

Exceptions on page 502
Describes exception (error) handling in the programs.

## Database schema

Defines database table structures with column type information to be reused in program variable definitions.

### Understanding database schemas

*Database schemas* hold the definition of the database tables and columns.

#### Purpose of database schema files

In program sources or form specification files, specify the database schema file with the `SCHEMA` instruction.

When the database schema is defined, it is possible to declare program variables and form fields by referencing the database table or column name.

The program variables and form fields will get the type of the database column, as defined in the schema file.

#### What contain database schema files?

The schema files contain the column data types (*sch* file), data validation rules (*val* file), and console/tty display attributes (*att* file).

**Note:** The *.val* and *.att* files are supported for backward compatibility and are not recommended in new developments.

For more details about schema file content, see Structure of database schema files on page 470.

#### How to declare program variables from column definitions?

Program variables can be defined with the `LIKE` keyword, to get the data type defined in the schema files:

```plaintext
SCHEMA stores
MAIN
    DEFINE rec_cust RECORD LIKE customer.*
    DEFINE name LIKE customer.cust_name
    ...
END MAIN
```

Multiple database schemas can be used by specifying the schema prefix in the `LIKE` clause:

```plaintext
DEFINE rec_cust RECORD LIKE orders:customer.*
DEFINE rec_item RECORD LIKE stock:item.*
```

For more details, see the `DEFINE` on page 390 instruction.

#### Locating database schema files

The `FGLDBPATH` environment variable can be used to define a list of directories where the compiler can find database schema files.

For more details, see `FGLDBPATH` on page 274.

#### When are database column types used to define program variable?

The data types, display attributes, and validation rules are taken from the database schema files during compilation.

**Important:** Make sure that the schema files of the development database correspond to the production database, otherwise the elements defined in the compiled version of your modules and forms will not match the table structures of the production database.
What unit for CHAR/VARCHAR types in .sch file?

When extracting a database schema with fgldbsch, the size of CHAR/VARCHAR columns is read from the database system tables.

When extracting CHAR/VARCHAR column sizes, fgldbsch writes a number of characters (not bytes) to the .sch file.

When compiling sources using DEFINE ... LIKE, the size of CHAR/VARCHAR columns in the .sch file will be interpreted as a number of bytes or characters following the FGL_LENGTH_SEMANTICS environment variable.

For more details, see also Extracting database schemas on page 525.

Optimized compilation with schema files

With large projects, the database schema file can contain thousands of column definitions.

To improve compilation time, the fglcomp compiler will automatically produce a .42d index file from the .sch schema file, in the same directory as the .sch file.

When the .sch file changes, the .42d index file is re-generated.

The .42d file can be safely removed, to cleanup source directories.

Extracting database schemas with fgldbsch

The database schema files are generated with the fgldbsch tool from the system tables of an existing database.

Note: It is strongly recommended that you re-generate the schema files when upgrading to a new compiler version. Bug fixes and new data type support can require schema file changes. If the schema file holds data type codes that are unknown to the current version, the compilers will raise the error -6634.

The fgldbsch must connect to the database server, with a db user allowed to query the database system tables (for example, INFORMATION_SCHEMA in an Oracle® MySQL database).

Note: For some types of databases, the table owner is mandatory to extract schema information. If you do not specify the -ow option in the comment line, fgldbsch will take the -un user name as default. If you do not use the -un/-up options because you are using indirect database connection with FGLPROFILE settings to identify the database user, or if the database user is authenticated by the operating system, the fgldbsch tool will try to identify the current database user after connection and use this name as table owner to extract the schema.

For more details, see Database schema extractor options on page 476.

Related concepts

File extensions on page 3153
This page describes the file extensions used by the language.

Database connections on page 656
Explains how to manage database connections in a program.

Form specification files on page 1237
Form specification files are the source files defining the layout and content of application forms.

SCHEMA

Defines the database schema files to be used for compilation.

Syntax 1: Database schema specification

```
SCHEMA dbname
```

1. `dbname` identifies the database schema file (.sch).
Syntax 2: Database schema and default connection specification

```
[DESCRIBE] DATABASE dbname
```

1. `dbname` identifies the database schema file (.sch).

**Usage**

The `SCHEMA dbname` instruction defines the database schema to be used for compilation, where `dbname` identifies the name of the database schema files (.sch).

The `[DESCRIBE] DATABASE` instruction defines the compilation database schema, and the default connection for the `MAIN` block when the program starts.

**Tip:** Instead of `[DESCRIBE] DATABASE`, use the `SCHEMA` instruction: The `SCHEMA` instruction defines only the compilation database schema and allows to use a database schema name different from the connection database name.

The `dbname` database name must be expressed explicitly; it cannot be a variable as in a `DATABASE` instruction inside a program block.

Use the `SCHEMA` instruction outside any program block, before a variable declaration with `DEFINE LIKE` instructions. `SCHEMA` must precede any program block in each module that includes a `DEFINE ... LIKE` declaration or `INITIALIZE ... LIKE` and `VALIDATE ... LIKE` statements. It must also precede any `DEFINE ... LIKE` declaration of module variables.

Database schema information such as data types for `DEFINE ... LIKE` are taken from the schema files during compilation. Make sure that the database schema file of the development database corresponds to the production database; otherwise the program variables defined in the p-code modules will not match the table structures of the production database.

For backward compatibility with IBM® Informix®, `dbname` can be written with different syntaxes. You can specify an Informix server, or a more complex Informix source with a string like "//server/database". Such database schema specification is not recommended, as it prevents use of database type other than Informix:

```
database | database @ server | "string"
```

When using a simple identifier for the database name, the compiler converts the name to lowercase, before searching the schema file. However, if a double quoted string is used as database name, the name will be used as is to find the schema file.

With the `SCHEMA` instruction, the name of the database schema during development can be different from the name of the database source used at runtime.

**Note:** To handle uppercase characters in the database name you must quote the name: `SCHEMA "myDatabase"

**Example**

```
SCHEMA dev_db -- Compilation database schema
DEFINE rec RECORD LIKE customer.*
MAIN
   DATABASE prod_db -- Runtime database specification
   SELECT * INTO rec.* FROM customer WHERE custno=1
END MAIN
```

**Related concepts**

DATABASE on page 682
Opens a new database connection in unique-session mode.

**Structure of database schema files**

A database schema is composed by three files (.sch, .val, .att)

**Column Definition File (.sch)**

The .sch database schema file contains the data types of database table columns.

**Description**

The data type of program variables or form fields used to hold data of a given database column must match the data type used in the database. The definition of these elements is simplified by centralizing the information in external .sch files, which contain column data types.

The .sch file can be produced by the fgldb sch tool, from the system views of the target database.

In form files, you can directly specify the table and column name in the field definition in the ATTRIBUTES section of forms.

In programs, you can define variables with the data type of a database column by using the LIKE keyword.

As column data types are extracted from the database system tables, you may get different results with different database servers. For example, Informix® provides the DATE data type to store simple dates in year, month, and day format (= DATE FGL type), while Oracle® stores dates as year to second (= DATETIME YEAR TO SECOND FGL type).

The table describes the fields you will find in a row of the .sch file:

**Table 147: Structure of the .sch file**

<table>
<thead>
<tr>
<th>Pos</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>STRING</td>
<td>Database table name.</td>
</tr>
<tr>
<td>2</td>
<td>STRING</td>
<td>Column name.</td>
</tr>
<tr>
<td>3</td>
<td>SMALLINT</td>
<td>Coded column data type. If the column is NOT NULL, you must add 256 to the value.</td>
</tr>
<tr>
<td>4</td>
<td>SMALLINT</td>
<td>Coded data type length.</td>
</tr>
<tr>
<td>5</td>
<td>SMALLINT</td>
<td>Ordinal position of the column in the table.</td>
</tr>
</tbody>
</table>

This table shows the data types and their corresponding type code that can be present in a schema file:

**Table 148: Database Schema file (.sch) data type codes**

<table>
<thead>
<tr>
<th>Data type name</th>
<th>Data type code (field #3)</th>
<th>Data type length (field #4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR</td>
<td>0</td>
<td>Maximum number of characters or bytes (see note)</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>1</td>
<td>Fixed length of 2</td>
</tr>
<tr>
<td>INTEGER</td>
<td>2</td>
<td>Fixed length of 4</td>
</tr>
<tr>
<td>FLOAT / DOUBLE PRECISION</td>
<td>3</td>
<td>Fixed length of 8</td>
</tr>
<tr>
<td>SMALLFLOAT / REAL</td>
<td>4</td>
<td>Fixed length of 4</td>
</tr>
<tr>
<td>Data type name</td>
<td>Data type code (field #3)</td>
<td>Data type length (field #4)</td>
</tr>
<tr>
<td>----------------</td>
<td>--------------------------</td>
<td>-----------------------------</td>
</tr>
</tbody>
</table>
| DECIMAL        | 5                        | If the decimal is defined with a precision and scale, the length is computed using this formula:  
|                |                          | length = (precision * 256) + scale  
|                |                          | If the decimal is defined as a floating point decimal (i.e. with no scale), the length is computed as follows:  
|                |                          | length = (precision * 256) + 255  
| SERIAL         | 6                        | Fixed length of 4            
| DATE           | 7                        | Fixed length of 4            
| MONEY          | 8                        | The length is computed using this formula:  
|                |                          | length = (precision * 256) + scale  
|                |                          | A MONEY cannot be defined with a floating point, is has always a scale.  
| Unused         | 9                        | N/A                         
| DATETIME       | 10                       | For DATETIME types, the length is determined using the formula:  
|                |                          | length = (digits * 256) + (qual1 * 16) + qual2  
|                |                          | where digits is the total number of digits used when displaying the datetime value. For example, a DATETIME YEAR TO MINUTE (YYYY-MM-DD hh:mm) uses 12 digits.  
|                |                          | The qual1 and qual2 elements identify datetime qualifiers using the following codes:  
|                |                          | • 0 = YEAR  
|                |                          | • 2 = MONTH  
|                |                          | • 4 = DAY  
|                |                          | • 6 = HOUR  
|                |                          | • 8 = MINUTE  
|                |                          | • 10 = SECOND  
|                |                          | • 11 = FRACTION(1)  
|                |                          | • 12 = FRACTION(2)  
|                |                          | • 13 = FRACTION(3)  
|                |                          | • 14 = FRACTION(4)  
|                |                          | • 15 = FRACTION(5)  
|                |                          | For example, a DATETIME YEAR TO MINUTE size length is computed as follows:  
|                |                          | (12 * 256) + (0 * 16) + 8 = 3080  
| BYTE           | 11                       | Length of descriptor         
<p>| TEXT           | 12                       | Length of descriptor         |</p>
<table>
<thead>
<tr>
<th>Data type name</th>
<th>Data type code (field #3)</th>
<th>Data type length (field #4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARCHAR</td>
<td>13</td>
<td>Maximum number of characters or bytes (see note)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If the length is positive:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>length = ( min_space * 256 ) + max_size</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If length is negative:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>length + 65536 = ( min_space * 256 ) + max_size</td>
</tr>
<tr>
<td>INTERVAL</td>
<td>14</td>
<td>For INTERVAL types, the length is determined using the following formula:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>length = ( digits * 256 ) + ( qual1 * 16 ) + qual2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>where digits is the total number of digits used when displaying the interval value. For example, a INTERVAL HOUR(5) TO FRACTION(3) (hhhh:mm:ss.fff) uses 12 digits.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The qual1 and qual2 elements identify datetime qualifiers according to this list:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 0 = YEAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 2 = MONTH</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 4 = DAY</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 6 = HOUR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 8 = MINUTE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 10 = SECOND</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 11 = FRACTION(1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 12 = FRACTION(2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 13 = FRACTION(3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 14 = FRACTION(4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 15 = FRACTION(5)</td>
</tr>
<tr>
<td>NCHAR</td>
<td>15</td>
<td>Maximum number of characters or bytes (see note)</td>
</tr>
<tr>
<td>NVARCHAR</td>
<td>16</td>
<td>Maximum number of characters or bytes (see note)</td>
</tr>
<tr>
<td>INT8</td>
<td>17</td>
<td>Fixed length of 10 (size of int8 structure)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>In programs, will be converted to a BIGINT type.</td>
</tr>
<tr>
<td>SERIAL8</td>
<td>18</td>
<td>Fixed length of 10 (size of int8 structure)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>In programs, will be converted to BIGINT type.</td>
</tr>
<tr>
<td>BOOLEAN (SQLBOOL)</td>
<td>45</td>
<td>Boolean type, in the meaning of Informix® front-end SQLBOOL (sqltype.h)</td>
</tr>
<tr>
<td>BIGINT</td>
<td>52</td>
<td>Fixed length of 8 (bytes)</td>
</tr>
<tr>
<td>BIGSERIAL</td>
<td>53</td>
<td>Fixed length of 8 (bytes)</td>
</tr>
<tr>
<td>Data type name</td>
<td>Data type code (field #3)</td>
<td>Data type length (field #4)</td>
</tr>
<tr>
<td>----------------</td>
<td>---------------------------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>VARCHAR2</td>
<td>201</td>
<td>Maximum number of characters</td>
</tr>
<tr>
<td></td>
<td></td>
<td>In programs, will be converted to a VARCHAR type.</td>
</tr>
<tr>
<td>NVARCHAR2</td>
<td>202</td>
<td>Maximum number of characters</td>
</tr>
<tr>
<td></td>
<td></td>
<td>In programs, will be converted to a VARCHAR type.</td>
</tr>
</tbody>
</table>

**Note:** Data type length (field #4) is a SMALLINT value encoding the length or composite length of the type. For character string types, the unit of the length used to define character program variables and form fields depends on the length semantics.

**Informix® SERIAL types**

When the database schema defines SERIAL, BIGSERIAL or SERIAL8 types, form fields referencing the serial column will get the NOENTRY attribute automatically, except if defined with the TYPE LIKE syntax.

**Informix® CHAR/VARCHAR types and SQL_LOGICAL_CHAR**

The Informix® SQL_LOGICAL_CHARS onconfig parameter is taken into account when extracting CHAR/VARCHAR column types, to convert the size in bytes from syscolumns.collength to a number of characters for the .sch file.

It is then possible to use FGL_LENGTH_SEMANTICS=CHAR when the application locale is UTF-8.

**Informix® DISTINCT types**

Informix® IDS version 9.x and higher allow you to define DISTINCT types from a base type with the CREATE DISTINCT TYPE instruction. In the syscolumns table, Informix® identifies distinct types in the coltype column by adding the 0x0800 bit (2048) to the base type code. For example, a distinct type defined with the VARCHAR built-in type (code 13) will be identified with the code 2061 (13 + 2048). Informix® sets additional bits when the distinct type is based on the LVARCHAR or BOOLEAN opaque types. If the base type is an LVARCHAR, the type code used in coltype gets the 0x2000 bit set (8192) and when the base type is BOOLEAN, the type code gets the 0x4000 bit (16384).

When extracting a schema from an Informix® database defining columns with DISTINCT types, the schema extractor will keep the original type code of the distinct type in the .sch file for columns using distinct types based on built-in types (with the 0x0800 bit set). Regarding the exception of opaque types, BOOLEAN-based distinct types get the code 45 (+ 256 if NOT NULL), and LVARCHAR-based distinct types are mapped to the code 201 (+ 256 if NOT NULL) if the -cv option enables conversion from LVARCHAR to VARCHAR2.

The fglcomp and fglform compilers understand the distinct type code bit 0x0800, so you can define program variables with a DEFINE LIKE instruction based on a column that was created with a distinct Informix® type.

**Example**

```
customer^customer_num^258^4^1^
customer^customer_name^256^50^2^
customer^customer_address^0^100^3^
order^order_num^258^4^1^
order^order_custnum^258^4^2^
order^order_date^263^4^3^
order^order_total^261^1538^4^```
Column Validation File (.val)
The .val database schema file holds functional and display attributes of database table columns.

Description
The .val file holds default attributes and validation rules for database columns.

Important: The form field attribute definition in the .val file is supported for backward compatibility. Do not use this feature in new developments.

In form files, the attributes are taken from the .val file as defaults if the corresponding attribute is not explicitly specified in the field definition of the ATTRIBUTES section. The attributes in the .val file can be considered as a default configuration for a form field.

In programs, you can validate variable values in accordance with the INCLUDE attribute by using the VALIDATE instruction.

The .val file can be generated by fgldbsch from the IBM® Informix® specific syscolval table, or can be edited by an external column attributes editor.

This table describes the structure of the .val file:

Table 149: Structure of the .val file

<table>
<thead>
<tr>
<th>Pos</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>STRING</td>
<td>Database table name.</td>
</tr>
<tr>
<td>2</td>
<td>STRING</td>
<td>Column name.</td>
</tr>
<tr>
<td>3</td>
<td>STRING</td>
<td>Column property name.</td>
</tr>
<tr>
<td>4</td>
<td>STRING</td>
<td>Column property value.</td>
</tr>
</tbody>
</table>

The supported attribute definitions are:

Table 150: Supported attribute definitions of the .val file

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTONEXT</td>
<td>Defines the AUTONEXT attribute.</td>
</tr>
<tr>
<td></td>
<td>When this attribute is defined, value is YES.</td>
</tr>
<tr>
<td>CENTURY</td>
<td>Defines the CENTURY attribute.</td>
</tr>
<tr>
<td></td>
<td>The value must be one of: R, C, F, or P.</td>
</tr>
<tr>
<td>COLOR</td>
<td>Defines the COLOR attribute.</td>
</tr>
<tr>
<td></td>
<td>The value is a color identifier (RED, GREEN, BLUE, ...)</td>
</tr>
<tr>
<td>COMMENTS</td>
<td>Defines the COMMENTS attribute.</td>
</tr>
<tr>
<td></td>
<td>The value is a quoted string or Localized String (&quot;xxx&quot;).</td>
</tr>
<tr>
<td>DEFAULT</td>
<td>Defines the DEFAULT attribute.</td>
</tr>
<tr>
<td></td>
<td>Number, quoted string or identifier (TODAY).</td>
</tr>
<tr>
<td>FORMAT</td>
<td>Defines the FORMAT attribute.</td>
</tr>
<tr>
<td></td>
<td>The value is a quoted string.</td>
</tr>
<tr>
<td>Attribute Name</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>INCLUDE</td>
<td>Defines an include list as the <code>INCLUDE</code> attribute. Value must be a list: <code>(value [, ...])</code>, where <code>value</code> can be a number, quoted string or identifier (<code>TODAY</code>).</td>
</tr>
<tr>
<td>INVISIBLE</td>
<td>Defines the <code>INVISIBLE</code> attribute. When this attribute is defined, value is <code>YES</code>.</td>
</tr>
<tr>
<td>JUSTIFY</td>
<td>Defines the <code>JUSTIFY</code> attribute. The value must be one of: <code>LEFT</code>, <code>CENTER</code> or <code>RIGHT</code>.</td>
</tr>
<tr>
<td>PICTURE</td>
<td>Defines the <code>PICTURE</code> attribute. The value is a quoted string.</td>
</tr>
<tr>
<td>SHIFT</td>
<td>Corresponds to the <code>UPSHIFT</code> and <code>DOWNSHIFT</code> attributes. Values can be <code>UP</code> or <code>DOWN</code>.</td>
</tr>
<tr>
<td>VERIFY</td>
<td>Defines the <code>VERIFY</code> attribute. When this attribute is defined, value is <code>YES</code>.</td>
</tr>
</tbody>
</table>

**Example**

customer^customer_name^SHIFT^UP^
customer^customer_name^COMMENTS^"Name of the customer"^ 
order^order_date^DEFAULT^TODAY^ 
order^order_date^COMMENTS^"Creation date of the order"^ 

**Column Video Attributes File (.att)**
The `.att` database schema file contains the default video attributes of database table columns.

**Description:**
The `.att` file is generated by `fgldbsch` from the IBM® Informix® specific `syscolatt` table.

**Important:** The form field video attributes definition in the `.att` file is supported for backward compatibility. Do not use this feature in new developments.

This table describes the structure of the `.att` file:

**Table 151: Structure of the .att file**

<table>
<thead>
<tr>
<th>Pos</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>STRING</td>
<td>Database table name.</td>
</tr>
<tr>
<td>2</td>
<td>STRING</td>
<td>Column name.</td>
</tr>
<tr>
<td>3</td>
<td>SMALLINT</td>
<td>Ordinal number of the attribute record.</td>
</tr>
<tr>
<td>4</td>
<td>STRING</td>
<td>COLOR attribute (coded).</td>
</tr>
<tr>
<td>5</td>
<td>CHAR (1)</td>
<td>INVERSE attribute (y/n).</td>
</tr>
<tr>
<td>6</td>
<td>CHAR (1)</td>
<td>UNDERLINE attribute (y/n).</td>
</tr>
<tr>
<td>7</td>
<td>CHAR (1)</td>
<td>BLINK attribute (y/n).</td>
</tr>
</tbody>
</table>
### Related concepts

**Form fields** on page 1242

*Form fields* are form elements designed for data input and/or data display.

### Database schema extractor options

The `fgldbsch` tool extracts the schema description for an existing database.

Schema information is extracted from the database catalog tables. The `fgldbsch` tool detects the type of database server after connection and queries the appropriated system catalog tables.

The database system must be available and the database client environment must be set properly in order to connect to the database engine and generate the schema files.

Generate the database schema files in the directory where the source code resides or in one of the directories listed in the `FGLDBPATH` environment variable.

**Related concepts**

**FGLDBPATH** on page 274

Defines the path to database schema files for compilers.

### Specifying the database source

The `-db dbname` option must be used to define the database source to which to connect.

The `dbname` and related database connection parameters can be present in the `FGLPROFILE` file. Otherwise, related options have to be provided with the `fgldbsch` command such as `-dv` for the driver.

```
fgldbsch -db test1
```

**Related concepts**

**Database source specification (source)** on page 661

**Database schema extractor options** on page 476

The `fgldbsch` tool extracts the schema description for an existing database.

### Specifying the database driver

The database driver can be specified with the `-dv dbdriver` option, if the default driver is not appropriate.

```
fgldbsch -db test1 -dv dbmora
```

**Related concepts**

**Database driver specification (driver)** on page 662

**fgldbsch** on page 2079

The `fgldbsch` tool generates the database schema files from an existing database.

### Passing database user login and password

The database user name and password can be specified with the `-un username` and `-up password` options.

```
fgldbsch -db test1 -un scott -up fourjs
```
Related concepts
User name and password (username/password) on page 664
fgldbsch on page 2079
The fgldbsch tool generates the database schema files from an existing database.

Data type conversion control

The fglcomp and fglform compilers expect known language data types (FGL types) in the schema file. While most data types correspond to IBM® Informix® SQL data types, some databases (including Informix®) can use specific types that do not map to an FGL type. Therefore, data types in the schema file are generated from the system catalog tables based on some conversion rules.

Type conversion can be controlled with the -cv option. Each character position of the string passed by this option represents a line in the conversion table of the corresponding source database. Give a conversion code for each data type (for example: -cv AABAAAB).

When using X as conversion code, the columns using the corresponding data types will be ignored and not written to the .sch file. This is particularly useful in the case of auto-generated columns like SQL Server's uniqueidentifier data type, when using a DEFAULT NEWID() clause.

Run the tool with the -ct option to see all the data type conversion tables, or use the -cx dbtype option to display the conversion table for a given database type (dbtype must be ifx, ora, db2, msv, pgs, mys, ...).

```
fgldbsch -cx ifx
```

```
<table>
<thead>
<tr>
<th>Informix</th>
<th>Informix A</th>
<th>Informix B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 BOOLEAN</td>
<td>BOOLEAN (t=45)</td>
<td>CHAR (1)</td>
</tr>
<tr>
<td>2 INT8</td>
<td>INT8</td>
<td>DECIMAL (19,0)</td>
</tr>
<tr>
<td>3 SERIAL8</td>
<td>SERIAL8</td>
<td>DECIMAL (19,0)</td>
</tr>
<tr>
<td>4 LVARCHAR(m)</td>
<td>VARCHAR2 (m)</td>
<td>VARCHAR2 (m)</td>
</tr>
<tr>
<td>5 BIGINT</td>
<td>BIGINT</td>
<td>DECIMAL (19,0)</td>
</tr>
<tr>
<td>6 BIGSERIAL</td>
<td>BIGSERIAL</td>
<td>DECIMAL (19,0)</td>
</tr>
</tbody>
</table>

(ns) = Not supported in 4gl.
```

In the above example, the -cv option instructs fgldbsch to use the types of the "Informix® A" column for all original column types except for BOOLEAN, BIGINT and BIGSERIAL, which must be converted to a VARCHAR2 (m) FGL type.

The IBM® Informix® LVARCHAR(m) type can be converted by default to a VARCHAR2 (m) pseudo type (code 201), which will be identified as a VARCHAR(m) by compilers.

In schema files, VARCHAR2 (m) (type code 201) is equivalent to VARCHAR (m) (type code 13), without the 255 bytes limitation of the original Informix® VARCHAR type.

Not all native data types can be converted to FGL types. For example, user-defined types or spatial types are not supported by the language. When a table column with such unsupported data type is found, fgldbsch stops and displays an error to bring the problem to your attention. Use the -ie option of fgldbsch to ignore the database tables having columns with unsupported types. When this option is used, none of the table columns definition will be written to the schema file.

Related concepts
Primitive Data types on page 289
Selecting the correct data type assists you in the input, storage, and display of your data.

fgldbsch on page 2079
The `fgldbsch` tool generates the database schema files from an existing database.

**Column Definition File (.sch) on page 470**
The `.sch` database schema file contains the data types of database table columns.

**Skip unsupported table definitions**
By default `fgldbsch` stops with an error, if a database table column is defined with an SQL data type that cannot be supported by Genero BDL.

Use the `-ie` option, to skip such tables from the schema extraction.

**Note:** When using the `-ie` option, the whole table definition is skipped when a column type is not supported.
Otherwise, the `.sch` file would contain a subset of the column definitions of the table, and a `SELECT * INTO record-defined-like.* FROM tabname` would fail because the number of columns in the table and into the record definition would not match.

```
fgldbsch -db test1 -ie
```

**Related concepts**
- `fgldbsch` on page 2079
The `fgldbsch` tool generates the database schema files from an existing database.

**Specifying the table owner**
With some databases, the owner of tables is mandatory to extract a schema, otherwise you could get multiple definitions of the same table in the `.sch` schema file, if tables with the same name exist in different database user schemas.

To prevent such duplicates, specify the schema owner with the `-ow owner` option. If this option is not used, `fgldbsch` will use the database login name passed with the `-un username` option. This is usually the case with SQL Server and SAP ASE, where the owner of tables is "dbo".

```
fgldbsch -db test1 -un scott -up fourjs -ow dbo
```

**Related concepts**
- `Database connections` on page 656
Explains how to manage database connections in a program.
- `fgldbsch` on page 2079
The `fgldbsch` tool generates the database schema files from an existing database.

**Force extraction of system tables**
By default `fgldbsch` does not extract the definition of database system tables.

Use the `-st` option to extract schema information of system tables.

```
fgldbsch -db test1 -st
```

**Related concepts**
- `fgldbsch` on page 2079
The `fgldbsch` tool generates the database schema files from an existing database.

**Specifying the output file name**
By default, the generated schema files get the name of the database source specified with the `-db` option.

The name of the schema file can be forced with the `-of filename` option.

Specify the output file name without the `.sch` extension.
Note: The file name specified with the –of option will also be used to generate the files containing column validation rules and column attributes (extracted from IBM® Informix® syscolval and syscolatt tables).

```
fgldbsch -db test1 -of myschema
```

**Related concepts**

**Column Validation File (.val)** on page 474  
The .val database schema file holds functional and display attributes of database table columns.

**Column Video Attributes File (.att)** on page 475  
The .att database schema file contains the default video attributes of database table columns.

**fgldbsch** on page 2079  
The fgldbsch tool generates the database schema files from an existing database.

### Extracting definition of a single table

In some cases, you may just want to extract schema file of new created tables.

Use the –tn tabname option, to extract schema information of a specific table.

```
fgldbsch -db test1 -tn customers
```

**Related concepts**

**fgldbsch** on page 2079  
The fgldbsch tool generates the database schema files from an existing database.

### Controlling the character case

By default, table and column names are converted to lower case letters to enforce compatibility with IBM® Informix®.

Force lower case, upper case or case-sensitive table and column names by using the –cl, –cu or –cc options.

```
fgldbsch -db test1 –cc
```

As a general rule, it is strongly recommended to keep table and column names in lowercase, in all areas (including the objects created in the database entity).

**Related concepts**

**fgldbsch** on page 2079  
The fgldbsch tool generates the database schema files from an existing database.

### Using the verbose mode

By default, fgldbsch extracts the database schema silently without any output.

Use the –v option to get verbose output from fgldbsch:

```
fgldbsch -db test1 –v
```

**Important:** Do not base other tools or development procedures on the output format of the fgldbsch –v option: The output can change in later versions.

**Related concepts**

**fgldbsch** on page 2079  
The fgldbsch tool generates the database schema files from an existing database.

### IBM® Informix® synonym tables

When using IBM® Informix®, fgldbsch extracts synonyms by default. Only PUBLIC synonyms are extracted, to avoid duplicates in the .sch file, when the same name is used by several synonyms for different table owners.
To extract PRIVATE synonyms, use the `-ow` option to specify the owner of the tables and synonyms.

```
fgldbsch -db test1 -ow mike
```

**Related concepts**

fgldbsch on page 2079

The fgldbsch tool generates the database schema files from an existing database.

**IBM® Informix® shadow columns**

Starting with IBM® Informix® IDS version 11.50.xC1, you can create shadow columns on tables by using DDL options such as ADD VERCOLS.

Shadow columns are visible in the system catalog tables and would be listed in the column descriptions of the `.sch` schema file. However, since shadow columns are not part of the `SELECT *` list, it is not expected to get these columns in the `.sch` schema file.

By default, the fgldbsch on page 2079 tool will not extract shadow columns from an IBM® Informix® database.

Use the `-sc` option to force the extraction of shadow columns:

```
fgldbsch -db test1 -sc
```

**Running schema extractor in old mode**

The fgldbsch program can be executed in old mode by specifying the `-om` option as first parameter, followed by the database source. You can pass the `-c` and `-r` options after the database source:

```
fgldbsch -om test1 -c -r
```

**Important:** Use the fgldbsch `-om` mode for IBM® Informix® databases only.

The `-c` option is equivalent to `-cv` `BBBBBBBBBB` in the default mode: Columns defined with an SQL type that is not a native Genero type will be converted to an equivalent type (see `-cv` and `-ct` options for more details).

If the `-r` option is specified, the schema extractor will ignore columns defined with unsupported SQL types. Unsupported types have no equivalent FGL type to store and handle the value, such as BLOB or CLOB for example. Understand that unlike the `-ie` option, which skips the whole table definition, `-r` will exclude table columns with unsupported types, but the other columns defined with supported types will be written to the `.sch` file. Thus, a record declared with `DEFINE RECORD rec LIKE table.*` (from a partial schema definition of a table) cannot be used in a `SELECT * INTO rec.*` statement, because the number of columns in the database table is different from the record definition.

**Note:** When using the fgldbsch `-om` mode, fgldbsch will extract system catalog tables (`informix.sys*`) for IBM® Informix® databases.

**Related concepts**

fgldbsch on page 2079

The fgldbsch tool generates the database schema files from an existing database.

**Importing modules**

Use the `IMPORT ...` instruction to import BDL, C or Java external modules in the current module.

The `IMPORT __JAVA__FGL__` instruction can be used to declare the usage of an external module. All (public) symbols of the external module can be referenced in the current module.

**Note:** The `IMPORT __JAVA__FGL__` instruction must be the first instruction in the current module. If you specify this instruction after `DEFINE`, `CONSTANT` or `GLOBALS`, fglcomp will report a syntax error.

The `IMPORT __JAVA__FGL__` instruction can import a compiled Genero module, a Java class or a C extension library:
• **IMPORT FGL modulename**: Imports a Genero module implementing functions, reports, types and variables.
• **IMPORT JAVA classname**: Imports a Java class or class element.
• **IMPORT libname**: Imports a C extension implementing functions and variables.

The name specified after the **IMPORT FGL** or **IMPORT JAVA** instruction is case-sensitive; program module (.4gl) or Java class must exactly match the file name.

**Note**: For backward compatibility, C extension library names specified after IMPORT are converted to lowercase by the compiler (therefore, we recommend you to use lowercase file names for C extensions). A character case mismatch will be detected on UNIX™ platforms, but not on Windows® where the file system is not case-sensitive. Regarding the usage of imported symbols in the rest of the code (not in the IMPORT instruction): C extensions and Genero symbols are case-insensitive, while Java symbols are case-sensitive.

**IMPORT C-Extension**

The IMPORT instruction imports C extension module elements to be used by the current module.

**Syntax**

```
IMPORT cextname
```

1. `cextname` is an identifier defining the C extension module to be imported (without the file extension).

**Usage**

Using IMPORT cextname instructs the compiler and runtime system to use the `cextname` C extension for the current module.

**Important**: At runtime, all imported C extension modules are loaded when the program starts.

The name of the module specified after the IMPORT keyword is converted to lowercase by the compiler. Therefore it is recommended to use lowercase file names only.

The C extension must exist as a shared library (.DLL or .so) and be loadable (environment variables must be set properly). C extension modules used with the IMPORT instruction do not have to be linked to fglrun: the runtime system loads dependent C extension modules dynamically.

The FGLLDPATH environment variable specifies the directories to search for the C extension modules. You may also have to set up the system environment properly (such as PATH on Windows® and LD_LIBRARY_PATH on UNIX™) if the C extension library is dependent on other libraries.

By default, the runtime system tries to load a C extension module with the name userextension, if it exists. This simplifies the migration of existing C extensions; you just need to create a shared library named userextension.so (or userextension.dll on Windows®), and copy the file to one of the directories defined in FGLLDPATH.

**Related concepts**

C-Extensions on page 2230

With C-Extensions, you can bind your own C libraries in the runtime system, to call C function from the application code.

**IMPORT FGL module**

The IMPORT FGL instruction imports module symbols.

**Syntax**

```
IMPORT FGL modulename
```

1. `modulename` is an identifier defining the module to be imported (without the file extension).
Usage

With `IMPORT FGL modulename`, the symbols of the named .42m module can be referenced in the current module.

**Important:** At runtime, the imported modules are only loaded on demand, when the program flow reaches an instruction that uses an element of the imported module. For example, when calling a function or when assigning a (public) module variable of the imported module.

The name specified after the `IMPORT FGL` instruction is case-sensitive.

The imported module symbols that can be referenced are:

- Public functions
- Public constants
- Public types
- Public module variables

Compilation with `IMPORT FGL`

`IMPORT FGL` instructs the `fglcomp` compiler and `fglrun` runtime system to load/check the specified modules.

When using only `IMPORT FGL` to define module dependency, there is no longer a need to link programs or use libraries.

With `IMPORT FGL`, the compiler can check the number of parameters and returning values in functions calls, and the autocompletion in source code editors is improved as it can suggest all imported symbols.

Auto-compilation of (local) imported modules

It is recommended to compile imported modules before compiling the importing module.

The `FGLLDPATH` environment variable specifies the directories to search for the .42m modules used by `IMPORT FGL`.

When the imported module is located in the same directory as the compiled module, if the .42m file of the imported module does not exist, or is older than the corresponding source file, `fglcomp` will automatically compile the imported module.

The automatic recompilation of imported modules applies recursively: For example, when a `main.4gl` module imports `module1` which in turn imports `module2`, and `module2.42m` is more recent as `module2.4gl`, `fglcomp` will automatically compile `module2.4gl`.

**Important:** Auto-compilation of imported modules is only supported if the imported module is in the current directory. Modules located in other directories and found by `FGLLDPATH` must already be compiled.

To avoid implicit compilation of imported modules, use the `--implicit=none` option of `fglcomp`. If the .42m file exists but the .4gl source file cannot be found, `fglcomp` imports the .42m file as is.

**Note:** The `fglcomp --implicit=none` option is provided for specific cases. Do not use this option when not really needed: Auto-compilation of imported modules is the recommended default.

Circular module references

Circular references between modules are allowed. A circular reference occurs when several modules define an interdependence with `IMPORT FGL`.

Circular module references can be direct or indirect, for example:

1. Direct circular reference: Module A imports module B, which in turn imports module A.
2. Indirect circular reference: Module A imports module B, which imports module C, which imports module A.

**Important:** When .42m files are not yet available, compiling modules with circular dependency is only possible when these modules exist in the same directory and auto-compilation is used. Otherwise, when inter-dependent
modules are located in different directories, or when using the --implicit=none option, all the inter-dependent modules must be specified in the fglcomp command.

Code example: Module "module_a.4gl":

```
IMPORT FGL module_b

MAIN
  CALL function_a1()
END MAIN

FUNCTION function_a1()
  DISPLAY "In module_a: function_a1()"
  CALL module_b.function_b1()
END FUNCTION

FUNCTION function_a2()
  DISPLAY "In module_a: function_a2()"
END FUNCTION
```

Module "module_b.4gl":

```
IMPORT FGL module_a

FUNCTION function_b1()
  DISPLAY "In module_b: function_b1()"
  CALL module_a.function_a2()
END FUNCTION
```

Compile and execute:

```
$ fglcomp module_a.4gl && fglrun module_a
In module_a: function_a1()
In module_b: function_b1()
In module_a: function_a2()
```

Tip: To get detailed information about the compilation process, use the --verbose option of fglcomp:

```
$ fglcomp --verbose module_a.4gl
[loading fglhelp]
[ parsing module_a.4gl]
[ parsing module_b.4gl]
[ building module_b]
[ writing module_b.42m]
[ building module_a]
[ writing module_a.42m]
[ total modules: 4 variables: 6 functions: 274 types: 9 fields: 10]
```

Identifying modules to be imported

When migrating existing projects using traditional linking, after compiling all the .4gl sources, consider using the --print-missing-imports option of fglrun, to print the IMPORT FGL suggestions for all the modules specified in the fglrun command line.

The fglrun command also provides the --print-imports option to identify all imported modules that are really used.

Note: --print-imports and --print-missing-imports are options of fglrun, not fglcomp.

The --print-missing-imports option will try to resolve all (function) symbols as done during linking, but instead of producing a .42r program, it will list the IMPORT FGL instructions to be added in each module, and thus avoid linking.
The **--print-imports** option prints all IMPORT FGL instructions, that are really used by a module, where at least one symbol (function, variable, constant, type, etc) is used by the importing module.

**Note:** Use the **--print-missing-imports** options instead of **--print-imports**: If you are missing an IMPORT FGL for a non-function symbol like a variable, constant, or type, the compiler will produce a compilation error. Regarding function symbols, the compilation is possible even when the module was not imported. Therefore, **--print-missing-imports** is useful to identify modules to be imported to resolve function symbols and avoid linking.

In the next example, mod1 can be imported in main, but mod2 already imports mod1:

```bash
$ head *.4gl
==> main.4gl <==
MAIN
  CALL func1()
END MAIN

==> mod1.4gl <==
FUNCTION func1()
  CALL func2()
END FUNCTION

==> mod2.4gl <==
IMPORT FGL mod1
FUNCTION func2()
  CALL func1()
END FUNCTION

$ fglcomp main.4gl mod1.4gl mod2.4gl

$ fglrun --print-imports main.42m mod1.42m mod2.42m
-- in main.4gl
IMPORT FGL mod1

-- in mod1.4gl
IMPORT FGL mod2

-- in mod2.4gl
IMPORT FGL mod1

$ fglrun --print-missing-imports main.42m mod1.42m mod2.42m
-- in main.4gl
IMPORT FGL mod1

-- in mod1.4gl
IMPORT FGL mod2
```

**Scope of module symbols (PRIVATE/PUBLIC)**

The PRIVATE/PUBLIC modifiers can be used to hide / publish symbols to other modules.

**Note:** Functions are by default public, for backward compatibility. Module variables, types and constants are by default private.

The following example declares a module variable that can be used by other modules, and a private function to be used only locally:

```plaintext
PUBLIC DEFINE custlist DYNAMIC ARRAY OF RECORD
  id INT,
  name VARCHAR(50),
  address VARCHAR(200)
END RECORD
...```
PRIVATE FUNCTION myfunction()
...
END FUNCTION

Resolving symbol name conflicts with module prefix

If a symbol is defined twice with the same name in two different modules, the symbol must be qualified by the name of the module.

This feature overcomes the traditional 4GL limitation, requiring unique function names within a program.

In the following example, both imported modules define the same "init()" function, but this can be resolved, by adding the module name followed by a dot before the function names:

```
IMPORT FGL orders
IMPORT FGL customers
MAIN
  CALL orders.init()
  CALL customers.init()
  ...
END MAIN
```

If a symbol is defined twice with the same name in the current and the imported module, an unqualified symbol will reference the current module symbol.

The following example calls the "init()" function with and without a module qualifier. The second call will reference the local function:

```
IMPORT FGL orders
MAIN
  CALL orders.init()  -- orders module function
  CALL init()  -- local function
  ...
END MAIN
FUNCTION init()
  ...
END FUNCTION
```

Qualifying imported symbols with fglcomp --qualify-imports

The fglcomp compiler provides the --qualify-imports option to automatically add module prefixes to all imported symbols that are not yet qualified.

**Note:** The --qualify-imports option does not compile sources: The imported modules need to be compiled as .42m pcode modules.

The modified source is written to the standard output stream (stdout).

For example, with two modules defining public types, constants, variables and functions:

```
-- module1.4gl
PUBLIC CONSTANT const1 = 500
PUBLIC TYPE type1 INTEGER
PUBLIC FUNCTION func1( p1 type1 )
  DISPLAY "func1:", p1
END FUNCTION

-- module2.4gl
PUBLIC DEFINE var2 STRING
PUBLIC CONSTANT const2 = "abc"
```
Assuming the following importing module:

```plaintext
IMPORT FGL module1
IMPORT FGL module2
MAIN
  DEFINE v type1
  LET v = const1
  CALL func1( v )
  LET var2 = "abc"
  DISPLAY const2
END MAIN
```

The `fglcomp --qualify-imports` will produce this output from the above source file:

```plaintext
IMPORT FGL module1
IMPORT FGL module2
MAIN
  DEFINE v module1.type1
  LET v = module1.const1
  CALL module1.func1( v )
  LET module2.var2 = "abc"
  DISPLAY module2.const2
END MAIN
```

If two modules define the same symbol name, `fglcomp --qualify-imports` will produce the error -8401.

**Mixing IMPORT FGL and .42r linking**

Traditional linking is still supported for backward compatibility. To ease migration from traditional linking to imported modules, you can mix IMPORT FGL usage with `fgllink`.

By default, even when IMPORT FGL is used, fglcomp does not raise an error, if a referenced function is not found in the imported modules. This is mandatory to compile the 42m file to be linked later with the module defining the missing function.

Use the `fglcomp -W implicit` or the `--resolve-calls` options, to check that all symbols are resolved with a corresponding IMPORT FGL instruction.

When the `-W implicit` option is used, fglcomp will print warning -8406 for any referenced function that cannot be found in an imported module.

**Note:** The `-W implicit` option is to be used when migrating linked modules to a solution where module dependency is only based on IMPORT FGL.

To enable strict symbol resolution by the compiler, use the `--resolve-calls` option. This option will force the compiler to check all function symbols referenced in a module, and raise error -8406, if a symbol is not found in the imported modules.

**Note:** The `--resolve-calls` option should be used to compile programs that are only based on IMPORT FGL and no longer use the link phase.

For more details about the linker, see Linking programs on page 2115.

**Example**

Module "account.4gl":

```plaintext
PRIVATE DEFINE current_account VARCHAR(20)

PUBLIC FUNCTION set_account(id)
  DEFINE id VARCHAR(20)
  LET current_account = id
END FUNCTION
```
Module "myutils.4gl":

PRIVATE DEFINE initialized BOOLEAN

PUBLIC TYPE t_prog_info RECORD
   name STRING,
   version STRING,
   author STRING
END RECORD

PUBLIC FUNCTION init()
   LET initialized = TRUE
END FUNCTION

PUBLIC FUNCTION fini()
   LET initialized = FALSE
END FUNCTION

PUBLIC FUNCTION tokenize(s STRING,
                  a DYNAMIC ARRAY OF STRING)
   DEFINE tok base.StringTokenizer,
         x INTEGER
   LET tok = base.StringTokenizer.create(s," \
\n")
   CALL a.clear()
   LET x=0
   WHILE tok.hasMoreTokens()
      LET x=x+1
      LET a[x] = tok.nextToken()
   END WHILE
END FUNCTION

Module "program.4gl":

IMPORT FGL myutils
IMPORT FGL account
DEFINE filename STRING
DEFINE proginfo t_prog_info  -- Type is defined in myutils

MAIN
   DEFINE arr DYNAMIC ARRAY OF STRING
   LET proginfo.name = "program"
   LET proginfo.version = "0.99"
   LET proginfo.author = "scott"
   CALL myutils.init()  -- with module prefix
   CALL set_account("CFX4559")  -- without module prefix
   CALL tokenize("aaa bbb ccc",arr)
   DISPLAY arr[2]
END MAIN

Related concepts
Program execution on page 488
This section describes program execution and language instructions related to program execution.

IMPORT JAVA classname

The IMPORT JAVA instruction imports Java module elements.

Syntax

IMPORT JAVA ▼ packagename . {...} ▼ classname

1.  packagename and classname define the Java class to be imported.
Usage

The **IMPORT JAVA** instruction can be used to import a Java class.

**Important:** At runtime, the imported Java classes are only loaded on demand, when the program flow reaches an instruction that uses the class. For example, when reaching the declaration of a variable defined to reference an object of a Java class.

The name specified after the **IMPORT JAVA** instruction is case-sensitive.

The **CLASSPATH** environment variable defines the directories for Java packages. See the Java documentation for more details.

**Note:** To mimic the Java import rules, the **fglcomp** compiler allows subsequent **IMPORT JAVA** instructions with the same class name. However, it is recommended that you avoid duplicating the same **IMPORT JAVA** instructions:

```
IMPORT JAVA java.util.regex.Matcher
...
IMPORT JAVA java.util.regex.Matcher
```

Related concepts

- **The Java interface** on page 2192
- The **Java interface** allows you to import Java classes and instantiate Java objects in your programs.

Program execution

This section describes program execution and language instructions related to program execution.

Executing programs

There are different ways to execute compiled programs, depending on the configuration and the development or production context.

**Prerequisites before executing a program**

Make sure that all required environment variables are properly defined, such as: **FGLPROFILE** on page 279, **FGGLGUI** on page 275, **FGLSERVER** on page 280, **FGLLDPATH** on page 278, **LC_ALL** (or **LANG**) on page 264.

To display program forms in graphical mode, the GUI front-end must run on the computer defined by **FGLSERVER**, and all network security components (such as firewalls) must allow TCP connections on the port defined by this environment variable.

Verify the database client environment settings, and check that the database server is running and can be accessed, for example by using a database vendor specific tool to execute SQL commands.

**Starting a program from the command line on the server**

A program can be executed with the **fglrun** tool from the server command line:

```
fglrun myprogram
```

This method is typically used in development context. After compiling the programs and forms, for example with the **make** utility, execute the programs with **fglrun**.

**Note:** The file extension (.42m or .42r) can be omitted. If no file extension is specified, **fglrun** will try to load **proname.42r**, then **proname.42m**.

**Executing sub-programs from a parent program with RUN**

Sub-programs can be executed from a parent program with the **RUN** instruction.
There can be limitations, depending on the platform where the parent program executes.

The `RUN` instruction provides **IN LINE/FORM MODE and WITHOUT WAITING** options, to be used depending on the purpose of the child program or command, and if the parent program runs in TUI or GUI mode.

For more details, see the reference page of the `RUN` instruction.

### Starting a program from the front-end

It is also possible to start programs on the application server from the platform where the front-end resides.

This is actually the typical way to start applications in a production environment.

- For a desktop front-end (GDC) application, define application shortcuts and use rlogin/ssh network protocols to start programs on the server or by using HTTP through a web server (GAS).
- For a web-browser application (GBC), configure the application server (GAS) to run applications from an URL.
- For a mobile device application (GMI/GMA), in a configuration where the programs run on a GAS application server, use the "runOnServer" front call, to start a program from the GAS.

### Starting programs on a mobile device

After deploying program files on a mobile device, it can be executed as a local application, typically with a tap on the application icon.

- For a GMA (Android™) application, program files and GMA must be bundled together in an .apk Android™ package to be deployed. For more details, see Deploying mobile apps on Android devices on page 3589.
- For a GMI (iOS) application, program files and GMI must be bundled together in an .ipa package to be deployed. For more details, see Deploying mobile apps on iOS devices on page 3604.
- To start programs on an application server from a small embedded mobile application (starter), use the `runOnServer` front call. For more details, see Running mobile apps on an application server on page 3616.

### Mobile app foreground and background modes

On mobile devices, apps can switch between the foreground and background mode.

The app state can be detected and managed in your Genero program code.

For more details, Background/foreground modes on page 3573.

### Common app directories on mobile platforms

On mobile devices, you can use the following APIs to get common directories:

1. `base.Application.getProgramDir` on page 2408 returns the directory path where the main .42m is located. Consider this location read-only and safe (no other app can access it). See also the `FGLAPPDIR` environment variable.
2. `os.Path.pwd` on page 2806 returns the path to the current working directory. When a mobile application is started, the GMA and the GMI set the working directory to the default application directory. Consider this location read-write and safe (no other app can access it).
3. The front call `standard.feInfo/dataDirectory` returns the front-end side temporary directory. Storage on this directory may be erased by the OS. On an embedded mobile application, as the runtime and the front-end run on the same system, the program can use this front call to retrieve a temporary directory and use the path to store temporary files. Consider this location read-write and unsafe. Applications executed remotely through a `runOnServer` front call, can use the `sandboxRunOnServer` directory under the directory returned by the `feInfo/dataDirectory` front call, to exchange files with the embedded application.

### Related concepts

- **Application locale** on page 512
  The `application locale` defines the language and codeset for your application.

- **Genero environment variables** on page 266
Operating system environment variables on page 264
Describes some well-known system environment variables that are used by Genero software components.

RUN
The RUN instruction executes the command passed as argument.

Syntax

```
RUN command
  [ IN [FORM] LINE ] MODE
  [ RETURNING variable ] WITHOUT WAITING
```

1. *command* is a string expression with the command to be executed.
2. *variable* is an integer variable receiving the execution status of the command.

Understanding the RUN command
The RUN instruction hands the argument command to the command interpreter. When not specifying the WITHOUT WAITING clause, the calling process waits for the called process to finish execution. Otherwise, the calling process waits the command termination.

**Important:** The RUN instruction has limited support on mobile platforms.

- The RUN instruction is not supported on mobile devices, because of operating system limitations.
- The RUN *command* WITHOUT WAITING is not supported when programs run on an application server and display on a mobile device, because the Genero GUI protocol is not able to handle multiple connections at the same time.

Defining the command execution shell
In order to execute the command line, the RUN instruction uses the OS-specific shell defined in the environment of the current user. On UNIX™, this is defined by the SHELL environment variable. On Windows®, this is defined by COMSPEC. On Windows®, the program defined by the COMSPEC variable must support the /c option as CMD.EXE.

Waiting for the subprocess
By default, the runtime system waits for the end of the execution of the command, suspending the execution of the current program. After executing the command, the display of the parent program is restored.

If you specify WITHOUT WAITING, the specified command line is executed as a background process, and generally does not affect the visual display. This clause can be used when the command takes some time to execute, and the parent program does not need the result to continue. It is also typically used in GUI mode to start another program. Do not use this clause in TUI mode when the sub-program displays forms, otherwise both programs would run simultaneously on the same terminal.

Catching the execution status
The RETURNING clause saves the termination status code of the command that RUN executes in a program variable of type SMALLINT. Examine the variable after execution to determine the next action to take. A status code of zero usually indicates that the command has terminated normally. A non-zero exit status indicates an error.

**Important:**
The execution status provided by the RETURNING clause is platform-dependent.

- On UNIX™ systems, the value is composed of two bytes having different meanings: The lower byte (x mod 256) of the return status defines the termination status of the RUN command. The higher byte (x / 256) of the return status defines the execution status of the program.
• On Windows® platforms, the execution status is zero for success, or a non-zero value if an error occurred: The value of the return status only defines the execution status of the program.

**IN LINE MODE and IN FORM MODE**

When using the TUI mode, programs operate by default in *line mode*, but as many statements take it into *form mode* (including OPTIONS statements that set keys, DISPLAY, OPEN WINDOW, DISPLAY FORM, and other screen interaction statements), typical interactive TUI programs are actually in *form mode* most of the time.

According to the type of command to be executed, you may need to use the **IN {LINE | FORM} MODE** clause with the RUN instruction. It defines how the terminal or the graphical front-end behaves when running the child process.

Besides RUN, the OPTIONS, START REPORT and START REPORT... TO PIPE instructions can explicitly specify a screen mode.

If no screen mode is specified in the RUN command, the current value from the OPTIONS statement is used. This is, by default, **IN LINE MODE**.

**Note:** With START REPORT using PIPE specifications, the default screen mode is **IN FORM MODE**.

When the RUN statement specifies **IN FORM MODE**, the program remains in *form mode* if it is currently in *form mode*, but it does not enter *form mode* if it is currently in *line mode*. When the prevailing RUN option specifies **IN LINE MODE**, the program remains in *line mode* if it is currently in *line mode*, and it switches to *line mode* if it is currently in *form mode*. This also applies to the PIPE option.

Typically, if you need to run another interactive program, you must use the **IN LINE MODE** clause:

• In TUI mode, the terminal is in the same state (in terms if *tty* options) as when the program began. Usually the terminal input is in cooked mode, with interrupts enabled and input not becoming available until after a newline character is typed.

• In GUI mode, if the WITHOUT WAITING clause in used, the front-end is warned before the child process is started (this causes a first network round-trip) After the child is started, the front-end is warned that the command was executed (second network round-trip). If the RUN command must wait for child termination (i.e. no WITHOUT WAITING clause is used), no particular action is taken.

However, if you want to execute a subprocess running silently (batch program without output), you must use the **IN FORM MODE** clause:

• In TUI mode, the screen stays in *form mode* if it was in *form mode*, which saves a clear / redraw of the screen. The FORM mode specifies the terminal raw mode, in which each character of input becomes available to the program as it is typed or read.

• In GUI mode, no particular action is taken to warn the front-end (there is no need to warn the front-end for batch program execution).

To summarize, no matter if you are in TUI or GUI mode, run silent (batch) programs in **FORM MODE**, and if the program to run is interactive, or displays messages to the terminal, or if you don’t known what it does, use the LINE MODE (which is the default).

**Important:** Executing an interactive Genero program with RUN … IN FORM MODE WITHOUT WAITING (or with RUN … WITHOUT WAITING and the FORM MODE is set by OPTIONS), can lead to unexpected behavior.

A good practice is to encapsulate child program and system command execution in functions.

**Example**

```plaintext
MAIN
  DEFINE result SMALLINT
  CALL runApplication("app2 -p xxx")
  CALL runBatch("ls -l", FALSE) RETURNING result
  CALL runBatch("ls -l > /tmp/files", TRUE) RETURNING result
END MAIN

FUNCTION runApplication(pname)
  DEFINE pname, cmd STRING
```
LET cmd = "fglrun " || pname
IF fgl_getenv("FGLGUI") == 0 THEN
   RUN cmd
ELSE
   RUN cmd WITHOUT WAITING
END IF
END FUNCTION

FUNCTION runBatch(cmd, silent)
   DEFINE cmd STRING
   DEFINE silent STRING
   DEFINE result SMALLINT
   IF silent THEN
      RUN cmd IN FORM MODE RETURNING result
   ELSE
      RUN cmd IN LINE MODE RETURNING result
   END IF
   IF fgl_getenv("OS") MATCHES "Win*" THEN
      RETURN result
   ELSE
      RETURN ( result / 256 )
   END IF
END FUNCTION

Related concepts
EXIT PROGRAM on page 492
The EXIT PROGRAM instruction terminates the execution of the program.

EXIT PROGRAM
The EXIT PROGRAM instruction terminates the execution of the program.

Syntax

EXIT PROGRAM [ exit-code ]

1. exit-code is a valid integer expression that can be read by the process which invoked the program.

Usage
Use the EXIT PROGRAM instruction to stop the execution of the current program instance.

exit-code must be zero by default for normal, successful program termination.

exit-code is converted into a positive integer between 0 and 255 (8 bits).

Example

MAIN
   DISPLAY "Emergency exit."
   EXIT PROGRAM -1
   DISPLAY "This will never be displayed."
END MAIN

Related concepts
RUN on page 490
The RUN instruction executes the command passed as argument.

**BREAKPOINT**

The BREAKPOINT instruction sets a program breakpoint when running in debug mode.

**Syntax**

```
BREAKPOINT
```

**Usage**

Normally, to set a breakpoint when you debug a program, you must use the break command of the debugger. But in some situations, you might need to set the breakpoint in program sources. Therefore, the BREAKPOINT instruction has been added to the language.

When you start `fglrun` in debug mode with the `-d` option, if the program flow encounters a BREAKPOINT instruction, the program execution stops and the debug prompt is displayed, to let you enter a debugger command. The BREAKPOINT instruction is ignored when not running in debug mode.

**Example**

```
MAIN
  DEFINE i INTEGER
  LET i=123
  BREAKPOINT
  DISPLAY i
  END MAIN
```

**Related concepts**

[Setting a breakpoint programmatically](page 2146)

Set a breakpoint in the program source code with the BREAKPOINT instruction.

**Runtime stack**

The runtime stack is used to pass/return values to/from functions.

When passing arguments to a function or when returning values from a function, you are using the runtime stack. When you call a function, parameters are pushed on the stack; before the function code executes, parameters are popped from the stack in the local variables defined in the function. On the other hand, each parameter returned by a function is pushed on the stack and popped into variables specified in the RETURNING clause of the caller.

Elements are pushed on the stack in a given order, then popped from the stack in the reverse order. This is transparent to the programmer. However, if you want to implement a C extension, you must keep this in mind.

According to the data type, parameters are passed and returned by value or by reference. When an element is passed/returned by value, a complete copy of the value is passed. When an element is passed by reference, only the handle of the object is passed/returned. If the type allows it, elements passed by reference can be manipulated in the called function to modify the value.

**Table 152: Function parameter and returning rules by language element type**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Data type or data structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>By value</td>
<td>BOOLEAN, BIGINT, INTEGER, SMALLINT, TINYINT, FLOAT, SMALLFLOAT, DECIMAL, MONEY, CHAR, VARCHAR, DATE, DATETIME, INTERVAL, records (by default, expanded) and static arrays (cannot be returned).</td>
</tr>
</tbody>
</table>
Passing simple typed values as parameter

Simple data types such as INTEGER, DECIMAL, VARCHAR are passed by value in function parameters. When passing a function parameter by value, the runtime system pushes a copy of the data on the stack.

The STRING data type is an exception to this rule for simple types: elements of this type are passed by reference. In fact the runtime system passes a reference to the string value, so the actual string data is not copied on the stack as for other simple types. However, the value of the caller cannot be modified. If a STRING parameter gets a new value in a function, a new string reference is created. Passed STRING parameters improve performances compared to CHAR/VARCHAR, with the same semantics as VARCHAR().

When passing a simple typed value to a function, the local variable receiving the value can be changed without affecting the variable used by the caller:

```plaintext
MAIN
    DEFINE c CHAR(10), s STRING
    LET c = "abc"
    LET s = "def"
    CALL func(c,s)
    DISPLAY c -- Shows "abc"
    DISPLAY s -- Shows "def"
END MAIN

FUNCTION func(pc,ps)
    DEFINE pc CHAR(10), ps STRING
    DISPLAY c -- Shows "abc" (this is a copy of the string)
    DISPLAY s -- Shows "def" (this is the original string)
    LET pc = "zz" -- Assigns new value to local variable
    LET ps = "zz" -- Assigns new value to local variable
END FUNCTION
```

Related concepts

- **Primitive Data types** on page 289
  Selecting the correct data type assists you in the input, storage, and display of your data.

- **Type conversions** on page 311
  Explains primitive data type conversion rules of the language.

Passing records as parameter

Passing records by expansion (. * notation)

All elements of a RECORD structure can be passed by value to a function with the dot star (. *) notation.

**Note:** The record. * parameter syntax is supported for backward compatibility. Consider passing records by reference, or by value by specifying the record name only.

When using the . * notation as function parameter, the record is expanded: Each member of the record structure is pushed on the stack. The receiving local variables in the function must be defined with the same record structure as the caller.

For backward compatibility, the runtime system allows to call functions defined with the individual parameters matching the record members (or the other way around - calling a function with individual values that match the record parameter definition of the function). However, this is bad practice.
**Important:** When using methods, the parameter type checking is enforced. It is not possible to pass records with the \.* notation like with legacy functions: The compiler will reject the code, if the complex parameters types (records, arrays) do not match.

In the next example, the `func_r()` function is defined with a record parameter, while the `func_ab()` function is defined with individual integer and string parameters:

```fortran
MAIN
   DEFINE rec RECORD
      a INT,
      b VARCHAR(50)
   END RECORD
   CALL func_r(rec.*)
   CALL func_ab(rec.*)     -- Bad practice!
   CALL func_r(101, 'aaa') -- Bad practice!
END MAIN

-- Function defining a record like that in the caller
FUNCTION func_r(r)
   DEFINE r RECORD
      a INT,
      b VARCHAR(50)
   END RECORD
   DISPLAY "func_r : ", r.*
END FUNCTION

-- Function defining two individual variables
FUNCTION func_ab(a, b)
   DEFINE
      a INT,
      b VARCHAR(50)
   DISPLAY "func_ab: ", a, b
END FUNCTION
```

### Passing records by reference with INOUT

To pass records by reference, define a TYPE with the RECORD structure, and use this type in the parameter definition of the function, followed by the INOUT keyword.

The function must be known by the fglcomp compiler: It is not possible to use this feature for linked functions. The function defining the INOUT parameter must be declared locally in the same module, or it must be defined in a module imported with IMPORT FGL.

The function can then be invoked by specifying the record name as parameter.

The record structure defined in the caller can then be modified inside the function body:

```fortran
TYPE CustRec RECORD
   cust_id INTEGER,
   cust_name VARCHAR(50)
END RECORD

MAIN
   DEFINE r CustRec
   CALL initialize_customer(r)
   DISPLAY r.*
END MAIN

FUNCTION initialize_customer(r CustRec INOUT)
   LET r.cust_id = 0
   LET r.cust_name = "<undefined>"
END FUNCTION
```
Passing records by value

Records can be passed by value (without the . notation) to methods for types:

```plaintext
FUNCTION (c t_cust) setLastOrderId(o t_order)
    LET c.last_order_id = o.order_id
END FUNCTION
```

Passing records by value is possible in API methods such as `util.JSON.stringify()`:

```plaintext
IMPORT util
...
DEFINE rec RECORD
    cust_id INTEGER,
    cust_name VARCHAR(50)
END RECORD
...
CALL util.JSON.stringify( rec )
...
```

The record is not fully copied on the stack.

Passing records implicitly by reference to methods of built-in classes

With API methods such as `util.JSON.parse()`, the record is implicitly passed by reference, and it can be modified by the method:

```plaintext
IMPORT util
...
DEFINE rec RECORD
    cust_id INTEGER,
    cust_name VARCHAR(50)
END RECORD
...
CALL util.JSON.parse( '{"cust_id":2735, "cust_name":"McCarlson"}', rec )
...
```

Related concepts

Records on page 408

Records allow structured program variables definitions.

Passing static arrays as parameter

It is possible to pass a complete static array as a function parameter, but this is not recommended. When passing a static array to a function, the complete array is copied on the stack and every element is passed by value. The receiving local variables in the function must be defined with the same static array definition as the caller:

```plaintext
MAIN
    DEFINE arr ARRAY[5] OF INT
    CALL func(arr)
END MAIN

-- function defining same static array as the caller
FUNCTION func(x)
    DEFINE x ARRAY[5] OF INT
    ...
END FUNCTION
```
Note: Dynamic arrays are passed by reference.

Related concepts

Static arrays on page 420
Static arrays have a predefined and limited size.

Passing dynamic arrays as parameter

Passing a dynamic array as a function parameter is legal and efficient. When passed as parameter, the runtime system pushes a reference of the dynamic array on the stack, and the receiving local variables in the function can then manipulate the original data.

Returning a dynamic array from a function is also possible: The runtime system pushes the reference of the dynamic array on the stack.

```
MAIN
  DEFINE arr DYNAMIC ARRAY OF INT
  DISPLAY arr.getLength()
  LET arr = init(10)
  DISPLAY arr.getLength()
  CALL modify(arr)
  DISPLAY arr[50]
  DISPLAY arr[51]
  DISPLAY arr.getLength()
END MAIN

FUNCTION init(c)
  DEFINE c INT
  DEFINE x DYNAMIC ARRAY OF INT
  FOR i=1 TO c
    LET x[i] = i
  END FOR
  RETURN x
END FUNCTION

FUNCTION modify(x)
  DEFINE x DYNAMIC ARRAY OF INT
  LET x[50] = 222
  LET x[51] = 333
END FUNCTION
```

Output of the program:

```
0
10
222
333
51
```

Related concepts

Dynamic arrays on page 422

Passing dictionaries as parameter

Passing a dictionary as a function parameter is legal and efficient. When passed as parameter, the runtime system pushes a reference of the dictionary on the stack, and the receiving local variables in the function can then manipulate the original data.
Returning a dictionary from a function is also possible: The runtime system pushes the reference of the dictionary on the stack.

```
MAIN
  DEFINE dic DICTIONARY OF STRING
  DISPLAY dic.getLength()
  LET dic = init()
  DISPLAY dic.getLength()
  CALL modify(dic)
  DISPLAY dic["first"]
  DISPLAY dic["second"]
  DISPLAY dic.getLength()
END MAIN

FUNCTION init()
  DEFINE x DICTIONARY OF STRING
  LET x["first"] = "abc"
  LET x["second"] = "def"
  RETURN x
END FUNCTION

FUNCTION modify(x)
  DEFINE x DICTIONARY OF STRING
  LET x["first"] = "xyz"
  LET x["third"] = "ijk"
END FUNCTION
```

Output of the program:

```
  0
  2
  xyz
  def
  3
```

**Related concepts**

*Dictionaries* on page 426

A dictionary holds an unordered collection of elements accessed by a key.

**Passing objects as parameter**

Like other object oriented programming languages, objects of built-in classes or Java classes are passed by reference. It would not make much sense to pass an object by value, actually. The runtime pushes the reference of the object on the stack (i.e. the object handler is passed by value), and the reference is then popped to the receiving object variable in the function. The function can then be used to manipulate the original object.

```
MAIN
  DEFINE ch base.Channel
  LET ch = base.Channel.create()
  CALL open(ch)
  CALL ch.close()
END MAIN

FUNCTION open(x)
  DEFINE x base.Channel -- Channel object reference
  CALL x.openFile("filename","r")
END FUNCTION
```

**Related concepts**

*OOP support* on page 570
The Java interface on page 2192
The Java interface allows you to import Java classes and instantiate Java objects in your programs.

Passing TEXT/BYTE values as parameter

BYTE or TEXT data types define large data object (LOB) handlers internally implemented as "locators".

When you pass a BYTE or TEXT to a function, the locator is pushed on the stack and popped to the receiving BYTE or TEXT variable in the function. The actual LOB data is not copied, only the locator is passed by value.

Important: Since the information of the locator structure is copied (like the file name specified with a LOCATE IN FILE instruction). If you modify the locator storage information inside the function with a LOCATE instruction, the locator in the caller will become invalid. Therefore, only read and write the actual data of BYTE and TEXT parameters in functions, do not modify the storage.

Returning simple typed values from functions

Simple data types such as INTEGER, DECIMAL, VARCHAR are returned by value with the RETURN instruction.

When returning a simple typed value, the runtime system pushes a copy of the data on the stack. The STRING data type is an exception to this rule: elements of this type are return by mutable reference: the whole string value is not copied on the stack, only the reference to the string value is copied.

```
MAIN
   DEFINE x INTEGER
   LET x = int_add(10,20)
END MAIN

FUNCTION int_add(n1,n2)
   DEFINE n1, n2 INTEGER
   RETURN (n1+n2)
END FUNCTION
```

Returning records from functions

In order to return a complete RECORD from a regular FUNCTION, you must put a copy of all record members on the stack, by expanding the record with the .* notation. This is however supported with backward compatibility and not best practice: Consider passing records as reference with the INOUT keyword, or use methods and return the record without using the .* notation.

```
TYPE t_rec RECORD
   name VARCHAR(50)
END RECORD

TYPE t_info RECORD
   code INT,
   message STRING
END RECORD
```

Returning records from methods

To return a record from a method, define the method with the returned type name in the RETURNS clause, and specify the record variable without using the .* notation in the RETURN instruction of the method body:

```
TYPE t_rec RECORD
   name VARCHAR(50)
END RECORD

TYPE t_info RECORD
   code INT,
   message STRING
END RECORD
```
FUNCTION (r t_rec) getInfo() RETURNS t_info
  DEFINE i t_info
  LET i.code = 999
  LET i.message = "xxxxxxx"
  RETURN i -- WITHOUT .* NOTATION!
END FUNCTION

FUNCTION main()
  DEFINE r t_rec
  DEFINE i t_info
  LET i = r.getInfo()
  DISPLAY i.*
END FUNCTION

Related concepts
RETURN on page 380
The RETURN instruction gives the control of execution back to the caller, optionally returning values on the stack.

Methods on page 452
A function declared with a receiver type defines a method for this type.

Returning dynamic arrays from functions
When returned by a function, dynamic arrays are pushed on the stack by reference.
Therefore, you can create a dynamic array in a function and return it to the caller for usage:

MAIN
  DEFINE arr DYNAMIC ARRAY OF INTEGER
  LET arr = create_array(10)
  DISPLAY arr.getLength()
END MAIN

FUNCTION create_array(n)
  DEFINE n, i INTEGER
  DEFINE arr DYNAMIC ARRAY OF INTEGER
  FOR i=1 TO n
    LET arr[i] = i
  END FOR
  RETURN arr
END FUNCTION

Related concepts
RETURN on page 380
The RETURN instruction gives the control of execution back to the caller, optionally returning values on the stack.

Returning dictionaries from functions
When returned by a function, dictionaries are pushed on the stack by reference.
Therefore, you can create a dictionary in a function and return it to the caller for usage:

MAIN
  DEFINE dic DICTIONARY OF STRING
  LET dic = create_dictionary(10)
  DISPLAY dic["item3"]
  DISPLAY dic["item10"]
END MAIN

FUNCTION create_dictionary(n)
  DEFINE n, i INTEGER
DEFINE dic DICTIONARY OF STRING
FOR i=1 TO n
    LET dic[SFMT("item%1",i)] = SFMT("This is item %1",i)
END FOR
RETURN dic
END FUNCTION

**Related concepts**

**RETURN** on page 380

The RETURN instruction gives the control of execution back to the caller, optionally returning values on the stack.

### Returning TEXT/BYTE values from functions

When returning a TEXT or BYTE value from a function, the locator is pushed in on the stack.

Storage information of the TEXT/BYTE is defined in the locator structure and controlled with the LOCATE instruction. The storage of the large object variable can be defined in a function, initialize the object with a value, and return it:

```plaintext
MAIN
    DEFINE t TEXT
    LET t = init_text(t)
    DISPLAY "t size = ", LENGTH(t)
END MAIN

FUNCTION init_text(t)
    DEFINE t TEXT
    LOCATE t IN MEMORY
    LET t = "abc"
    RETURN t
END FUNCTION
```

The above sample will produce following output:

```
t size = 3
```

**Related concepts**

**RETURN** on page 380

The RETURN instruction gives the control of execution back to the caller, optionally returning values on the stack.

### Implicit data type conversion on the stack

When a value or a reference is popped from the stack, implicit data conversion takes place. This means, for example, that you can pass a string value to a function that defines the receiving variable as a numeric data type; no compilation error will occur, but you can get a runtime error if the string cannot be converted to a numeric. The same principle applies to values returned from functions, since the stack is also used in this case.

```plaintext
MAIN
    DEFINE s STRING
    LET s = "123"
    CALL display_integer(s) -- Will be accepted
    LET s = "abc"
    CALL display_integer(s) -- Will fail with conversion error
END MAIN

FUNCTION display_integer(x)
    DEFINE x INTEGER
    DISPLAY x
END FUNCTION
```
Related concepts

Type conversions on page 311
Explains primitive data type conversion rules of the language.

Exceptions

Describes exception (error) handling in the programs.

Understanding exceptions

Exceptions are abnormal runtime events that can be trapped for control.

If an instruction executes abnormally, the runtime system throws exceptions that can be handled by the program.

Specific exception actions can be taken based on the class of the exception.

Runtime errors (exceptions) can be trapped by a WHENEVER exception handler or by a TRY/CATCH block. The WHENEVER exception is provided for compatibility with Informix 4GL, consider using TRY/CATCH blocks instead.

Note: Some specific errors such as error -1338 cannot be trapped. In such case, the program will always stop and show the error, even if a WHENEVER exception handler or TRY/CATCH block is used.

A Genero exception is identified by its number and has a description. For a complete list of BDL errors, see Genero BDL errors on page 3154.

Exception handlers are typically used to detect database errors when executing SQL statement. For more details, see SQL execution diagnostics on page 591

Exception classes

Exception classes indirectly define the exception type.

The default action can be changed by specifying the exception class in the WHENEVER instruction.

Table 153: Exception classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Error reason</th>
<th>Default Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR (or SQLERROR)</td>
<td>Language or SQL statement error.</td>
<td>STOP</td>
</tr>
<tr>
<td>ANY ERROR (or ANY SQLERROR)</td>
<td>Language, SQL statement and expression error.</td>
<td>CONTINUE (1)</td>
</tr>
<tr>
<td>NOT FOUND</td>
<td>SQL statements returning status NOTFOUND.</td>
<td>CONTINUE</td>
</tr>
<tr>
<td>WARNING</td>
<td>SQL statements setting SQLCA.SQLAWARN flags.</td>
<td>CONTINUE</td>
</tr>
</tbody>
</table>

For example, the following WHENEVER instruction defines the behavior for the ANY ERROR exception class:

WHENEVER ANY ERROR CONTINUE

Related reference

Genero BDL errors on page 3154
System error messages sorted by error number.

Exception actions

Exception actions define the type of action to be taken when an exception occurs.

There are five exception actions that can be executed if an exception is raised:
**STOP**

The program is immediately terminated. A message is displayed to the standard error with the location of the related statement, the error number, and the details of the exception.

**CONTINUE**

The program continues normally. The exception is ignored, but can be checked by testing the STATUS register, or the SQLCA.SQLCODE register for SQL errors.

**CALL exception-function**

The function `exception-function` is called by the runtime system. The function can be defined in any module, and must have zero parameters and zero return values. The STATUS variable will be set to the corresponding error number.

**GOTO exception-label**

The program execution continues at the label identified by `exception-label`, as if a GOTO instruction was issued after trapping the exception.

**RAISE**

This statement instructs the runtime system that the exception must propagated to the calling function.

**Important:** The WHENEVER [ANY] ERROR RAISE is not supported in a REPORT routine.

---

**WHenever instruction**

Use the WHENEVER instruction to define how exceptions must be handled for the rest of the module.

**Syntax**

```
WHENEVER exception-class 
   exception-action
```

where `exception-class` is one of:

1. [ANY] ERROR
2. [ANY] SQLERROR
3. NOT FOUND
4. WARNING

and `exception-action` is one of:

1. CONTINUE
2. STOP
3. CALL [module.]function
4. RAISE
5. GOTO label

---

1. `function` can be any function name defined in the program.
2. `module` is the name of a module imported with IMPORT FGL.
3. `label` must be a label defined in the current program block (main, function or report routine).
**Usage**

The **WHENEVER** instruction defines the exception handling by associating an *exception class* with an *exception action*.

**Important:** The **WHENEVER** instruction is similar to a C preprocessor macro: Its effect is local to the module and defines the error handling for the rest of the module, unless a new **WHENEVER** instruction is encountered by the compiler, or a **TRY/CATCH** block is used. A **WHENEVER** instruction defines only the exception handling for that module and therefore does not affect callers. Eventually, use **WHENEVER ERROR RAISE** to propagate exceptions occurring in the module.

If no **WHENEVER** instruction is used, the default is **WHENEVER ERROR STOP**. Stopping the program in case of error is the recommended default. However, this default does not catch expression errors like type conversion errors. Consider using the `fglrun.mapAnyErrorToError` FGLPROFILE entry, to catch conversion errors. For more details, see Default exception handling on page 508.

This code example shows a typical **WHENEVER** instruction usage:

```
WHENEVER ERROR CONTINUE
DROP TABLE mytable -- SQL error will be ignored
CREATE TABLE mytable ( k INT, c VARCHAR(20) )
WHENEVER ERROR STOP
IF SQLCA.SQLCODE != 0 THEN
   ERROR "Could not create the table..."
END IF
```

**Note:** Unlike the **TRY/CATCH** block, a **WHENEVER ERROR CONTINUE** exception handler does not stop the evaluation of an expression. For example, in the expression `[(Pi() / 0) + Pi()]` (where `Pi()` is a user function returning the number Pi), the function is called twice, even if the division by zero produces error -1202. This behavior exists to be compatible with Informix 4GL legacy code.

Exception classes **ERROR** and **SQLERROR** are synonyms: In the previous example it is also possible to use **WHENEVER SQLERROR** instead of **WHENEVER ERROR**.

Actions for classes **ERROR**, **WARNING** and **NOT FOUND** can be set independently:

```
WHENEVER ERROR STOP
WHENEVER WARNING CONTINUE
WHENEVER NOT FOUND GOTO not_found_handler
...```

The effect of the **WHENEVER** instruction is local to the current module, and applies to all source lines following that instruction (cross-function definitions). In the next example, **module1.4gl** uses a **WHENEVER** instruction that takes effect in the second function, but does not affect the calling module **main.4gl**:

```
$ head module1.4gl main.4gl
===> module1.4gl <==
FUNCTION function1()
   WHENEVER ANY ERROR CONTINUE -- applies to subsequent lines
END FUNCTION

FUNCTION function2()
   DEFINE x INTEGER
   LET x = "aaa"
   DISPLAY "function2: x = ", x, " STATUS = ", STATUS
END FUNCTION

===> main.4gl <==
IMPORT FGL module1
MAIN
   DEFINE x INTEGER
   WHENEVER ANY ERROR STOP
```
CALL module1.function2()
    LET x = "aaa"
END MAIN

$ fglcomp main.4gl && fglrun main.42m
function2: x =             STATUS =       -1213
Program stopped at 'main.4gl', line number 6.
FORMS statement error number -1213.
A character to numeric conversion process failed.

In the above example, the error -1213 in line 6 of main.4gl is expected, because the WHENEVER instruction of the main.4gl module applies.

When using the WHENEVER ... CALL function instruction, the program flow will go to the specified function and return to the code block where the exception occurred:

MAIN
    DEFINE x INTEGER
    WHENEVER ANY ERROR CALL error_handler
        -- WHENEVER handler takes effect
        LET x = 1/0
        DISPLAY "Back in MAIN..."
END MAIN

FUNCTION error_handler()
    DISPLAY "error_handler: ", STATUS
END FUNCTION

-- output:
error_handler:       -1202
Back in MAIN...

Note: In a WHENEVER ... CALL instruction, do not specify parentheses after the function name.
A TRY/CATCH block takes precedence over the last WHENEVER instruction, see the following example:

MAIN
    DEFINE x INTEGER
    WHENEVER ANY ERROR CONTINUE
        -- WHENEVER handler takes effect
        LET x = 1/0
        DISPLAY "WHENEVER: ", STATUS
        -- WHENEVER handler is hidden by TRY/CATCH block
    TRY
        LET x = 1/0
    CATCH
        DISPLAY "CATCH   : ", STATUS
    END TRY
    -- WHENEVER handler takes again effect
    CALL func()
END MAIN

FUNCTION func()
    DEFINE x INTEGER
    LET x = 1/0
    DISPLAY "WHENEVER: ", STATUS
END FUNCTION

-- Output:
WHENEVER:       -1202
CATCH   :       -1202
WHENEVER: -1202

The RAISE option can be used to propagate exceptions to the caller, which typically traps the error in a TRY/CATCH block:

```
-- main.4gl
IMPORT FGL myutils
MAIN
    TRY
        -- Pass a NULL form name to get error -1110
        CALL myutils.open_form(NULL)
    CATCH
        DISPLAY "Error: ", status
    END TRY
END MAIN

-- myutils.4gl
FUNCTION open_form(fn)
    DEFINE fn STRING
    WHENEVER ERROR RAISE -- Propagate exceptions to caller
        OPEN FORM f1 FROM fn
    END FUNCTION
```

**Important:** WHENEVER [ANY] ERROR RAISE is not supported in a REPORT routine.

**Related concepts**

The SQLCA diagnostic record on page 594

The SQLCA variable is a predefined record containing SQL statement execution information.

**STATUS** on page 567

**STATUS** is a predefined variable that contains the execution status of the last instruction.

**TRY - CATCH block**

Use TRY / CATCH blocks to trap runtime exceptions in a delimited code block.

**Syntax:**

```
TRY
    instruction
    [...]
CATCH
    instruction
    [...]
END TRY
```

**Usage:**

Any language instruction in the TRY block will be executed until an exception is thrown. After an exception the program execution continues in the CATCH block. If no CATCH block is provided, the execution continues after END TRY.

If no exception is raised by the statements between the TRY and CATCH keywords, the instructions in the CATCH section are ignored and the program flow continues after END TRY.

This code example shows a TRY block executing an SQL statement:

```
TRY
    SELECT COUNT(*) INTO num_cust FROM customers WHERE ord_date <= max_date
CATCH
    ERROR "Error caught during SQL statement execution:", SQLCA.SQLCODE
```
A TRY block can be compared with WHENEVER ANY ERROR GOTO. Here is the equivalent of the previous code example:

```
WHENEVER ANY ERROR GOTO catch_error
  SELECT COUNT(*) INTO num_cust FROM customers WHERE ord_date <= max_date
  GOTO no_error
LABEL catch_error:
WHENEVER ERROR STOP
  ERROR "Error caught during SQL statement execution:", SQLCA.SQLCODE
LABEL no_error
```

The TRY statement can be nested in other TRY statements. In this example, the instruction in line #5 will be executed in case of SQL error:

```
TRY
  TRY
    SELECT COUNT(*) INTO num_cust FROM customers
  CATCH
    ERROR "Try block 2: ", SQLCA.SQLCODE
  END TRY
  CATCH
    ERROR "Try block 1: ", SQLCA.SQLCODE
END TRY
```

**Note:** Unlike the WHENEVER instruction, a TRY/CATCH block exception handler stops the evaluation of an expression at first error. For example, in the expression `(Pi() / 0) + Pi()` (where Pi() is a user function returning the number Pi), the function is called only once, because the expression evaluation stops at division by zero error -1202.

The TRY statement takes control over the current WHENEVER [ANY] ERROR handler, for the code lines between the TRY and CATCH keywords. However, between the CATCH and END TRY keywords, the current WHENEVER handler gets the control back. For example, in the next code example, the program output will show "In CATCH/END TRY":

```
DEFINE where STRING
MAIN
  WHENEVER ERROR CALL error_handler
  TRY
    LET where = "In TRY/CATCH"
    CONNECT TO "dummy"
  CATCH
    LET where = "In CATCH/END TRY"
    CONNECT TO "dummy"
  END TRY
END MAIN

FUNCTION error_handler()
  DISPLAY where
END FUNCTION
```

The WHENEVER ERROR RAISE instruction can be used module-wide to define the behavior when an exception occurs in a function that is called from a TRY / CATCH block. If an exception occurs in a statement after the WHENEVER ERROR RAISE instruction, the program flow returns from the function and raises the exception as if it had occurred in the code of the caller. If the exception is thrown in the MAIN block, the program stops because the exception cannot be processed by a caller. In this example, the instruction in line #5 will be executed if an exception occurs in the cust_report() function:

```
MAIN
  TRY
```
CALL cust_report()
CATCH
  ERROR "An error occurred during report execution: ", STATUS
END TRY
END MAIN

FUNCTION cust_report()
  WHENEVER ERROR RAISE
  START REPORT cust_rep ...
  ...
END FUNCTION

Important: It is not possible to set a debugger break point at TRY, CATCH or END TRY: The TRY statement is a pseudo statement, the compiler does not generate p-code for this statement.

Related concepts
SQL execution diagnostics on page 591
If an SQL statement execution fails, error description can be found in the SQLCA.SQLCODE, SQLSTATE, STATUS and SQLERRMESSAGE predefined registers.

The SQLCA diagnostic record on page 594
The SQLCA variable is a predefined record containing SQL statement execution information.

Tracing exceptions
Exception can be logged in a file when using the STARTLOG() function.

Exceptions are automatically logged in a file, if all the following conditions are true:

- The STARTLOG function has been previously called to specify the name of the exception logging file.
- The exception action is set to CALL, GOTO or STOP. Exceptions are not logged when the action is CONTINUE or RAISE.
- The exception class is an ERROR, ANY ERROR or WARNING. NOT FOUND exceptions cannot be logged.

In other words, errors will not be logged in the case of WHENEVER [ANY] ERROR CONTINUE, or when controlled by a TRY/CATCH block.

Each log entry contains:
- The system-time
- The location of the related instruction (source-file, line)
- The error-number
- The text of the error message, giving human-readable details for the exception

Related concepts
SQL error identification on page 595
Identify SQL exceptions in your programs with SQLCA.SQLCODE.

Related reference
Genero BDL errors on page 3154
System error messages sorted by error number.

Default exception handling
Default exception handling must be adapted to your programming pattern.

Default program behavior when exception occurs
By default, when a language or SQL error occurs, the program stops and shows a message to the end user. However, expression errors like division by zero or invalid type conversions will not stop the program. In other words, the default is WHENEVER ERROR STOP + WHENEVER ANY ERROR CONTINUE.

In TUI mode, with default exception handlers, the error message is displayed in the terminal.
In GUI mode, with default exception handlers, the error message is displayed in a popup message box, that the user can read before the program stops. If showing detailed error messages to the end user is considered as a security risk (to prevent attacks), the original error message can be replaced by a generic error message defined in the gui.programStoppedMessage FGLPROFILE entry:

```plaintext
gui.programStoppedMessage = "An unexpected error occurred, program will stop."
```

**Enforce expression error handling**

By default, the WHENEVER ANY ERROR action is to CONTINUE the program flow. This means that when a program makes for example a division by zero or when a string cannot be converted to a number, the program continues. This can lead to unexpected program behavior, but is the default to be backward compatible for legacy applications.

To make your code more robust, use WHENEVER ANY ERROR STOP.

Alternatively, in case of expression errors, you can force the runtime system to execute the action defined with WHENEVER ERROR exception class, with the following FGLPROFILE entry:

```plaintext
fglr.run.mapAnyErrorToError = true
```

When this entry is set to true, expression errors such as a division by zero will be trapped and execute the action defined by the last WHOEVEr ERROR instruction.

When using the default exception handler (WHENEVER ERROR STOP), the program with stop at any expression error, and display the corresponding error message.

FGLPROFILE file:

```plaintext
fglr.run.mapAnyErrorToError = true
```

Program code:

```plaintext
MAIN
  DEFINE x INT
  WHENEVER ERROR CALL my_error_handler
  LET x = 1 / 0   -- error handler will be called here
  DISPLAY "It continues...."
END MAIN

FUNCTION my_error_handler()
  DISPLAY "Handler: ", STATUS
END FUNCTION
```

**Related reference**

- Genero BDL errors on page 3154
- System error messages sorted by error number.

**Non-trappable errors**

Non-trappable errors are fatal errors that generally prevent further program execution.

If a non-trappable error occurs, neither WHENEVER instructions, nor TRY/CATCH blocks can trap the error.

In case of non-trappable error, the runtime system will do the following:

1. In GUI mode, display a pop-up window with the error message,
2. If STARTLOG() was previously called, write an error record to the log file,
3. Print the error message to stderr,
4. Stop the program.
Note: Some non-trappable errors such as -1212 can occur at runtime initialization. In such case, the GUI error message box and STARTLOG() output cannot be done.


Note: Some non-trappable errors may be trappable in a specific context. But as a general rule, consider the above errors always non-trappable.

Examples

WHENEVER and TRY/CATCH usage examples.

Example 1: Defining a error handler function

This code example defines a WHENEVER ERROR handler function called my_error_handler. After connecting to the database, a SELECT statements tries to fetch a row from a table that does not exist, and raises SQL error -217 when connected to Informix®:

```sql
MAIN
WHENEVER ERROR CALL my_error_handler
DATABASE stores
SELECT dummy FROM systables WHERE tabid=1
END MAIN

FUNCTION my_error_handler()
DISPLAY "Error:", STATUS
EXIT PROGRAM 1
END FUNCTION
```

Program output:

```
Error:  -217
```

Example 2: SQL error handling with WHENEVER

This code shows a typical SQL error handling block. It uses WHENEVER ERROR CONTINUE before executing SQL statements, tests the SQLCA.SQLCODE register for errors after each SQL instruction, and resets the default exception handler with WHENEVER ERROR STOP after the set of SQL commands to be controlled:

```sql
MAIN
DEFINE
   tabname VARCHAR(50),
   sqlstmt STRING,
   rowcount INTEGER

# In the DATABASE statement, no error should occur...
DATABASE stores

# But next SELECT may fail, if the user enters an invalid table name.
WHENEVER ERROR CONTINUE
PROMPT "Enter a table name:" FOR tabname
LET sqlstmt = "SELECT COUNT(*) FROM " || tabname
PREPARE s FROM sqlstmt
IF sqlca.sqlcode THEN
   DISPLAY "SQL Error occurred:", sqlca.sqlcode
   EXIT PROGRAM 1
END IF
EXECUTE s INTO rowcount
```
IF sqlca.sqlcode THEN
    DISPLAY "SQL Error occurred: ", sqlca.sqlcode
    EXIT PROGRAM 1
END IF
WHENEVER ERROR STOP

... (more instructions, stopping the program in case of error)
END MAIN

Program output in case of invalid table name:

SQL Error occurred: -217

Example 3: Typical TRY / CATCH block

This example uses a TRY/CATCH block to trap errors. In this case, we try to connect to an invalid database, which
will raise an SQL error and make the program flow go to the line after the CATCH statement:

MAIN
TRY
    DATABASE invalid_database_name
    DISPLAY "Will not be displayed"
CATCH
    DISPLAY "Exception caught, SQL error: ", SQLCA.SQLCODE
END TRY
END MAIN

Program output (with Informix®):

Exception caught, SQL error: -329

Example 4: TRY / CATCH in conjunction with WHENEVER

This code illustrates the fact that a TRY/CATCH block can be used together with a WHENEVER instruction: The
program first executes a WHENEVER ANY ERROR to define an error handler named foo and later it uses a TRY/
CATCH block to trap expression errors.

In this example, we intentionally force a division by zero. After the TRY/CATCH block, we force another division by
zero error, which will call the foo error handler:

MAIN
    DEFINE i INTEGER
    WHENEVER ANY ERROR CALL foo
TRY
    DISPLAY "Next exception should be handled by the catch statement"
    LET i = i / 0
CATCH
    DISPLAY "Exception caught, status: ", STATUS
END TRY
-- Previous error handler is restored after the TRY - CATCH block
    LET status = 0
    DISPLAY "Next exception should be handled by the foo function"
    LET i = i / 0
END MAIN

FUNCTION foo()
    DISPLAY "Function foo called, status: ", STATUS
END FUNCTION
Example 5: WHENEVER RAISE exception propagation

This example shows the usage of WHENEVER ... RAISE to propagate a potential exception to the caller. First the program defines the foo function as exception handler with WHENEVER ANY ERROR CALL foo, then it calls the do_exception function, which instructs the runtime system to propagate a potential error to the caller. As result, the division by zero in line #13 will be caught by the error handler defined in the MAIN block and call the foo function:

```
MAIN
    DEFINE i INTEGER
    WHENEVER ANY ERROR CALL foo
    DISPLAY "Next function call will generate an exception"
    DISPLAY do_exception(100, 0)
    WHENEVER ANY ERROR STOP -- reset default handler for rest of program

FUNCTION do_exception(a, b)
    DEFINE a, b INTEGER
    WHENEVER ANY ERROR RAISE
    RETURN a / b
END FUNCTION

FUNCTION foo()
    DISPLAY "Exception caught, status: ", STATUS
END FUNCTION
```

Program output:

```
Next function call will generate an exception
Exception caught, status:    -1202
```

Localization

Localization support allows you to implement programs that follow specific language and cultural rules.

Programs execute in a specific application locale. Beside the support of a locale specification which defines the character set used by programs, the internationalization of an application requires all strings in the sources that are subject to translation to be extracted and centralized. Localized strings are used to keep application messages and form labels in external resource files, which can be provided in different languages.

Application locale

The application locale defines the language and codeset for your application.

The application locale defines:

- The language (for messages),
- The country or territory (for currency symbols and date formats),
- The code set (for character set encoding).

A program needs to be able to determine its locale and act accordingly, to support different languages and character sets.

Important:
The same code point can represent distinct glyphs in different character sets. Even if the glyphs/characters seem to display properly on the screen, an invalid locale configuration in one of the software components will result in invalid characters in your database system.

Take for example a client application configured to display glyphs (font) for CP437. If the application gets a 0xA2 (decimal 162) code point, it displays an o-acute character. Now imagine that the DB client is configured with character set CP1252. In this character set, the code point 0xA2 is actually the cent currency sign. As a result, if the user enters the o-acute char (0xA2 in CP437) in the database, it will actually be interpreted as cent sign (0xA2 in CP1252) by the database server. When fetching that character back to the client, the database server returns the 0xA2 code point, which displays correctly as o-acute on the CP437 configured client, and the end user sees what was entered before. But with a different client application configured properly with CP1252 as DB client codeset, the end user will see the cent currency sign instead of the o-acute character.

Understanding locale settings
This is an introduction to application locale definition.

It is critical to understand how the different components of a program handle locale settings. Each component (i.e. runtime system, database server, database client software, front-end, and terminal emulator) have to be configured properly to get the correct character set conversions through the whole chain. The chain starts on the end-user workstation with front-end windows and ends in the database storage files.

Figure 23: The Locale Settings schema on page 514 shows the different components of a Genero Business Development Language process.

The yellow rectangles show where locale configuration parameters have to be set:

1. The source files are encoded in a given character set. When compiling sources (fglcomp, fglform), the compilers use the OS locale (LANG/LC_ALL) to encode the resulting program files (.42m, .42f). For more details, see Defining the application locale on page 519.  
   **Important:** When compiling, make sure that the LANG/LC_ALL environment variables match the encoding of the source files.

2. At runtime (fglrun), the OS locale (LANG/LC_ALL) for the runtime system must match the code set of the program files (.42m, .42f). For more details, see Defining the application locale on page 519.  
   **Important:** At runtime, make sure that the LANG/LC_ALL environment variables match the encoding of the program files.

3. The locale of the database client must match the locale of the runtime system. Each database vendor uses its own locale configuration system. For more details, see Database client settings on page 526.  
   **Important:** At runtime, make sure that the database client locale matches the application locale (i.e. LANG/LC_ALL).

4. The locale of the database server defines the encoding for data on the server side. This encoding can be different from the database client locale (one can for example store the data in UTF-8, while client programs use ISO-8859-15), but it is usually the same character set. See database vendor documentation for more details.

5. When using the TUI mode, the terminal emulator must be configured with the character set corresponding to the application locale (2).
Figure 23: The Locale Settings schema

The typical mistake is to forget to set the runtime system locale (LANG/LC_ALL), or the database client software locale.

A character string is just a set of bytes; The same code might represent different characters in different code sets. Therefore, systems cannot detect that the current locale is correct, and won't raise any error, except when a set of bytes does not represent a valid code point in the current code set. For example, the Latin letter é with acute (UNICODE: U+00E9) will be encoded as 0xE9/233 in CP1252, but in CP437 it will be encoded with 0x82/130. The codes 233 or 130 are valid characters in both code sets. If the database uses CP1252, 233 will represent an é and 130 will represent a curved quote. If the client application uses CP437, the é will be encoded as 130, stored as a curved quote, then fetched from the database as is and displayed back as é in the CP437 code page. As result, the end user cannot see that the character stored in the database is actually wrong, until another properly configured DB client application queries the database.

Note: Pay attention that on recent UNIX™ systems, the default OS locale is UTF-8 by default. If your application has been developed on an older system, it is probably using a single-byte character set like ISO-8859-15 or CP1252, and program need to be executed in this locale, not in the UTF-8 locale.

It is also important to identify database server character set (in other words what code set characters are stored in the database). Usually the database character set is defined when creating a database entity.
The best way to test if the characters inserted in the database are correct is to use the database vendor SQL interpreter and select rows inserted from a BDL program. The rows most hold non-ASCII data to check if the code of the characters is correct. Some databases support the ASCII() or better, the UNICODE() SQL function to check the code of a character. Use such function to determine the value of a character in the database field. If the character code does not correspond to the expected value in the character set of the database server, there is a configuration mistake somewhere.

If you run a BDL application in TUI mode (or a batch program doing DISPLAYs), you must properly configure the code set in the terminal window (X11 xterm, Windows® CMD, putty, etc). If the terminal code set does not match the runtime system locale, you will get invalid characters displayed on the screen. On Windows® platforms, the OEM code page of the CMD window can be queried/changed with the chcp command. On a Gnome terminal, go to the menu "Terminal" - "Set Character Encoding".

Related concepts
Defining the application locale on page 519
This section describes the settings defining the application locale, changing the behavior of the compilers and runtime system.

Database client settings on page 526
This section describes the settings defining the locale for the database client.

Locale and character set basics on page 516
This section is an introduction to locale and character set basics.

Quickstart guide for locale settings
This is a quick step-by-step guide to properly configure locale settings for your Genero application.

Setting the locale involves different components, which must all be configured properly.

Tip: This is a quickstart guide for locale settings. It is highly recommended that you read the complete set of articles regarding localization.

1. The application locale is defined by the character set used in your source files (.4gl, .per, .str). The same character set will be used in the compiled files (.42m, .42f, .42s). The current application locale can be checked with fglrun -i command, which displays information such as the character set name and length semantics.

2. Set the operating system locale corresponding to the application locale.
   - On UNIX® based systems (including Mac® OS-X®), define the LANG (or LC_ALL) environment variable. Use locale -a command to check if the locale exists on the machine. If not, it must be installed. If not set, LANG defaults to POSIX (ASCII).
   - On Windows® platforms, check if the regional settings for non-UNICODE applications match the application locale. If the regional settings do no match, you can define the LANG environment variable with a locale name supported by Microsoft® C Runtime Library, such as French_France.1252, or set LANG=.fglutf8 for the UTF-8 character set.
   - On Android™ and iOS mobile devices, the application locale is always UTF-8 and the length semantics is CHAR. This cannot be changed.

3. Check that the operating system character set name is listed in the charmap.alias file, to allow the runtime system to map it to a standard IANA character set name.

4. When using UTF-8 as character encoding, define the length semantics with the FGL_LENGTH_SEMANTICS={BYTE|CHAR} environment variable. On server platforms, Genero uses Byte Length Semantics by default for compatibility reasons. It's highly recommended to set FGL_LENGTH_SEMANTICS=CHAR to use Character Length Semantics. On mobile platforms, character length semantics is the default (this means that FGL_LENGTH_SEMANTICS does not need to be defined when running on a mobile device, it defaults to CHAR, and cannot be set to BYTE).

5. Set the database client locale with a character set corresponding to the application locale. For example, with Informix®, this is defined with the CLIENT_LOCALE environment variable. The actual name of the database client locale may be different from the application locale name. But remember that the application and database client character sets must match.
6. Check the length semantics used by the database. For example, with Oracle, it is recommended that you set the database option `NLS_LENGTH_SEMANTICS='CHAR'`, if the application uses CLS (typically with UTF-8).

7. With UTF-8, use the proper SQL character data type to store UTF-8 data: This data type can be different depending on the type of database server. For more details, see SQL character type for Unicode/UTF-8 on page 617.

8. Set the front-end locale (and font). By front-end, we mean the program the end user interacts with. This can be a Genero front-end or a terminal emulator like Gnome-term, Putty, or a Windows® Console. When using a Genero front-end, the front-end character set is defined by the type of the front-end, and the conversion from/to application character set is automatic, but you may need to select a font that is different from the system default. If you want to execute a TUI application in a terminal emulator, you must be sure that the terminal is configured to display the correct character set. This is for example defined with the `chcp` command on Windows®, or in the "Set Character Encoding" menu option of a Gnome-term.

9. Define the date, numeric and monetary formats with the DBDATE, DBMONEY, DBFORMAT environment variables. On server platforms such as UNIX™ and Windows®, these default to US formats (month/day/year for dates, the dot as decimal separator and $ as currency symbol). On mobile platforms, these default to the regional settings defined on the device.

**Locale and character set basics**

This section is an introduction to locale and character set basics.

Before starting with application/database design, configuration and settings, you must know some basics concerning language and character sets on computers. Check also the operating system and database server manuals covering localization or character set handling.

**Why do I need to care about the locale and character set?**

If you don't know what you are doing with character sets, the end user might get strange characters displayed on the screen, and will probably not be able to input non-ASCII characters.

In the worst case, as character set conversion can be symmetric for single-byte character sets, the end user might see correct characters on the workstation, but on the back-end you can get invalid characters in the database files.

By upgrading to a newer OS, Genero Business Development Language runtime or database system, or if a character set mapping utility was used somewhere in the chain, you can even get mixed character encoding in the database files.

**Related concepts**

Understanding locale settings on page 513

This is an introduction to application locale definition.

**Characters, code points, character sets, glyphs and fonts**

In computers, a *character* is the unit of information corresponding to a symbol of a natural language. This can be a letter, a digit, a punctuation mark, a mathematical or even musical symbol.

To represent a character in memory or in a file, computers must encode the character in a specific numeric value called *code point*. This code point uniquely identifies a character in a given *character set*.

Mapping a character to a code point is called *character encoding*. The same code point might represent a different character in several character sets.

The *glyph* is the graphical representation of the character. In other words, it's the way the character is drawn on the screen or on a printer.

Computers implement the glyph of characters with *fonts*, by mapping a code point to a bitmap image or drawing instructions based on math formulas or vector graphics.

**The ASCII character set**

ASCII stands for the American Standard Code for Information Interchange.

ASCII is a well-known character encoding based on the English alphabet. Characters are encoded in a single byte, using the 7 lower bits only.
Up to 127 characters, printable and non-printable (like control characters), are defined in ASCII.

**Note:** Nearly all other character sets (using 8 bits or multiple bytes such as UTF-8) define the first 127 characters as the ASCII character set.

Aliases for ASCII include ISO646-US, ANSI_X3.4-1968, IBM367, cp367, and more.

**Single-byte character sets (SBCS)**

A single-byte character set defines the encoding for characters on a unique byte. The size of a character is always one byte.

Example of single-byte character sets include ISO-8859-1, MS code page CP1252.

Genero Business Development Language supports single-byte character sets.

**Related concepts**

Length semantics settings on page 521

**Double-byte character sets (DBCS)**

A double-byte character set defines the encoding for characters on two bytes. The size of a character is always two bytes.

Example of double-byte character sets include UCS-2, used by SQL Server in NCHAR and NVARCHAR columns. Note that UTF-16 is not a (fixed) double-byte character set: You can have characters encoded on 2 or 4 bytes. UCS-2 is actually a subset of UTF-16.

Note that Genero Business Development Language does not support double-byte character sets.

**Related concepts**

Length semantics settings on page 521

**Multibyte character sets (MBCS)**

A multibyte character set defines the encoding for characters on a variable number of bytes. The size of a character can be one (usually ASCII chars), two, three or more bytes, depending on the character set.

Example of multibyte character sets are BIG5, EUC-JP, and UTF-8. BIG5 and EUC-JP characters can be one or two bytes long, while UTF-8 characters can be 1, 2, 3 or 4 bytes long (usually a maximum of 3 is sufficient).

Genero Business Development Language supports multibyte character sets.

**Related concepts**

Length semantics settings on page 521

**Single-byte character sets (SBCS)** on page 517

**Double-byte character sets (DBCS)** on page 517

**Character size unit and length semantics**

When programming an application for an occidental language such as English, a single-byte character set can be used, and the logical size, storage size and print width of characters is the same. For example, in ISO-8859-1, the ê character takes one logical position, has a storage size of one byte and a print width of one.

When programming an international application using multiple languages and a multibyte character set encoding, you must distinguish three size units:

1. The size in **character unit**, to count or position logical characters used in a string. For example, the strings abc and åðê have both a length of 3, in character units.
2. The size in **byte unit**, used to encode the character in a given character set. For example, a Latin ê acute character will use a unique byte in the ISO-8859-1 character set, but needs two bytes in UTF-8.
3. The size in **width unit**, used in formatting and alignments. The width is the length of the glyph/font of characters, especially in a fixed font. For example, a Latin character will take one width unit, while an Chinese character will take 2 width units.
Working with byte units in a multibyte character set can be difficult: You need to calculate sizes, lengths and substring offsets in a number of bytes, when the natural way is to count in characters.

Length semantics define the unit to be used for character data type definition, character string lengths and positions. With *Byte Length Semantics*, a length is expressed in bytes, while *Character Length Semantics* counts in characters.

**Related concepts**

Length semantics settings on page 521

**The UNICODE Standard**

*UNICODE* is a standard specification to map all possible characters to a numeric value, in order to cover all possible languages in a unique character set. UNICODE defines the mapping of characters to a numeric whole number, but it does not define how these integers are encoded in bytes.

The character encoding defines how the UNICODE number will be encoded in a byte or sequence of bytes.

Several character encodings are based on the UNICODE standard, such as UTF-7, UTF-8, UTF-16, UTF-32, UCS-2, and UCS-4. Each of these character sets use a different encoding method. For example, with UTF-8, the letter Æ is encoded with two bytes as 0xC3 and 0xB6, while the same character will be encoded 0x00C6 with UTF-16.

When Microsoft™ Windows® users talk about UNICODE, they typically mean UCS-2 or UTF-16, while UNIX™ users typically mean UTF-8.

**Related concepts**

Length semantics settings on page 521

When do I need a UNICODE character set? on page 518

**When do I need a UNICODE character set?**

With internationalization, people want to use different languages within the same application; for example, to have Chinese, Japanese, English, French and German addresses of customers in their database.

*UNICODE* is a character encoding specification that defines characters for all languages. More and more databases use an UNICODE character set on the database server, because it "standardizes" all data from different client applications.

If needed, the client application can then use a different character set like ISO-8859-1 or BIG5: The database software takes care of character set conversions. However, if the end user needs to deal with different languages, all components of the system (from database backend to GUI front-end) must work in UNICODE.

The UNICODE character set supported by Genero Business Development Language is UTF-8. Double-byte based UNICODE character sets such as UCS-2 or UTF-16 are not supported. The database server can however store character data in another UNICODE character set, as long as the database client is able to handle to conversion to/from UTF-8 for the Genero runtime system.

**Related concepts**

The UNICODE Standard on page 518

**What is the standard for UNICODE encoding?**

*UNICODE* is the standard for internationalization, but not all platforms/systems use the same UNICODE character set / encoding.

Recent UNIX™ distributions define UTF-8 as the default character set locale, XML files are UTF-8 by default, while Microsoft™ Windows® standard character encoding is UTF-16 (NTFS) / UCS-2 (SQL Server). Recent Microsoft™ Windows® 10 updates support UTF-8 as system locale for non-Unicode applications (in beta stage while writing these lines).

Genero BDL supports UTF-8 and this is the encoding that must be used with implement UNICODE applications.

**Note:** Files encoded in UTF-8 can start with the UTF-8 Byte Order Mark (BOM), a sequence of 0xEF 0xBB 0xBF bytes, also known as *UNICODE U+FEBF*. When reading files, Genero BDL will ignore the UTF-8 BOM,
if it is present at the beginning of the file. This applies to instructions such as LOAD, as well as I/O APIs such as base.Channel.read() and readLine().

**Related concepts**

The UNICODE Standard on page 518

**What is my current character set?**

**UNIX™ platforms**

UNIX™-like platforms provide the LANG / LC_ALL environment variables to define the locale.

Each process / terminal can set its own locale. On recent UNIX-like systems, the locale is en_US.utf8 by default.

Check for available locales with the locale -a command. Some systems come with only a few locales installed, you must then install an additional package to get more languages.

You must also define the correct character set in the terminal (xterm or gnome-term), otherwise non-ASCII characters will not display properly.

**Microsoft™ Windows® platforms**

On Windows® platforms, for non-UNICODE (that means non-UTF-16/UCS-2) applications, you have ACP and OEMCP code pages.

ACP stands for ANSI Code Page and were designed by Microsoft™ for GUI applications specifically, while OEMCP defines old code pages for MS/DOS console applications.

If needed, select the default ACP/OEMCP code pages for non-UNICODE application in the language and regional settings panel of Windows® (make sure you define the settings for non-UNICODE applications).

Code page can be changed in each console window with the chcp command.

With Genero Business Development Language, use the LANG environment variable on Windows® to define the character set for BDL. However, it is strongly recommended to use the default Windows® system locale and avoid setting LANG on Windows®.

**Related concepts**

Checking the locale configuration on UNIX platforms on page 535

**Defining the application locale**

This section describes the settings defining the application locale, changing the behavior of the compilers and runtime system.

**Language and character set settings**

**Purpose of application locale definition**

The locale settings matters at compile time and at runtime. At runtime, the locale changes the behavior of the character handling functions, such as UPSHIFT and DOWNSHIFT. It also changes the handling of the character strings, which can be single byte or multibyte encoded. Compilation errors will occur if the source files contain characters that do not exist in the encoding defined by the current locale.

Always check that the local environment variable matches the locale of your Genero application, during development and at runtime:

```
$ fglrun -i
Charmap      : UTF-8
Multibyte    : yes
Stateless    : yes
Length Semantics : CHAR
```
Mobile platforms

On iOS and Android™ mobile platforms, the locale is automatically defined to be UTF-8. This cannot be changed. The language conventions and system messages are defined by the device settings.

Microsoft™ Windows® platforms

On Windows® platforms, if you don't specify the LANG environment variable, the language and character set defaults to the system locale which is defined by the regional settings for non-Unicode applications. For example, on a US-English Windows®, this defaults to the 1252 code page. You typically leave the default on Windows® platforms (it is not recommended to set the LANG variable, unless your application uses a different character set to the Windows® system locale).

On Windows® platforms, the syntax of the LANG variable is:

```
language[_[territory][.codeset]]

| .codeset
```

For example:

```
C:\> set LANG=English_USA.1252
```

UNIX™ platforms

On UNIX™-based platforms, The LC_ALL (or LANG) environment variable defines the global settings for the language used by the application.

With the LANG environment variable (or LC_ALL, on UNIX™), you define the language, the territory (aka country) and the codeset (aka character set or code page) to be used. The format of the value is normalized as follows, but may be specific on some operating systems:

```
language_territory.codeset
```

For example:

```
$ LC_ALL=en_US.iso88591; export LC_ALL
```

What are possible locales on my platform?

Usually OS vendors define a specific set of values for the language, territory and codeset. For example, on a UNIX™ platform, you typically have the value “en_US.ISO8859-1” for a US English locale, while Microsoft™ Windows® requires the “English_USA.1252” value. For more details about supported locales, refer to the operating system documentation.

A list of available locales can be found on UNIX™ platform by running the locale –a command. You may also want to read the man pages of the locale command and the setlocale function. On Windows® platforms, search the Microsoft™ MSDN documentation for "Language and Country/Region Strings".

UNICODE support (UTF-8)

To support multiple languages in your application, you must use UNICODE. The encoding supported by Genero for UNICODE applications is UTF-8.

On UNIX™ platforms, UTF-8 locales are natively supported with LANG/LC_ALL.
On Windows® platforms, UTF-8 is not well supported by the operating system: Defining the LANG environment variable to code page 65001 will not work. To workaround this limitation, Genero implements UTF-8 support on Windows® by setting the LANG environment variable to the value .fglutf8:

C:\> set LANG=.fglutf8

Related concepts
Checking the locale configuration on UNIX platforms on page 535
Locale settings (LANG) corrupted on Microsoft platforms on page 535

Length semantics settings

Understanding length semantics

The length semantics of character string data matters when using a multibyte character set. Length semantics involves data type length specification for database column and program variable definitions, as well as string manipulations (for string lengths, character positions, offsets and substring ranges).

In a single-byte characters set like ISO-8859-1, a character is encoded on one byte. The length of a string can be counted in bytes or characters, the unit does not matter. In other words, the length semantics is identical in bytes or characters, with a single byte encoding. However, with a multibyte character set like UTF-8 or BIG5, a character can be encoded on several bytes. In such case, the unit regarding length semantics matters, because the number of bytes of a character string can be different from the number of characters.

For multibyte characters sets, the language supports Byte Length Semantics (BLS) and Character Length Semantics (CLS) specification. BLS or CLS usage depends on the current character set of the application. BLS is typically used with a character set such as BIG5, because for historical reasons programmers are used to counting 2 bytes for each Chinese character. For UTF-8, which is a variable size encoding, the recommendation is to use CLS instead. CLS simplifies data type definition and string handling when using UTF-8.

Programming areas concerned by length semantics are illustrated in the following code example:

```
SCHEMA shop

# CREATE TABLE mytable (
#   k INT,
#   vc VARCHAR(10)
#   -- what is the unit for the column size and how many
#   -- characters can be stored in this column?
# )

MAIN

DEFINE buf, tmp VARCHAR(50)  -- what is the unit for the size?
DEFINE rec RECORD LIKE mytable  -- what is the size of vc member?
DEFINE str STRING, len INT

DATABASE shop

SELECT LENGTH(vc) INTO len -- What unit use string functions in SQL?
   FROM mytable WHERE k = 45

LET buf = "abcdef..."  -- How many chars can this variable hold?

DISPLAY length(buf) -- In what unit is the length expressed?

LET tmp = buf[1,5]  -- What is the unit for char positions?

LET str = buf

DISPLAY str.getLength() -- What is the unit for the length?
DISPLAY str.getIndexOf("def") -- What is the unit for the offset?
```
Using Byte Length Semantics

Byte Length Semantics must be used if the current locale defines a multibyte character set different from UTF-8.

Important:

- Byte Length Semantics is the default on UNIX™ and Windows® platforms.
- Byte Length Semantics cannot be set on mobile platforms.

With BLS, the size of CHAR/VARCHAR program variables is expressed in byte units. In a single-byte character set like ISO-8859-1, every character is encoded on a unique byte, so the number of bytes equals the number of characters. When using BLS with a multibyte character set, you must be aware of the storage size in byte units:
Character encoding requires more than one byte, so the number of bytes to store a multibyte string is greater than the number of characters. For example, in a BIG5 encoding, one Chinese character needs 2 bytes, so if you want to hold a BIG5 string with a maximum of 10 Chinese characters, you must define a CHAR(20). When using UTF-8, characters can take one or several bytes which can use two or three times more storage space than character count. You need to choose the right expansion factor to define CHAR or VARCHAR variables in byte units.

```plaintext
-- Using Byte Length Semantics
DEFINE var VARCHAR(10)  -- Can store 10 bytes / 10 single-byte chars.
```

In order to use BLS, you can define the `FGL_LENGTH_SEMANTICS` environment variable to "BYTE", or just leave it unset, if BLS is the default on your platform. For example, on UNIX™:

```plaintext
$ FGL_LENGTH_SEMANTICS="BYTE"
$ export FGL_LENGTH_SEMANTICS
```

Using Char Length Semantics

Character Length Semantics is used with multibyte character sets such as UTF-8: Migrating to UTF-8 by using CLS will allow you to leave the source code untouched, even when doing complex string/substring manipulations.

The database typically also uses UTF-8 and CLS. If the database uses UTF-8 and only supports BLS, programs can still use CLS with UTF-8.

Important: Char Length Semantics is the default on iOS and Android™ mobile platforms, and cannot be changed (Byte Length Semantics cannot be used on mobile: only UTF-8 character set is allowed).

With CLS, the size of a CHAR/VARCHAR program variable is expressed in character units, and the number of bytes needed to store these characters is allocated automatically. A VARCHAR(10) variable will hold 10 characters, of any byte length. Furthermore, language functions and class methods dealing with character string length and positions will use character units.

```plaintext
-- Using Character Length Semantics
DEFINE var VARCHAR(10)  -- Can store 10 chars in UTF-8, or any encoding.
LET var = "Forêt"  -- 5 chars, that take 6 bytes in UTF-8
DISPLAY length(var)  -- Displays a length of 5 (characters)
DISPLAY ",",var[4,5],""
-- Displays [êt]
```

To enable Char Length Semantics, define the `FGL_LENGTH_SEMANTICS` environment variable to "CHAR". For example, on UNIX™:

```plaintext
$ FGL_LENGTH_SEMANTICS="CHAR"
$ export FGL_LENGTH_SEMANTICS
```
Length Semantics in SQL

On the database server side, the length semantics used for character data types varies from one vendor to another. Some databases use BLS, other use CLS, and other support both semantics.

**Important:** The length semantics used by the Genero runtime (defined by `FGL_LENGTH_SEMANTICS`) should match the length semantics used by the database server: If a database column is defined as a CHAR of 10 bytes, the corresponding BDL program variable should also be defined as a CHAR of 10 bytes (using `FGL_LENGTH_SEMANTICS=BYTE`, the default). If the database column is defined as a CHAR of 10 characters, the variable should be defined as a CHAR of 10 characters (using `FGL_LENGTH_SEMANTICS=CHAR`).

This table shows the character data type length semantics of supported database servers:

**Table 154: Character data type length semantics of supported database servers**

<table>
<thead>
<tr>
<th>Database Engine</th>
<th>Length semantics in character data types</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle®</td>
<td>Supports both Byte or Character Length Semantics in character type definition, can be defined globally for the database or at column level (with the `CHAR(10 BYTE</td>
<td>CHAR)` syntax). Character string data is stored in database character set for CHAR / VARCHAR columns and in national character set for NCHAR / NVARCHAR columns. See Oracle DB Guide for more details.</td>
</tr>
<tr>
<td>Informix®</td>
<td>Uses Byte Length Semantics for the size of character columns. Can apply a ratio when creating columns, based on the <code>SQL_LOGICAL_CHAR</code> server configuration parameter. Character string data is stored in the database character set defined by <code>DB_LOCALE</code>.</td>
<td>BLS</td>
</tr>
<tr>
<td>IBM® DB2®</td>
<td>Uses Byte Length Semantics for the size of character columns. Character data is stored in the database character set defined by the <code>CODESET</code> of <code>CREATE DATABASE</code>. See DB2 LUW Guide for more details.</td>
<td>BLS</td>
</tr>
<tr>
<td>Database Engine</td>
<td>Length semantics in character data types</td>
<td>Summary</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>Microsoft® SQL Server</td>
<td>CHAR / VARCHAR sizes are specified in bytes; data is stored in the character set defined by the database collation.</td>
<td>BLS/CLS</td>
</tr>
<tr>
<td></td>
<td>NCHAR / NVARCHAR sizes are specified in characters; data is stored in UCS-2.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Microsoft SQL Server Guide for more details.</td>
<td></td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>Uses Character Length Semantics for the size of character columns.</td>
<td>CLS</td>
</tr>
<tr>
<td></td>
<td>Character string data is stored in the database character set defined by WITH ENCODING of CREATE DATABASE.</td>
<td></td>
</tr>
<tr>
<td>Oracle® MySQL / Maria DB</td>
<td>Uses Character Length Semantics for the size of character columns.</td>
<td>CLS</td>
</tr>
<tr>
<td></td>
<td>Character string data is stored in the server character set defined by a configuration parameter.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See Oracle MySQL / Mariadb Guide for more details.</td>
<td></td>
</tr>
<tr>
<td>SQLite</td>
<td>Uses Character Length Semantics for the size of character columns.</td>
<td>CLS</td>
</tr>
<tr>
<td></td>
<td>Character string data is stored in UTF-8.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See SQLite Guide for more details.</td>
<td></td>
</tr>
<tr>
<td>SAP Adaptive Server Enterprise</td>
<td>CHAR / VARCHAR sizes are specified in bytes; data is stored in the db character set.</td>
<td>BLS/CLS</td>
</tr>
<tr>
<td></td>
<td>NCHAR / NVARCHAR sizes are specified in characters; data is stored in the db character set.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>UNICHAR / UNIVARCHAR sizes are specified in characters; data is stored in UTF-16.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See SAP ASE Guide for more details.</td>
<td></td>
</tr>
</tbody>
</table>
### Advanced features

<table>
<thead>
<tr>
<th>Database Engine</th>
<th>Length semantics in character data types</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAP HANA</td>
<td>VARCHAR sizes are specified in bytes; only for ASCII-7 character strings. NVARCHAR sizes are specified in characters; for UNICODE character strings. CHAR/NCHAR are not supported by SAP HANA.</td>
<td>BLS (ASCII-7) / CLS (UNICODE)</td>
</tr>
</tbody>
</table>

Other SQL elements like functions and operators are affected by the length semantic. For example, Informix® LENGTH() function always returns a number of bytes, while Oracle's LENGTH() function returns a number of characters (use LENGTHB() to get the number of bytes with Oracle).

It is important to understand properly how the database servers handle multibyte character sets. Check your database server reference manual. In most documentations you will find a "Localization" chapter which describes those concepts in detail.

### Extracting database schemas

Database schema files (.sch) are used to resolve column data types when compiling .4gl modules and .per form files. This file contains size information for CHAR and VARCHAR types. It is important to identify the unit used by the database columns, to properly define CHAR/VARCHAR variables in programs and fields in forms.

Most database engines (like Oracle DB, SQL Server, PostgreSQL, SQLite) provide catalog tables with column size information in character units. In this case, the fgldbsch tool extracts the column sizes in character units, without further conversion. If the column sizes is provided in bytes by catalog tables, fgldbsch will try to detect character length semantic usage in the database and apply a reduction factor to convert the number of bytes to chars.

For example, with Informix®, when using the SQL_LOGICAL_CHAR onconfig parameter, fgldbsch will convert the size stored in bytes in syscolumns.collength column, to a number of characters, by dividing that number of bytes by the value of SQL_LOGICAL_CHAR.

As a result - independently from the length semantics used in your programs - the CHAR/VARCHAR type sizes in the schema file are always expressed in character units. When using Byte Length Semantics, this makes no difference in a single-byte locale, because one character occupies a single byte. In a multibyte encoding (UTF-8) with BLS, this method guarantees that the program variable will not hold more ASCII characters than the database column can hold. When using Character Length Semantics with a multibyte character set, the size in characters will define character type variables in the same unit.

For example, with BLS, a VARCHAR(10 (bytes or chars)) column will define a VARCHAR(10 (bytes)) in programs. With CLS, a VARCHAR(10 (chars)) column will define a VARCHAR(10 (chars)) in programs.

### Moving from single-byte to UTF-8

Migration to Unicode (UTF-8) is facilitated with Char Length Semantics:

1. Verify that your database uses Char Length Semantics.
2. Convert your sources and string files from your single-byte locale to UTF-8 (iconv).
3. Enable Char Length Semantics with FGL_LENGTH_SEMANTICS=CHAR.
4. Compile and run your programs untouched.

### Related concepts

- SQL character type for Unicode/UTF-8 on page 617
This section explains database server specifics regarding Unicode / UTF-8 support with character string SQL types.

**Multibyte character sets (MBCS)** on page 517
**Single-byte character sets (SBCS)** on page 517
**Character size unit and length semantics** on page 517
**Manipulating character strings** on page 588

### Collation ordering settings

The runtime system supports a sorting functionality in tables. To sort the data rows, the runtime systems uses the standard C library functions to order character strings.

The environment variable LC_COLLATE can be used to control sort order in Genero. You can for example define this variable as "C" or "POSIX" to get a binary sort order.

When using LC_COLLATE, set the LANG environment variable to define the global locale, if you use LC_ALL, it will overwrite all other LC_* variables defined.

**Related concepts**

**LC_ALL (or LANG)** on page 264
Defines the current application locale on UNIX™ platforms.

### Numeric and currency locale settings

The environment variables LC_MONETARY and LC_NUMERIC are ignored.

To perform decimal to/from string conversions, the runtime system uses the DBMONEY or DBFORMAT environment variables. These variables define hundreds / decimal separators and currency symbols for MONEY data types.

**Related concepts**

**DBFORMAT** on page 268
Defines the characters to be used for the currency symbol, decimal and thousands separators for numeric values.

**DBMONEY** on page 271
Defines the characters to be used for the currency symbol and decimal separator for numeric values, when DBFORMAT is not defined.

### Date and time locale settings

The environment variable LC_TIME is ignored.

To perform date to/from string conversions, the runtime system uses by default the DBDATE environment variable.

**Related concepts**

**Date, numeric and monetary formats** on page 532
This section describes how Genero BDL handles date, time, numeric and monetary formats.

### Database client settings

This section describes the settings defining the locale for the database client.

Each database software has its own client character set configuration.

**Important:** In order to properly send/receive character string data to/from the database server, the database client locale and application locale settings must match.
<table>
<thead>
<tr>
<th>Database Client</th>
<th>Locale settings</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle® database server</td>
<td>The database client locale must be set with NLS_LANG environment variable.</td>
<td>By default, the client locale is set from the database server locale. The locale can also be defined after connection, with the ALTER SESSION instruction. However, this is not supported by Genero BDL.</td>
</tr>
<tr>
<td>IBM® Informix®</td>
<td>The database client locale is defined by the CLIENT_LOCALE environment variable, and DB_LOCALE must be set to the database locale.</td>
<td>If CLIENT_LOCALE is not defined, other settings are used when defined (DBDATE / DBTIME / GL_DATE / GL_DATE TIME, as well as standard LC_* variables).</td>
</tr>
<tr>
<td>IBM® DB2®</td>
<td>The database client locale is defined by the DB2CODEPAGE profile variable.</td>
<td>The DB2CODEPAGE variable can be set with the db2set command. However, you usually do not need to set this variable: If DB2CODEPAGE is not defined, DB2® uses the operating system code page on Windows® and the LANG/ LC_ALL locate setting on UNIX™. When using a UTF-8 locale on Windows®, DB2CODEPAGE must be set to 1208.</td>
</tr>
<tr>
<td>IBM® Netezza®</td>
<td>With IBM® Netezza®, there is no configuration setting to define database client locale.</td>
<td>When using CHAR/VARCHAR columns, the application character set (LC_ALL, LANG) must match the Latin-9 / ISO-8859-15 character set. When using NCHAR/NVARCHAR columns, the application character set must be UTF-8.</td>
</tr>
<tr>
<td>Microsoft® SQL Server with SNC driver (Microsoft® ODBC)</td>
<td>On Windows® platforms, the database client locale is defined by the language settings for non-Unicode applications. The current ANSI code page (ACP) is used by the SQL Server client and the Genero runtime system. On Linux® platforms, the database client locale is always UTF-16. The ODI driver uses the MS ODBC Wide Char API, and makes the required character set conversions between the application locale and UTF-16. See Microsoft® ODBC documentation for more details regarding character set configuration.</td>
<td></td>
</tr>
<tr>
<td>Microsoft® SQL Server with FTM driver (FreeTDS)</td>
<td>The database client character set is defined by the client charset parameter in freedts.conf, or with the ClientCharset parameter in the DSN of the odbc.ini file. See FreeTDS documentation for more details regarding character set configuration.</td>
<td></td>
</tr>
<tr>
<td>Database Client</td>
<td>Locale settings</td>
<td>Notes</td>
</tr>
<tr>
<td>----------------</td>
<td>----------------</td>
<td>-------</td>
</tr>
</tbody>
</table>
| Microsoft® SQL Server with ESM driver (Easysoft). | The database client character set is defined by the \texttt{Client\_CSet} parameter in the DSN of the odbc.ini file. Depending on the application locale and SQL Server CHAR/VARCHAR or NCHAR/NVARCHAR usage, you might also need to define the \texttt{Server\_CSet} and/or the \texttt{Server\_UCSet} parameters. | To support all UNICODE characters when using UTF-8 with NCHAR/NVARCHAR columns, you need to define \texttt{Client\_CSet=\\texttt{UTF-8}} and \texttt{Server\_UCSet=\\texttt{UTF-16LE}}. When using CHAR/VARCHAR types in the database and when the database collation is different from the client locale, you must also set the \texttt{Server\_CSet} parameter to an iconv name corresponding to the database collation. Some examples:  
  - If \texttt{Client\_CSet=ISO-8859-15} and the db collation is \texttt{Latin1\_*} (=CP1252), you must set \texttt{Server\_CSet=WINDOWS-1252} (otherwise, the characters €, Š, š, Ž, ž, Œ, œ, Ÿ which are encoded differently  
  - If \texttt{Client\_CSet=BIG5} and the db collation is \texttt{Chinese_Taiwan\_Stroke\_BIN}, you must set \texttt{Server\_CSet=BIG5HKSCS}. |
| PostgreSQL | The database client locale must be set with the \texttt{PGCLIENTENCODING} environment variable, or with the \texttt{client\_encoding} configuration parameter in \texttt{postgresql.conf}. | After the database connection, the locale can be set with the \texttt{SET CLIENT\_ENCODING} instruction. This is not recommended with Genero BDL. Check the \texttt{pg\_conversion} system table for available character set conversions. |
| Oracle® MySQL and MariaDB | The database client locale is defined by the \texttt{default\_character\_set} option in the MySQL configuration file. | MySQL/MariaDB support the \texttt{SET NAMES} SQL instruction, to change the character set after connecting to the database. However, this is not supported with Genero: The driver needs to know the character set at connection initialization. Use the \texttt{default\_character\_set} configuration option. |
| SAP ASE | By default, the database client character set is defined by the operating system locale where the database client runs. On Windows®, it is the ANSI code page of the login session (can be overwritten by setting the \texttt{LANG} environment variable). On UNIX™ it is defined by the \texttt{LC\_CTYPE}, \texttt{LC\_ALL} or \texttt{LANG} environment variables. | You may need to edit the $\texttt{SYBASE}/locales/locales.dat file, to map the OS locale name to a known ASE character set. See ASE ODBC documentation for more details regarding character set configuration. |
No database client locale configuration is required with SAP HANA: The database driver makes the appropriate charset conversions when needed. CHAR/NCHAR are not supported by SAP HANA. VARCHAR columns store only ASCII-7 characters and therefore can only be used with an ASCII/POSIX application locale. NVARCHAR columns store UNICODE characters, and can be used with a single-byte or UTF-8 application locale.

No database client locale configuration is required with SQLite: The database driver makes the appropriate charset conversions when needed. SQLite databases use UTF-8 encoding. If the locale used by the runtime system (LANG/LC_ALL) is not UTF-8, Genero will do the appropriate character set conversions.

SQLite character type for Unicode/UTF-8 on page 617
This section explains database server specifics regarding Unicode / UTF-8 support with character string SQL types.

Front-end locale configuration
The host operating system on the front-end workstation must be able to handle the character set and fonts.

For instance, a Western-European Windows® is not configured to handle Arabic applications. If you start an Arabic application, some graphical problems may occur (for instance the title bar won't display Arabic characters, but unwanted characters instead).

The GUI front-end software must support the conversion of the runtime system character set to/from the character set used internally by the client, and must be configured with the correct font to display the characters used by the application. For example, the default font for a front-end installed on an English Windows® system might not be able to display Japanese characters. You must then change the font in the front-end configuration panel. Refer to the front-end documentation to see how character set conversion and fonts can be configured.

When using a TUI program in a terminal emulator such as Putty, XTerm or even the Windows® Console, make sure the terminal is configured properly to display the characters of the application locale. For example, on a Windows® Console you can use the chcp command to change the current code page.

Locale matters when writing programs
The language locale used when writing source code defines the runtime locale, except when developing in ASCII.

Development and runtime character set must match
When writing a form or program source file, you use a specific character set. This character set depends upon the text editor or operating system settings you are using on the development platform. For example, when writing a string constant in a .4gl module, containing Arabic characters, you probably use the ISO-8859-6 character set. The character set used at runtime (during program execution) must match the character set used to write programs.

At runtime, a Genero program can only work in a specific character set. However, by using localized strings, you can start multiple instances of the same compiled program using different locales. For a given program instance the character set used by the strings resource files must correspond to the locale. Make sure the string identifiers use ASCII only.

Byte length semantics and substring expressions
When using Byte Length Semantics (BLS), all character positions in strings are actually byte positions. In a multibyte environment, if you don't pay attention to this, you can end up with invalid characters in strings. For example, an expression using a subscript operator \([x, y]\) might refer to a byte position which is in fact in the middle of
a multibyte character. If possible, use Character Length Semantics (CLS) with a multibyte locale to avoid such problems, or use only STRING methods to parse character strings.

**Related concepts**
- Defining the application locale on page 519
- Length semantics settings on page 521
- Language and character set settings on page 519
- Localized strings on page 538

**Localized strings** provide a means of writing applications in which the text of strings can be customized on site.

**Runtime system messages**
This section describes how to translate default English .msg message files in a different language.

Runtime system error messages are provided in .iem message files. The system message files use the same technique as user defined message files. The default message files (.msg) are located in the $FGLDIR/msg/en_US directory.

For backward compatibility with IBM® Informix® 4GL, some of these system error messages are used by the runtime system to display messages during a dialog instruction. For example, end users may get the error message -1309 "There are no more rows in the direction you are going" when scrolling an a DISPLAY ARRAY list in TUI mode.

If your application language is not English, you will need to translate some of the system messages to a specific locale and language. If your application language is English, you might just want to customize the default messages.

Here are some examples of system messages that can appear at runtime:

**Table 156: Examples (subset) of system messages for localized strings**

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1204</td>
<td>Invalid year in date.</td>
</tr>
<tr>
<td>-1304</td>
<td>Error in field.</td>
</tr>
<tr>
<td>-1305</td>
<td>This field requires an entered value.</td>
</tr>
<tr>
<td>-1306</td>
<td>Please type again for verification.</td>
</tr>
<tr>
<td>-1307</td>
<td>Cannot insert another row - the input array is full.</td>
</tr>
<tr>
<td>-1309</td>
<td>There are no more rows in the direction you are going.</td>
</tr>
</tbody>
</table>

To use your own customized system messages, do this:

1. Create a new directory under $FGLDIR/msg, using the same name as your current locale. For example, if LANG=fr_FR.ISO8859-1, you must create $FGLDIR/msg/fr_FR.ISO8859-1.
2. Copy the original system message source files (.msg) from $FGLDIR/msg/en_US to the locale-specific directory.
3. Edit the source files with the .msg suffix and translate the messages.
4. Recompile the message files with the fglmkmsg tool to produce .iem files. Make sure you have set the correct locale!
5. Run a program to check if the new messages are used.

With this technique, you can deploy multiple message files in different languages and locales in the same FGLDIR/msg directory.

You can use the fglmkmsg tool with the -r option to revert a .iem file to a source .msg file.
There is no need to translate all messages of the .msg files: Most of the error messages are unexpected during a program execution and therefore can stay in English. The messages subject of translation can be found in the 4gusr.msg and rds.msg files.

The locale can be set with different environment variables (see setlocale manual pages for more details). To identify the locale name, the runtime system first looks for the LC_ALL value, then LC_CTYPE and finally LANG.

Pay attention to locale settings when editing message files and compiling with fglmkmsg: The current locale must match the locale used in the .msg files.

The .iem files used at runtime must match the current locale used by programs. This is automatic, as long as you put the correct files in the corresponding $FGLDIR/msg/$LANG directory.

Related concepts
Message files on page 1160
Message files centralize strings and larger texts identified by a number, that can be used in programs.

Using the charmap.alias file
The charmap.alias file can be used to map a system specific locale to a standard IANA locale.

The name of the character set defined within the LANG/LC_ALL environment variables can vary from system to system. For example, on a given platform, the ISO-8859-1 character set may be named "iso88591", while others platform will use "8859-1".

Note: On UNIX™-like systems, installed locales can be listed with the locale -a command.

Example of locale configuration on HP/UX:

```
$ export LANG=en_US.iso88591
$ locale
LANG=en_US.iso88591
LC_CTYPE="en_US.iso88591"
LC_COLLATE="en_US.iso88591"
LC_MONETARY="en_US.iso88591"
LC_NUMERIC="en_US.iso88591"
LC_TIME="en_US.iso88591"
LC_MESSAGES="en_US.iso88591"
LC_ALL=
$ locale charmap
"iso88591.cm"
```

To communicate with other components like front-ends, or identify the encoding of XML files, Genero programs must use a normalized name for character sets. This normalized name must follow the IANA specifications [RFC2978].

In order to convert the operating system specific locale codeset name to an IANA name, the runtime system uses the charmap.alias mapping file, located in $FGLDIR/etc. Add your operating system specific locale, if not listed in the charmap.alias file.

The charmap.alias file has the following syntax:

```
system-codeset-name  IANA-codeset-name
```

Note: Comment lines can be added by using a # hash character at the beginning of the line.

Example of charmap.alias file:

```
# Linux
ISO_646.IRV:1983 ASCII
ANSI_X3.4-1968 ASCII
# Mac OS X
US-ASCII ASCII
```
**Date, numeric and monetary formats**

This section describes how Genero BDL handles date, time, numeric and monetary formats.

Dates, numbers and monetary values must be displayed and entered in a format used in the country/region. These formats can be defined with the `DBDATE` and `DBFORMAT` environment variables.

Date and numeric format settings matter for data display and data input. For example, when displaying a `DATE` value to a form field, it will implicitly be formatted by the `DBDATE` setting. When the user enters a date in a form field bound to a `DATE` variable, the entered digits will be interpreted by the `DBDATE` format.

The default values of these environment variables depend on the type of platform where the program executes:

When using the `FORMAT` field attribute or the `USING` operator to format dates with abbreviated day and month names - by using `dd/MM/yyyy` markers - the system uses English-language based texts for the conversion. This means, day (`dd`) and month (`MM`) abbreviations are not localized depending on the locale settings, they will always be in English.

- On desktop/server platforms, the default formats are set for the United States of America:
  - Dates are formatted as `mm/dd/yyyy`.
  - The decimal separator is a dot.
  - The currency symbol is the dollar sign (\$).

- On mobile platforms, the default formats are set from the regional settings defined on the device.
  - Dates are formatted depend on the regional settings.
  - The decimal separator is defined depend on the regional settings.
  - The currency symbol is not defined. No currency symbol will display.

**Note:** While it is possible to define environment settings for date and numeric formats with `FGLPROFILE` entries, it is strongly recommended to leave the defaults, so as to get the expected formats if the user changes the regional settings on the mobile device.

**Related concepts**

- Formatting `DATE` values on page 321
- Formatting `DATETIME` values on page 322
- Formatting numeric values on page 318

**Using the Ming Guo date format**

Genero BDL can be configured to use the The Ming Guo calendar.

The Ming Guo (or Minguo) calendar is still used in some Asian regions like Taiwan. This calendar is equivalent to the Gregorian calendar, except that the years are numbered with a different base. In the Ming Guo calendar, the first year (1) corresponds to the Gregorian year 1912, the year the Republic of China was founded.
Digit-based year Ming Guo date format can be enabled by adding the C1 modifier at the end of the value set for the DBDATE environment variable:

```bash
$ DBDATE="Y3MD/C1"
$ export DBDATE
```

With this DBDATE setting, dates will be displayed with a year following the Ming Guo calendar, and date input will also be interpreted based on that calendar. For example, if the user enters 90/3/24, it is equivalent to an input of 2002/3/24 when using the Gregorian calendar. Basically, the runtime system will subtract 1912 or add 1912 respectively when displaying or reading date values).

When using the C1 modifier, the possible values for the Yn symbol are Y4, Y3, Y2.

The MDY() operator is sensitive to the C1 modifier usage in DBDATE. For example, if DBDATE=Y3MD/C1, MDY(3,24,1) will build a date the corresponds in the Gregorian to MDY(3,24,1912).

The USING operator supports the c1 modifier as well. The c1 modifier must be specified at the end of the format. You can for example use the following format string: "yyy-mm-ddc1".

The C2 modifier to use Era names is not supported.

Unlike Informix® 4gl, when using negative years, the minus sign is placed over the left-most zero of the year, to avoid miss-aligned dates.

For example, if DBDATE=Y3MD/C1:

```
MDY(3,2, 1) USING "yyy/mm/ddc1"
MDY(3,2,-1) USING "yyy/mm/ddc1"
```

Will align properly as follows:

```
0001/03/02
-001/03/02
```

**Note:** Front-ends may not support the Ming Guo calendar for widgets like DATEEDIT.

**Related concepts**

Date, numeric and monetary formats on page 532

This section describes how Genero BDL handles date, time, numeric and monetary formats.

**User's preferred language**

An application can get the user's preferred language and territory as configured on the front-end platform.

The user preferred language can, for example, be useful in selecting appropriate content based on language preferences, and in starting other programs by setting the expected application locale.

To get the user preferred language as defined in the front-end, perform a standard.feInfo front call with the userPreferredLang option:

```plaintext
PRIVATE DEFINE fe_lang STRING

FUNCTION get_fe_lang()
    IF fe_lang IS NULL THEN
        CALL ui.Interface.frontCall( "standard", "feInfo", ["userPreferredLang"], [fe_lang] )
    END IF
    RETURN fe_lang
END FUNCTION
```

The front-end locale configuration depends on the type of front-end:

- For Genero Browser Client (GBC), the front-end locale is defined in the web browser preferences.
• For Genero Desktop Client (GDC), by default the front-end locale is defined by the operating system language settings. It can be configured with a GDC option.
• For Genero Mobile for Android (GMA) and Genero Mobile for iOS (GMI), the front-end locale is defined by the device language settings.

The format of the returned value is:

```
language_territory
```

For example, when running the GDC front-end on a Linux® platform with LC_ALL defined as `en_US.utf8`, the front call will return:

```
en_US
```

**Related concepts**

- `feInfo` on page 2651
  Queries general front-end properties.

**Right-to-left languages support**

Genero supports right-to-left languages, such as Arabic and Hebrew.

**Right-to-left mode**

For specific front-end clients, Genero supports right-to-left languages with the reverse mode. With reverse mode, all forms are mirrored and the text direction changes to right-to-left, for display and input.

The reverse mode is enabled at runtime. The same form files are used to display in the default left-to-right and right-to-left mode.

**Note:** Right-to-left display is implicit on mobile devices. It is enabled depending on language settings on the device.

**Application locale**

Reverse mode is used with an application locale that defines a language written from right to left.

For example, for Arabic support on a Linux® platform, you can use the following LC_ALL value when using the ISO-8859-6 codeset:

```
$ export LC_ALL=ar_DZ.iso88596
```

**Reverse mode configuration**

To enable reverse mode, set the `reverse` style attribute for the `UserInterface` class to "yes" in your .4st file:

```
<StyleList>
  ...
  <Style name="UserInterface">
    <StyleAttribute name="reverse" value="yes" />
  </Style>
  ...
</StyleList>
```

**Related reference**

- `UserInterface style attributes` on page 1219
User Interface presentation style attributes define general options related to the application user interface.

**Troubleshooting locale issues**
This section describes common issues related to language locale definition.

**Locale settings (LANG) corrupted on Microsoft™ platforms**

On Microsoft™ Windows® XP / 2000 platforms, some system updates (Services Pack 2) or Office versions do set the LANG environment variable with a value for Microsoft™ applications (for example 1033).

Such value is not recognized by Genero as a valid locale specification.

Make sure that the LANG environment variable is properly set in the context of Genero applications.

**Related concepts**
- What is my current character set? on page 519
- Language and character set settings on page 519

**A form is displayed with invalid characters**

You may have different code sets on the development machine and the production server.

The typical mistake that can happen is the following: You have edited and compiled a form-file on your Windows® development workstation with the CP1253 encoding; When using the .42f form-file on a UNIX-server with encoding ISO-8859-7, invalid characters will appear.

Keep in mind that all source files must be created/edited in the encoding of the server (where fglcomp and fglrun will be executed). Consider writing your sources in pure ASCII-7, and put language/codeset specific messages in localized strings.

**Related concepts**
- Understanding locale settings on page 513
- Language and character set settings on page 519
- Compiling source files on page 2106

**Checking the locale configuration on UNIX™ platforms**

On UNIX™ systems, the `locale` command without parameters outputs information about the current locale environment.

The locale is typically set with the LANG or, preferably with the LC_ALL environment variable. When the LC_ALL variable is defined, LANG is ignored:

```bash
$ export LANG=en_US.ISO8859-1
$ unset LC_ALL

$ locale
LANG=en_US.ISO8859-1
LANGUAGE=
LC_CTYPE="en_US.ISO8859-1"
LC_NUMERIC=en_US.UTF-8
LC_TIME=en_US.UTF-8
LC_COLLATE="en_US.ISO8859-1"
LC_MONETARY=en_US.UTF-8
LC_MESSAGES="en_US.ISO8859-1"
LC_PAPER=en_US.UTF-8
LC_NAME="en_US.ISO8859-1"
LC_ADDRESS="en_US.ISO8859-1"
LC_TELEPHONE="en_US.ISO8859-1"
LC_MEASUREMENT=en_US.UTF-8
LC_IDENTIFICATION="en_US.ISO8859-1"
```
LC_ALL=

$ export LC_ALL=POSIX

LANG=en_US.ISO8859-1
LANGUAGE=
LC_CTYPE="POSIX"
LC_NUMERIC="POSIX"
LC_TIME="POSIX"
LC_COLLATE="POSIX"
LC_MONETARY="POSIX"
LC_MESSAGES="POSIX"
LC_PAPER="POSIX"
LC_NAME="POSIX"
LC_ADDRESS="POSIX"
LC_TELEPHONE="POSIX"
LC_MEASUREMENT="POSIX"
LC_IDENTIFICATION="POSIX"
LC_ALL=POSIX

Use the fglrun -i command to check current locale settings for FGL:

$ fglrun -i
Charmap      : ASCII
Multibyte    : no
Stateless    : yes
Length Semantics : BYTE

Here the charset used is the ASCII (POSIX) charset. Clearing the LC_ALL environment variable and setting the LC_CTYPE to a different value as LANG gives the following:

$ unset LC_ALL
$ export LC_CTYPE=zh_TW.big5

$ locale
LANG=fr_FR.ISO-8859-1
LANGUAGE=
LC_CTYPE=zh_TW.big5
LC_NUMERIC=en_US.UTF-8
LC_TIME=en_US.UTF-8
LC_COLLATE="fr_FR.ISO-8859-1"
LC_MONETARY=en_US.UTF-8
LC_MESSAGES="fr_FR.ISO-8859-1"
LC_PAPER=en_US.UTF-8
LC_NAME="fr_FR.ISO-8859-1"
LC_ADDRESS="fr_FR.ISO-8859-1"
LC_TELEPHONE="fr_FR.ISO-8859-1"
LC_MEASUREMENT=en_US.UTF-8
LC_IDENTIFICATION="fr_FR.ISO-8859-1"
LC_ALL=

$ fglrun -i
Charmap      : BIG5
Multibyte    : yes
Stateless    : yes
Length Semantics : BYTE

If the current locale settings is not supported by FGL, you get the following output:

$ fglrun -i
Error: locale not supported by C library, check LANG/LC_ALL.
**Tip:** If the locale environment is not correct, check the value of the following environment variables: LANG, LC_ALL, LC_CTYPE, LC_NUMERIC, LC_TIME, LC_COLLATE, etc.

The next example shows how to define a UTF-8 character set with character length semantics (FGL_LENGTH_SEMANTICS):

```bash
$ export LC_ALL=en_US.UTF-8
$ export FGL_LENGTH_SEMANTICS=CHAR
$ fglrun -i
Charmap      : UTF-8
Multibyte    : yes
Stateless    : yes
Length Semantics : CHAR
```

### Related concepts

**Operating system environment variables** on page 264

Describes some well-known system environment variables that are used by Genero software components.

**Verifying if the locale is properly supported by the runtime system**

Check the current LANG/LC_ALL locale settings by using the `-i` option of `fglrun`:

```bash
$ fglrun -i
Charmap          : UTF-8
Multibyte        : yes
Stateless        : yes
Length Semantics : CHAR
```

The lines printed with this option indicate if the locale can be supported by the operating system libraries.

If the locale settings are wrong or unsupported, the command will display an error:

```bash
$ fglrun -i
Error: locale not supported by C library, check LANG/LC_ALL.
```

### Table 157: Locale information descriptions

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Charmap</td>
<td>This is the normalized IANA name of the character set used by the runtime system to communicate with external components (front-end, I/O of XML files). The mapping from the system locale name to a normalized name is defined in <code>$FGLDIR/etc/charmap.alias</code>.</td>
</tr>
<tr>
<td>Multibyte</td>
<td>This line indicates if the character set is a MBCS (Multi-Byte Character Set) and an SBCS Single Byte Character Set.</td>
</tr>
<tr>
<td>Stateless</td>
<td>A few character sets use an internal state that can change during the character flow. Only stateless character sets can be supported (the value must be 'yes').</td>
</tr>
<tr>
<td>Length Semantics</td>
<td>BYTE indicates Byte Length Semantics, while CHAR indicates character length semantics. See Length semantics settings on page 521 for more details.</td>
</tr>
</tbody>
</table>

### How to retrieve the list of available locales on the system

On UNIX™ systems, the `locale -a` command gives the names of available locales:

```bash
$ locale -a
```
How to retrieve the list of available codesets on the system

On UNIX™ systems, the `locale -m` command gives the names of available codesets:

```
$ locale -m
...
ISO-8859-1
ISO-8859-10
ISO-8859-13
ISO-8859-14
ISO-8859-15
...
```

Localized strings

Localized strings provide a means of writing applications in which the text of strings can be customized on site.

This string localization feature is a simple way to define external resource files which the runtime system can search, in order to assign text to elements displayed by programs. It can be used to implement internationalization in your application, or to use site-specific text, for example, when business terms are specific to the territory where the application is used.

The localized string resource files (.42s) are loaded at runtime and shared by all `fglrun` processes. Localized strings are used to replace the original strings found in the p-code modules (.42m), in the compiled form (.42f), and in any XML resource files loaded in the abstract user interface tree (.4ad,.4st,.4tb, etc).

Related concepts

Form specification files on page 1237
Form specification files are the source files defining the layout and content of application forms.

The abstract user interface tree on page 1106
The abstract user interface tree is the XML representation of the application forms displayed to the end user.

Steps for application internationalization

Follow these steps to internationalize your application.

1. Identify the current character set used in your sources and make sure the application locale (LANG/LC_ALL) is set correctly.
2. In .4gl sources, add a `%` prefix to the strings that must be localized (translated). For parameterized messages, replace concatenated strings by a `Sfmt()` usage with `%n` placeholders for variable message parts.
3. In .per sources LAYOUT section, replace hard-coded form elements like text labels with static `LABEL` form items and define the `TEXT` attributes with a `%` prefix in the ATTRIBUTES section.
4. In XML resources, add `<LStr />` elements under the elements where text attributes must be localized.
5. Extract the strings from the .4gl sources with `fglcomp -m` and use `fglform -m` for .per sources.
6. Organize the generated .str source string files (identify duplicated strings and put them in a common file).
7. At this point, the string identifiers (on the left) are the same as the string texts (on the right). These string identifiers can be used as is, or can be changed to clear ASCII identifiers such as "customer.list.title". Using simple identifiers allows you to distinguish strings depending on the context and use ASCII encoding for your sources. Keeping string identifiers with the original text requires no source changes (except adding the `%` prefix), but makes sources dependent to a locale: If you want to support multiple languages, you must use UTF-8 in sources and at runtime.
8. When using simple ASCII identifiers, replace original strings with the new string identifiers. Strings to be replaced can be located by their % prefix. You can, for example, use a script with an utility like the sed UNIX™ command to read the .str files and apply the changes automatically.

9. Recompile the .4gl and .per sources (when using simple ASCII strings identifiers, sources are expected to be full ASCII now).

10. Compile the .str files in the locale used by these files.

11. Setup FGLPROFILE fglrun.localization.* entries, to let fglrun find the string resource files.

12. Run your programs to check whether the application displays the text properly.

13. Copy the existing .str files, and translate the string text into another language (make sure the locale is correct).

14. Compile the new .str files, and copy the .42s files into another distribution directory, defined with the FGLRESOURCEPATH environment variable.

15. Run your programs again, to check that texts and labels of the other language are displayed.

16. Next changes to the .per and .4gl source files are done in the ASCII locale, and .str string files must be edited with their specific locale.

Related concepts

Application locale on page 512
The application locale defines the language and codeset for your application.

Creating source string files

A source string file contains localized string definitions for a given language (or localization context).

What is a source string file?

A source string file is basically a mapping table that defines an identifier for each string.

After compiling source string files, programs can load and use a given string by referencing its identifier (or key).

By convention, the source files of localized strings have the .str extension.

Syntax

Define a list of string identifiers, and the corresponding text, by using the following syntax:

"string-identifier" = "string-text"

For example:

"Cancel" = "Annuler"

Note: Localized string keys are case sensitive. Consider using lower case characters only to avoid mistakes.

As an alternative, you can define string identifiers as a dot-separated list of identifiers:

identifier. [...] = "string-text"

For example:

common.button.cancel = "Annuler"

If needed, you can add comment lines with the # or -- markers, like in other Genero source files:

# a comment
-- another comment
Special characters

The fglmkstr compiler accepts the backslash "\" as the escape character, to define non-printable characters:

```
\l  \n  \r  \t  \n
```

Example

```
# A comment line
"Original text" = "Original text"
forms.customer.list = "Customer List"
special.characters.backslash = "\\"
special.characters.newline = "\n"
```

Related concepts

Extracting strings from sources on page 542
Localized strings can be easily extracted from .4gl and .per source files.

Localized strings in program sources

How to specify a localized string in .4gl and .per sources?

Defining localed strings in programs

A localized string is specified in the source code of program modules or form specification files with the %"string" notation, to identify the string that must be replaced at runtime by the corresponding text found in compiled string files.

In programs, localized strings can also be loaded dynamically with the LSTR() operator.

Syntax 1: Static localized string

```
%"sid"
```

1. **sid** is a character string literal that defines both the string identifier and the default text.

Syntax 2: Dynamic localized string

```
LSTR(eid)
```

1. **eid** is a character string expression used at runtime as the string identifier to load the text.

Static localized strings

A static localized string specification begins with a percent sign (%), followed by the identifier of the string which will be used to find the text to be loaded. Since the identifier is a string, you can use any type of characters, but it is recommended that you use a naming convention. For example, you can specify a path by using several names separated by a dot:

```
MAIN
  DISPLAY %"common.message.welcome"
END MAIN
```

The string after the percent sign defines both the localized string identifier and the default text to be used for extraction, or the default text when no string resource files are provided at runtime.

You can use this notation in form specification files any place where a string literal can be used.
It is not possible to specify a static localized string directly in the area of containers like GRID, TABLE, TREE or SCROLLGRID. Use static LABEL form items to define localized strings in layout labels:

```
LAYOUT
  GRID
  { [lab01 |f001 ]
  } END
END
ATTRIBUTES
LABEL lab01: TEXT=%"myform.label01";
EDIT f001 = FORMONLY.field01;
END
```

**Dynamic localized strings**

The language provides a special operator to load a localized string dynamically, using an expression as string identifier.

The name of this operator is `LSTR()`.

The following code example builds a localized string identifier with an integer and loads the corresponding string with the `LSTR()` operator:

```
MAIN
  DEFINE n INTEGER
  LET n = 234
  DISPLAY LSTR("str\"|n) -- loads string 'str234'
END MAIN
```

**Related concepts**

- **Text literals** on page 326
  Text literals define a character string in an expression.

- **Localized strings in XML resource files**
  In XML resource files, localized string specification must follow the XML syntax and therefore must be defined as an XML node.

**Syntax: Localized string in XML files**

```
<ParentNode attribute = "default" ... />
<LStr attribute = "sid" ... /></ParentNode>
```

1. `ParentNode` is the node type of the parent where the localized strings must be applied.
2. `attribute` is the attribute in the parent node that will get the localized string identified by `sid`.
3. `default` is the default text of an attribute, if not localized string is found for `sid`.
4. `sid` is a character string literal that defines both the string identifier and the default text.

**Description**

In .42m p-code modules, the localized strings are coded in a proprietary binary format. But, for XML files such as action defaults files (.4ad), the localized strings must be written with a specific node, following the XML standards. To support localized strings in XML files, any file loaded into the Abstract User Interface tree is parsed to search for
<LStr> nodes. The <LStr> nodes define the same attributes as in the parent node with localized string identifiers, for example:

```xml
<Label text="Hello!" >
  <LStr text="label01" />
</Label>
```

The runtime system automatically replaces corresponding attributes in the parent node (text="Hello!"), with the localized text for the string identifier (label01) found in the compiled string files. After interpretation, the <LStr> nodes are removed from the XML data.

To take effect, a localized attribute in the <LStr> node must have a corresponding attribute in the parent node.

**Extracting strings from sources**

Localized strings can be easily extracted from .4gl and .per source files.

Use the fglcomp and fglform compilers with the -m option to extract localized strings.

```
$ fglcomp -m mymodule.4gl
```

The compilers dumps all localized string to stdout. This output can be redirected to a file to generate the default source string file with all the localized strings used in the source file. Source string files can then be re-organized, to centralize common messages in a unique .str file, and can then be compiled by fglmkstr into .4st files to be used by the runtime system.

**Related concepts**

[Creating source string files](#) on page 539

A *source string file* contains localized string definitions for a given language (or localization context).

**Organizing string resources**

Good practice in use of localized strings.

**What application strings to localize?**

When modifying sources to add % prefixes to the strings that have to be localized, you need to consider which string is the subject of internationalization, and leave other strings without a % prefix.

For example, strings used to build an SQL statement at runtime obviously must not be localized:

```sql
LET sql = "SELECT * FROM customers ", where_part
```

Typical strings to be localized are application messages:

```sql
MESSAGE %"The customer name is mandatory!"
```

Furthermore, you may also need to localize application data. In this case, you can store localized string identifiers in the database, and use the LSTR() function at runtime to get the localized string from your string resource files:

```sql
SELECT order_warning INTO rec.order_warning
  FROM orders WHERE ...
LET msg = LSTR(rec.order_warning)
DISPLAY BY NAME msg
```
Messages with parameters

Applications often display messages with variable parts. The message text is usually built at runtime with comma concatenation expression, where the message is split into different string literals:

```scheme
LET msg = "There are ", ord_count USING "<<<&", ", orders not yet validated for ", rec.cust_name, ",."
```

To simplify translation, consider reviewing the message construction by using the `SFMT()` operator, to set the variable parameters in your messages:

```scheme
LET msg = SFMT(%"orders.message.valid_count", ord_count, rec.cust_name)
```

You can then easily define the corresponding localized strings with `%n placeholders:

```scheme
-- English string file:
orders.message.valid_count = "%1 orders not yet validated for %2."  
-- French string file:
orders.message.valid_count = "%1 commandes ne sont pas encore validées pour %2." 
```

Note that in `SFMT()` calls, the `%n placeholders can be specified at different positions, depending on the language needs.

Development and runtime locale

The character set encoding (LANG/LC_ALL locale) used in sources and at runtime must match. For more details, see Application locale on page 512.

A good practice is to have sources in ASCII, and have string resources in the locale of your choice: The runtime locale can be a specific ISO8859-? encoding for each language, or UTF-8, to have a common encoding for all languages to be supported. However, you can also use UTF-8 in sources and at runtime, if you want to use original texts as string identifiers in your sources.

Note: The locale to be used at runtime will depend on the database locale used. You may need to support a set of string files using ISO8859-? and a set of files using UTF-8, if you need to deploy your application with ISO8859-? databases and UTF-8 databases.

String identifiers

A localized string must be identified with a unique name. By default, if you add a `% prefix before existing strings and you extract the strings with `fglcomp -m or `fglform -m, you will get string identifiers with the original text:

```
"OK" = "OK"
"Cancel" = "Cancel"
"Close" = "Close"
"There are %1 orders not yet validated for %2." = "There are %1 orders not yet validated for %2." 
```

At this point, you can keep the original text for string identifiers, or re-define more abstract identifiers (without quotes, such as `common.button.text.ok`).

Using the original text as string identifier has the advantage of been fast. It also simplifies translation because the original text is directly visible for the translator. However, the character set encoding should be UTF-8.

Note: If you want to leave the original text for string identifiers, you must make sure that the locale used at compile time matches the runtime locale. If the languages to be supported do not fit in a single encoding like ISO8859-15, you will have to convert your sources to UTF-8 and use UTF-8 at runtime.
Using abstract identifiers allows to maintain the sources in pure ASCII. Additionally, you can give an indication of the usage context by using a clear identifier. It is recommended that you also group common messages in a single string resource file. Using abstract identifiers will simplify uniqueness checking.

common.str:

```plaintext
common.button.text.accept = "OK"
common.button.text.cancel = "Cancel"
common.button.text.close = "Close"
common.topmenu.text.accept = "Validate"
...
```

orders001.str:

```plaintext
orders.messages.valid_count = "There are %1 orders not yet validated for %2." 
...
```

**Create directories for each language**

At runtime, the .42s string resource files to be loaded must be declared with the fglrun.localization.* FGLPROFILE entries.

In order to provide a set of string files for each language you want to support, organize the string files in directories dedicated to a given language:

```plaintext
/opt/app/resource/strings/fr_FR.iso8859-15  -- French strings in iso8859-15 code-set
/opt/app/resource/strings/jp_JP.utf8        -- Japanese strings in utf-8 code-set
```

**Related concepts**

- **Loading localized strings at runtime** on page 544
  Understand the rules for using localized strings at runtime.

- **Compiling string resource files (.str)**
  The .str source string files must be compiled to .42s binary files, in order to be loaded by the runtime system.

  To compile a source string file, use the fglmkstr compiler.

  ```plaintext
  $ fglmkstr filename.str
  ```

  The fglmkstr tool generates a .42s file with the filename prefix.

  **Important:** When compiling a .str source string file, you must set the locale (character set) corresponding to the encoding used in the .str file.

  **Related concepts**

  - **Extracting strings from sources** on page 542
    Localized strings can be easily extracted from .4gl and .per source files.

  - **Loading localized strings at runtime**
    Understand the rules for using localized strings at runtime.

  **Distributing compiled string files**

  The compiled string files (.42s) must be distributed with the program files in a directory specified in the FGLRESOURCEPATH environment variable.
Setting the correct locale
The locale (LANG/LC_ALL) corresponding to the encoding used in the .42s files must be set before starting the application. If the locale is wrong, the strings will not be loaded properly.

How does the runtime system load the strings?
The .42s compiled string resource files are loaded in the following order of precedence:
1. The files defined in FGLPROFILE,
2. A file having the same name as the current program (myprog.42m loads myprog.42s),
3. A file with the name "default.42s".

Each .42s string resource file is searched in several directories, as described in the FGLRESOURCEPATH reference topic.

String resource file sharing
Like .42m program pcode files, the .42s string resource files are shared by all fglrun processes running on the computer: The string file is loaded into memory with the mmap operating system function.

Defining a list of string files in FGLPROFILE
Specify a list of compiled string files with entries in the FGLPROFILE configuration file with the fglrun.localization entries.

First, define the total number of files with:

```plaintext
fglr.run.localization.file.count = integer
```

For each file, define the file name (with the .42s extension), including an index number (start index must be 1):

```plaintext
fglr.run.localization.file.index.name = "filename.42s"
```

Warning switches can be specified in FGLPROFILE.

If the text of a string is not found at runtime, the runtime system can show a warning, for development purposes.

```plaintext
fglr.run.localization.warnKeyNotFound = boolean
```

By default, this warning switch is disabled.

What happens if a 42s string file is not found?
If the .42s string file was defined with fglrun.localization.* FGLPROFILE entries, it is considered as mandatory, and the runtime system will raise error -8006 if the file is not found. If the programname.42s and default.42s string files are not found, no error is raised, because these are fallback string resource files.

What happens if a string is not defined in a resource file?
If a localized string is not defined in one of the compiled string files, the runtime system uses the string identifier as default text.

What happens if a string is defined more that once?
When a localized string is defined in several compiled string files, the runtime system uses the first string found.

For example, if the string "hello" is defined in program.42s as "hello from program", and in default.42s as "hello from default", the runtime system will use the text "hello from program".
Organizing .42s resource files in distribution directories

A set of .42s files using the same language and codeset is typically copied in a distribution directory with a name identifying the locale.

For example:

```
/opt/app/resource/strings/en_US.is08859-15 -- English strings in iso8859-15 codeset
/opt/app/resource/strings/fr_FR.is08859-15 -- French strings in iso8859-15 codeset
/opt/app/resource/strings/jp_JP.utf8        -- Japanese strings in utf-8 codeset
```

At runtime, specify the string file search path in the DBPATH/FGLRESOURCEPATH environment variable by adding the name of current locale as sub-directory. For example, to find the correct string files in one of the locale-specific directories shown above, set the FGLRESOURCEPATH variable as follows (UNIX™ shell):

```
$ echo $LC_ALL
jp_JP.utf8
$ FGLRESOURCEPATH="$FGLRESOURCEPATH:/opt/app/resource/strings/$LC_ALL"
$ export FGLRESOURCEPATH
$ echo $FGLRESOURCEPATH
/opt/app/forms:/opt/app/resource/strings/jp_JP.utf8
```

Localized string files on mobile devices

On mobile devices, the language is determined by the operating system regional settings.

- On iOS devices, go to Settings >> General >> Language & Region
- On Android™ devices, go to Settings >> General Management >> Language and Input

The selected language is identified by a locale code following the ISO 639 standard. Below are some language code examples; see the mobile OS documentation for information about available languages and their corresponding ISO 639-x codes.

- en - English (for all regions)
- en_US - English in the United States
- en_GB - English in the United Kingdom

On startup, the mobile app will search for localized string files (.42s) in the following directories:

3. `appdir/defaults`: the fallback directory where default string files are located.

In order to localize your application, you simply need to place your .42s localized string files in the appropriate language sub-directory.

Note: If the .42s file names do not match the main program name, define the list of localized strings files in the app's fglprofile file.

If you want to distinguish language categories (Simplified/Tradition Chinese), or if you want to use different texts for territories that share the same language (English in USA or Great Britain), create language sub-directories with the exact OS locale identifier:

- For English in the USA, use "en_US"
- For English in the United Kingdom, use "en_GB"
- For English in Canada, use "en_CA"
• etc...

appdir/en_US/mystings.42s
appdir/en_GB/mystings.42s
appdir/en_CA/mystings.42s

If the language category or region can be ignored, create language sub-directories with names matching the language identifier only:

• For English, use "en"
• For French, use "fr"
• For German, use "de"
• etc...

appdir/en/mystings.42s
appdir/fr/mystings.42s
appdir/de/mystings.42s

A default set of string files can be provided under appdir/defaults, in case the regional settings of the device do not match one of the locale directories you provide, otherwise the application will stop with error -8006:

appdir/defaults/mystings.42s

For more details about the mobile app directory structures (appdir), see Directory structure for GMA apps on page 3589 and Directory structure for GMI apps on page 3604.

Selecting the application language at runtime

The program can reset the path to find localized strings at runtime with the base.Application.reloadResources(path) method. This can be used to implement a login dialog, where the end user can choose the application language.

However, the base.Application.reloadResources(path) method will only have an impact on subsequent loaded resources: The localized strings of already loaded .42m modules and .42f forms are left unchanged. Subsequent loaded forms and modules will get localized strings from new resource lookup path. Therefore, this method must be called at the beginning of the program, before loading the first application form.

A typical pattern is to do display an initial form that allows the user to select the language, build the path to the localized strings files corresponding to the selected language, call the base.Application.reloadResources(path) method, and reload the initial form:

```import os
main

define done boolean
define rec record
    user string,
    pswd string,
    lang char(2)
end record,
path string

-- Login dialog with language selection
let rec.lang = "en" -- Can be "en", "fr", "ge" ...
while not done
    let path = os.path.join(base.application.getProgramDir(), rec.lang)
    call base.application.reloadResources(path)
    open form f from "main"
    display form f
    input by name rec.* without defaults
    on change lang
        exit input -- restarts the input with the new locale settings```
ON ACTION cancel
  EXIT PROGRAM
AFTER_INPUT
  LET done = TRUE
END_INPUT
END WHILE
  -- Here starts the real application code
... 
END MAIN

Related concepts
Deploying mobile apps on Android devices on page 3589
This section contains information to create a mobile application to be deployed on Android™ devices.

Deploying mobile apps on iOS devices on page 3604
This section contains information to create a mobile application to be deployed on iOS devices.

Locale and character set basics on page 516
This section is an introduction to locale and character set basics.

Predefined application strings
The runtime system may need to display text to the user.

For example, the runtime system library includes a report viewer, which displays a form. By default the text is in English, and you may need to localize the text in another language. So the strings of this component must be 'localizable', as in other application strings.

To customize the built-in strings, the runtime system uses the mechanism of localized strings.

All strings used by the runtime system are centralized in a unique file:

$FGLDIR/src/default.str

which is compiled into:

$FGLDIR/lib/default.42s

This file is always loaded by the runtime system.

To overwrite the defaults, you can redefine these strings in your own localized string files.

Best practices for localized strings
This section describes good practices to localize your application messages.

Program files and runtime language charset
Localization implies choosing a locale character set to use when executing a program.

The character set used at runtime must be the same as (or compatible with) the development character set: The compiled program files (.42m, .42f) are encoded in the character set used during compilation.

If the character set used in existing source files is different from the character set used at runtime for a different target language, consider using ASCII-7 only in your source code, and put application messages in .str localized string files, using the appropriate character set for each target language.

To support multiple languages at runtime, use the UTF-8 character set to encode your .str (and .42s) localized string files. When using UTF-8 in your source code and at runtime, the string keys can contain UTF-8 characters and you can use the original text as string key.

For more details about character set usage, see Application locale on page 512.
Defining the string key as identifier or as original text?

With localized strings, you have the choice of using the original text as string key, or of using a more programmatic-type string identifier.

String key as original text (English):

"The transaction has been validated." = "La transaction a été validée."

String key as identifier:

shipments.transaction.validated = "La transaction a été validée."

Pros and cons when using the original text as string key:

1. It is the fastest solution to localizing your application; you just need to add a percent sign (%) before texts in your source code, extract texts with fglcomp -m, and translate string files.
2. It simplifies the translation process, since the original text is always available as the string key.
3. If the charset used during development is different from the charset used at runtime, make sure to use ASCII-7 only characters in the string key, otherwise your .str file will be dependent on the locale character set used in development.
4. Long texts can be used as string keys. For common messages that are often used in your sources, consider defining string constants in a dedicated module as described below.

Pros and cons when using identifiers as string key:

1. You can clearly distinguish messages depending on the context. For example, the test "Ok" may be the same in the English button label and transaction status, but may require different texts in other languages.
2. Distinct identifiers are easier to manage in the translation process, for example to store ids and texts in a database.
3. String identifiers can be used directly with the %"ident" notation in sources, and do not required you to define constants for common strings.
4. String identifiers require you to replace the original text in the source code by the string identifier preceded by the % sign. When using the original text as key, you just need to add the % sign in the source.

Note: There is no constraint to exclusively using one of these patterns, you are free to use both methods for the same application.

Defining CONSTANT strings for common messages

Instead of repeating the same long string key in many places, group all common localized strings in modules and define constants:

```4gl
-- mystrings.4gl module
PUBLIC CONSTANT STR_CONF_DEL_REC = "%Are you sure you want to delete this record?"
PUBLIC CONSTANT STR_TX_COMMITTED = "%The transaction has been committed."
...
```

Note: When using the IMPORT FGL instruction, constant definitions are available with code completion.

Using parameterized strings

If a message contains a variable part, or must display a value that is only known at runtime, consider using the SFMT() operator to replace %n placeholders in your strings.

```4gl
orders.item.validated = "The item %1 has been validated."
```

Since %n placeholders are replaced by position, it is easy to put the placeholder at the position required by the language grammar.
English strings file:

```
stock.items.count = "Stock %1 contains now %2 aditional items."
```

French strings file:

```
stock.items.count = "%2 elements ajoutés dans le stock %1." 
```

When using the original text as string key, it is good practice to identify parameter placeholders with a different notation to the %n notation of SFMT(). For example, you can use P1, P2, etc:

"The item P1 has been validated." = "The item %1 has been validated."

Composed messages can be defined with %n and SFMT():

"P1 must be entered." = "%1 must be entered."
"Product code" = "Product code"
"Customer code" = "Customer code"

Then in the program code:

```
LET str = SFMT("P1 must be entered.", "Customer code")
```

To get the string "Customer Code must be entered."

**Example**

Here is an example using localized strings.

The source string file "common.str" (to be compiled with fglmkstr):

```
common.accept = "OK"
common.cancel = "Cancel"
common.yes = "Yes!"
common.no = "No!"
```

The source string file "customer.str" (to be compiled with fglmkstr):

```
customer.mainwindow.title = "Customers"
customer.listwindow.title = "Customer List"
customer.l_custnum = "Number:" 
customer.l_custname = "Name:" 
customer.c_custname = "The customer name"
customer.q_delete = "Do you want to delete this customer?"
```

The FGLPROFILE configuration file parameters:

```
fglrun.localization.file.count = 1
fglrun.localization.file.1.name = "common.42s"
```

Remark: The 'customer' string file does not have to listed in FGLPROFILE since it is loaded as it has the same name as the program.

The form specification file "customer.per":

```
ACTION DEFAULTS
    ACTION accept (TEXT="%common.accept")
    ACTION cancel (TEXT="%common.cancel")
END
LAYOUT (TEXT="%customer.mainwindow.title")
GRID
The program "customer.4gl" using the strings file:

```
MAIN
  DEFINE rec RECORD
    custnum INTEGER,
    custname CHAR(20)
  END RECORD
  OPEN FORM f1 FROM "customer"
  DISPLAY FORM f1
  INPUT BY NAME rec.*
  ON ACTION delete
    MENU "%customer.mainwindow.title"
      ATTRIBUTES(STYLE="dialog", COMMENT="%customer.q_delete")
      COMMAND "%common.yes"
      COMMAND "%common.no"
    END MENU
  END ACTION
END MAIN
```

Related concepts

Steps for application internationalization on page 538
Follow these steps to internationalize your application.

Creating source string files on page 539
A source string file contains localized string definitions for a given language (or localization context).

Compiling string resource files (.str) on page 544
The .str source string files must be compiled to .42s binary files, in order to be loaded by the runtime system.

Globals

Global variables can be shared among all modules of a program.

Understanding global blocks

Global symbols can be defined with the GLOBALS instruction

The GLOBALS instruction can be used to declare variables, constants and types for the whole program.

Important: Defining GLOBAL symbols (variables, constants, types) is deprecated and should be avoided. Instead of globals, use PUBLIC symbols in modules to be imported with IMPORT FGL.

Related concepts

Importing modules on page 480
Use the IMPORT ... instruction to import BDL, C or Java external modules in the current module.

**GLOBALS**

The Globals / END GBLs block and the GBLs instruction.

**Important:** Defining GLOBAL symbols (variables, constants, types) is deprecated and should be avoided. Instead of globals, use PUBLIC symbols in modules to be imported with IMPORT FGL.

**Syntax 1: Global block declaration**

```
GLOBALS
declaration-statement

END GLOBALS
```

1. `declaration-statement` is a variable, constant or type declaration.

**Syntax 2: Importing definitions from a globals file**

```
GLOBALS "filename"
```

1. `filename` is the file containing the definition of globals. `filename` can be a relative or an absolute path. To specify a path, the slash (/) directory separator can be used for UNIX™ and Windows® platforms.
2. Use this syntax to include global declarations in the current module.

**Usage**

To extend the scope of variables, constants or user types to the whole program, define a module containing a GBLs ... END GBLs block and include this global module with the GBLs "filename" statement in other modules.

**Note:** Initialization of global variables (DEFINE var INT = 100) is not supported.

The `filename` should contain the .4gl suffix. However, the compiler uses the file name as it is. Therefore it can accept other file extensions such as GBLs "stock.glb".

The `filename` can be a relative or an absolute path. To specify a path, the slash (/) directory separator can be used for UNIX™ and Windows® platforms.

When the `filename` specified by GBLs defines a relative file path, the globals file will be searched relatively to the location of the source provided to fglcomp.

**Note:** For compatibility, fglcomp searches globals files in pwd first, then the search is relative to the source file path. This can simplify build scripts: the compiler can be called from any directory.

**Related concepts**

- Variables on page 390
  - Explains how to define program variables.
- Constants on page 405
  - The definition of constants allows to centralize common static values.
- Types on page 431
  - Types can be defined by the programmer to centralize the definition of complex/structured variables.

**Rules for globals usage**

Follow the rules described in this topic in order to use globals properly.

**Important:** Defining GLOBAL symbols (variables, constants, types) is deprecated and should be avoided. Instead of globals, use PUBLIC symbols in modules to be imported with IMPORT FGL.
Rules for **GLOBALS** usage:

1. If you modify the globals file, you must recompile all the modules that include the file.
2. If a local element has the same name as another variable that you declare in the **GLOBALS** statement, only the local variable is visible within its scope of reference.
3. You can declare several **GLOBALS** .. **ENDGLOBALS** blocks in the same module.
4. A source file with **GLOBALS** .. **ENDGLOBALS** block must not contain any executable statement.
5. Do not write a declaration statement outside a **GLOBALS** .. **ENDGLOBALS** block in the globals file.
6. You do not need to compile the source file containing the **GLOBALS** block. However, it is recommended to compile the globals file to detect errors.
7. You can declare several **GLOBALS** "filename" instructions in the same globals using module.
8. Although you can include multiple **GLOBALS** .. **ENDGLOBALS** statements in the same application, do not declare the same identifier within more than one **GLOBALS** declaration. Even if several declarations of a global elements defined in multiple places are identical, declaring any global element more than once can result in compilation errors or unpredictable runtime behavior.
9. A **GLOBALS** .. **ENDGLOBALS** block can hold **GLOBALS** "filename" instructions. In such case, the specified files will be included recursively.

**Database schema in globals**

Globals files can define the database schema to be used by the compiler to resolve **DEFINE** ... **LIKE** statements.

The schema specification must appear before the **GLOBALS** keyword starting the globals block.

The schema specification is propagated to the modules including the globals file defining the database schema. These modules can use **DEFINE** ... **LIKE** without an explicit **SCHEMA** instruction.

Furthermore, when using the **DATABASE** instruction instead of **SCHEMA**, if the module including the globals contains the **MAIN** block, the **DATABASE** specification of the globals file will be propagated and result in an implicit database connection at runtime.

**Example**

```
SCHEMA stores
GLOBALS
    DEFINE cust_rec LIKE customer.*
    ...
END GLOBALS
```

**Related concepts**

Database schema on page 467

Defines database table structures with column type information to be reused in program variable definitions.

**Content of a globals file**

A globals file contains a **GLOBALS** .. **ENDGLOBALS** block.

As a **GLOBALS** block can also be defined in regular modules, it is possible to include a source containing more than a **GLOBALS** block. When including such module, the sections before and after the **GLOBALS** block are ignored by the compiler. The source defining the global elements can be compiled individually.

For example, it is allowed to define a module A with a **GLOBALS** .. **ENDGLOBALS** block, followed by function definitions. This module can be compiled and functions will be taken into account. Module A can then be included in module B with a **GLOBALS** "filename" instruction, and when compiling module B the function definitions of the included module A will be ignored. **IMPORT** instructions before the a **GLOBALS** .. **ENDGLOBALS** block will also be ignored in such case.
Examples
Globals usage examples.

**Example 1: Multiple GLOBALS file**
Module "labels.4gl": This module defines the text that is displayed on the screen

```plaintext
GLOBALS
    CONSTANT g_lbl_val = "Index:"
    CONSTANT g_lbl_idx = "Value:"
END GLOBALS
```

Module "globals.4gl": Declares a global array and a constant containing its size

```plaintext
GLOBALS "labels.4gl" -- this statement could be line 2 of main.4gl

GLOBALS
    DEFINE g_idx ARRAY[100] OF CHAR(10)
    CONSTANT g_idxsize = 100
END GLOBALS
```

Module "database.4gl": This module is dedicated to database access

```plaintext
GLOBALS "globals.4gl"

FUNCTION get_id()
    DEFINE li INTEGER
    FOR li = 1 TO g_idxsize -- this could be a FOREACH statement
        LET g_idx[li] = g_idxsize - li
    END FOR
END FUNCTION
```

Module "main.4gl": Fill in the global array and display the result

```plaintext
GLOBALS "globals.4gl"

MAIN
    DISPLAY "Initializing constant values for this application..."
    DISPLAY "Filling the data from function get_idx in module database.4gl..."
    CALL get_id()
    DISPLAY "Retrieving a few values from g_idx"
    CALL display_data()
END MAIN

FUNCTION display_data()
    DEFINE li INTEGER
    LET li = 1
    WHILE li <= 10 AND li <= g_idxsize
        DISPLAY g_lbl_idx CLIPPED || li || " " || g_lbl_val CLIPPED ||
        g_idx[li]
        LET li = li + 1
    END WHILE
END FUNCTION
```
Predefined constants

The language defines a set of global constants that can be used in the programs.

NULL

The NULL constant is provided as the "nil" value.

Syntax

```plaintext
NULL
```

Usage

When comparing variables to NULL, use the IS NULL operator, not the equal operator.

If an element of an expression is null, the expression is evaluated to NULL.

Variables are initialized to NULL or to zero, according to their data type.

Empty character string literals ("") are equivalent to NULL.

Important: NULL cannot be used with the == equal comparison operation, you must use IS NULL. For more details, see the IF statement.

Example

```plaintext
MAIN
    DEFINE s CHAR(5)
    LET s = NULL
    DISPLAY "s IS NULL evaluates to:"
    IF s IS NULL THEN
        DISPLAY "TRUE"
    ELSE
        DISPLAY "FALSE"
    END IF
END MAIN
```

Related concepts

Variable default values on page 395
Variables get a default value when defined.

Primitive Data types on page 289
Selecting the correct data type assists you in the input, storage, and display of your data.

TRUE

TRUE is a predefined constant to be used in boolean expressions.

Syntax

```plaintext
TRUE
```

Usage

TRUE is a predefined constant that can be used as a boolean value in boolean expressions.

The TRUE constant is equal to 1 (one).

TRUE and FALSE are typically used as return values of functions that give a binary result.
Example

```
MAIN
  DEFINE short BOOLEAN
  LET short = is_short("abcdef")
  IF short THEN
    DISPLAY "String is short."
  END IF
END MAIN

FUNCTION is_short(s)
  DEFINE s STRING
  IF s.getLength() < 10 THEN
    RETURN TRUE
  ELSE
    RETURN FALSE
  END IF
END FUNCTION
```

Related concepts

BOOLEAN on page 293
The BOOLEAN data type stores a logical value, TRUE or FALSE.

FALSE

FALSE is a predefined constant to be used in boolean expressions.

Syntax

```
FALSE
```

Usage

FALSE is a predefined constant that can be used as a boolean value in boolean expressions.

The FALSE constant is equal to 0 (zero).

TRUE and FALSE are typically used as return values of functions that give a binary result.

Example

```
MAIN
  DEFINE odd BOOLEAN
  LET odd = is_odd(125763)
  IF odd THEN
    DISPLAY "Number is odd."
  END IF
END MAIN

FUNCTION is_odd(value)
  DEFINE value INTEGER
  IF value MOD 2 = 1 THEN
    RETURN TRUE
  ELSE
    RETURN FALSE
  END IF
END FUNCTION
```

Related concepts

BOOLEAN on page 293
The **BOOLEAN** data type stores a logical value, TRUE or FALSE.

**NOTFOUND**

**NOTFOUND** is a predefined constant used to check if an SQL statement returns rows.

**Syntax**

```plaintext
NOTFOUND
```

**Usage**

The **NOTFOUND** constant is used to test the execution status of an SQL statement returning a result, to check whether rows have been found.

The **NOTFOUND** constant is equal to 100.

You typically compare `SQLCA.SQLCODE` to **NOTFOUND**, after a `SELECT` statement execution.

**Example**

```plaintext
MAIN
  DATABASE stores
  SELECT tabid FROM systables WHERE tabid = 1
  IF SQLCA.SQLCODE = NOTFOUND THEN
    DISPLAY "No row was found"
  END IF
END MAIN
```

**Related concepts**

- **SQL execution diagnostics** on page 591
  If an SQL statement execution fails, error description can be found in the `SQLCA.SQLCODE`, `SQLSTATE`, `STATUS` and `SQLERRMESSAGE` predefined registers.
- **SELECT** on page 700
  Produces a result set from a query on database tables.

**Configuration options**

Compiler and runtime system can be controlled with several configuration settings.

**OPTIONS (Compilation)**

**OPTIONS** outside program blocks defines semantics of the language for the compiler.

**Syntax**

```plaintext
OPTIONS
  [ SHORT CIRCUIT ]
  [ , ... ]
```

**Usage**

The **OPTIONS** statement used before any `MAIN`, `FUNCTION` or `REPORT` program block defines language semantics options, that will take effect for the current module only. Unlike runtime options, compiler options cannot be changed during program execution.

The statement to define compiler options must be placed before the `MAIN` block in the main module, or before the first `FUNCTION / REPORT` block in other modules.
The OPTIONS compiler directive allows for the control of features **Controlling semantics of AND / OR operators** on page 558

**Example**

```fortran
OPTIONS SHORT CIRCUIT
MAIN
    DISPLAY "Global Options example"
END MAIN
```

**Related concepts**

*OPTIONS (Runtime)* on page 558
The OPTIONS instruction inside program blocks controls program behavior at runtime.

**Controlling semantics of AND / OR operators**

The OPTIONS SHORT CIRCUIT defines the semantics of AND/OR operators.

When using OPTIONS SHORT CIRCUIT at the beginning of a module, the runtime system will optimize the evaluation of boolean expressions involving AND and OR operators, by using the short-circuit evaluation method (also called minimal evaluation method). This behavior is enabled for the whole module.

By default, the behavior of AND and OR operators is to evaluate all operands on the left and right side of the operator. In fact this is not required. If the left operand of the AND evaluates to FALSE, there is no need to evaluate the right operand, because the result of the AND operator will be false, anyway. Similarly, when the left operand of an OR expression evaluates to TRUE, there is no need to evaluate the right operand, since the result of the boolean expression will be true, anyway.

This method can improve performances and simplify programming. However, existing code may rely on the fact that all parts of a boolean expression are evaluated, especially when calling functions that do some processing. By using the short-circuit evaluation method, it is unsure that the function used in the right operand of an AND/OR will be called, because it depends on the result of the left operand.

By using short-circuit evaluation, it is possible to reference a dynamic array in the same boolean expression, after checking that the index is in the current array element range:

```fortran
IF x<=arr.getLength() AND arr[x].order_date > TODAY THEN
  ...
END IF
```

With the default AND semantics, in this code, the right operand is also evaluated. If the x index is greater than the array length, new array elements will be automatically created in the expression on the right of the AND operator. To avoid this situation, you are forced to write the following code, when OPTIONS SHORT CIRCUIT is not used:

```fortran
IF x<=arr.getLength() THEN
  IF arr[x].order_date > TODAY THEN
    ...
  END IF
END IF
```

**Related concepts**

*Boolean expressions* on page 331
This section covers boolean expression evaluation rules.

**OPTIONS (Runtime)**

The OPTIONS instruction inside program blocks controls program behavior at runtime.

**Syntax**

```fortran
OPTIONS options-clause [, ...]
```
Usage

Use the `OPTIONS` instruction inside a function block to control the behavior of the runtime system for the rest of the program execution.

A program can execute successive `OPTIONS` statements at different places in the code.

The runtime `OPTIONS` statement allows for control of the following runtime features:

- **Defining the position of reserved lines** on page 559
- **Defining default TTY attributes** on page 560
- **Defining field tabbing order method** on page 561
- **Defining the field input loop** on page 561
- **Application termination** on page 563
- **Front-end termination** on page 563
- **Defining the message file** on page 563
- **Defining control keys** on page 564
- **Setting default screen modes for sub-programs** on page 565
- **Enabling/disabling SQL interruption** on page 565

**Defining the position of reserved lines**

The `OPTIONS` element `LINE` defines position of dedicated screen lines.

**Syntax**

```plaintext
OPTIONS
  | MENU LINE line-value
  | MESSAGE LINE line-value
  | COMMENT LINE {OFF | line-value}
  | PROMPT LINE line-value
  | ERROR LINE line-value
  | FORM LINE line-value
```

**Usage**

The `OPTIONS` statement can define the positions of reserved lines for menus, forms and messages.

Reserved window lines are used in **TUI mode**. These options are not required in **GUI mode**. In GUI mode, these options have no effect, except when using the **traditional mode**, where program windows are rendered as in a dumb terminal.

- **COMMENT LINE** specifies the position of the comments for fields. Field comments are defined with the `COMMENT` attribute in the form specification file. The default is `(LAST-1)` for the `SCREEN`, and `LAST` for all other windows. The field comment display can be disabled with `COMMENT LINE OFF`.
- **ERROR LINE** specifies the position on the screen for the text of the `ERROR` statement. The text of the `ERROR` statement is always displayed independently to the current window. The default is the `LAST` line of the screen.
- **MESSAGE LINE** specifies the position of the message line in the current window. This reserved line displays the text of the `MESSAGE` statement. The default is `FIRST+1` (line 2 in the current window). Note that the default message line position is the same as the `MENU` option comment line.
- **FORM LINE** specifies the window line where `forms` are displayed. The default is `FIRST+2` (line 3 in the current window).
- **MENU LINE** specifies the position of the menu line in the current window. This line displays the menu name and options, as defined by the `MENU` statement. The default is the `FIRST` line in the current window.
- **PROMPT LINE** specifies the position of the prompt line where the text of `PROMPT` statements is displayed. The default value is the `FIRST` line in the current window.

You can specify any of the following positions for each reserved line:
Table 158: Reserved line expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIRST</td>
<td>The first line of the screen or window.</td>
</tr>
<tr>
<td>FIRST + integer</td>
<td>A relative line position from the first line.</td>
</tr>
<tr>
<td>integer</td>
<td>An absolute line position in the screen or window.</td>
</tr>
<tr>
<td>LAST - integer</td>
<td>A relative line position from the last line.</td>
</tr>
<tr>
<td>LAST</td>
<td>The last line of the screen or window.</td>
</tr>
</tbody>
</table>

Related concepts

Genero user interface modes on page 1109
User interface modes allow you to adapt the application form rendering to different types of displays.

Defining default TTY attributes

The `OPTIONS {INPUT | DISPLAY} ATTRIBUTES` defines default TTY attributes for dialogs and display statements.

Syntax

```
OPTIONS { |
  INPUT ATTRIBUTES ( \ FORM \ WINDOW \ attributes ) |
  DISPLAY ATTRIBUTES ( \ FORM \ WINDOW \ attributes ) |
}
```

Usage

`OPTIONS INPUT ATTRIBUTES` defines the default color and terminal effect attributes that will be used in subsequent dialog statement.

`OPTIONS DISPLAY ATTRIBUTES` defines the default attributes for display statements.

The display attributes are based on dumb terminal (i.e. TTY) possibilities, but will be rendered accordingly on GUI mode. Graphical front-ends can be configured to render TTY attributes in a specific way. Instead of TTY based attributes, consider using presentation styles in new developments.

Any display attribute defined by the `OPTIONS` statement remains in effect until the runtime system encounters a statement that redefines the same attribute. This can be another `OPTIONS` statement, or an `ATTRIBUTE` clause in one of the following statements:

- `CONSTRUCT`
- `INPUT`
- `DISPLAY`
- `DIALOG`
- `INPUT ARRAY`
- `DISPLAY ARRAY`
- `OPEN WINDOW`

The `ATTRIBUTE` clause in these statements only redefines the attributes temporarily. After the window closes or after the dialog statement terminates, the runtime system restores the attributes from the most recent `OPTIONS` statement.

The `FORM` keyword in `INPUT ATTRIBUTE` or `DISPLAY ATTRIBUTE` clauses instructs the runtime system to use the input or display attributes of the current form. Similarly, you can use the `WINDOW` keyword of the same clauses to instruct the program to use the input or display attributes of the current window. You cannot combine the `FORM` or `WINDOW` attributes with any other attributes.
This table shows the valid input and display attributes:

**Table 159: Input and display attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLACK, BLUE, CYAN, GREEN,</td>
<td>The TTY color of the displayed text.</td>
</tr>
<tr>
<td>MAGENTA, RED, WHITE,</td>
<td></td>
</tr>
<tr>
<td>YELLOW</td>
<td></td>
</tr>
<tr>
<td>BOLD, DIM, INVISIBLE,</td>
<td>The TTY font attribute of the displayed text.</td>
</tr>
<tr>
<td>NORMAL</td>
<td></td>
</tr>
<tr>
<td>REVERSE, BLINK, UNDERLINE</td>
<td>The TTY video attribute of the displayed text.</td>
</tr>
</tbody>
</table>

**Defining the field input loop**

The OPTIONS INPUT [NO] WRAP instructions defines field wrapping in dialogs.

**Syntax**

```plaintext
OPTIONS INPUT [NO] WRAP
```

**Usage**

By default, an interactive statement such as `CONSTRUCT` or `INPUT` terminates when the focus leaves the last field controlled by the dialog instruction.

The OPTIONS INPUT WRAP instruction can change this behavior, causing the cursor to move from the last field to the first, repeating the sequence of fields until the dialog is validated or canceled.

The INPUT NO WRAP option restores the default input loop behavior.

**Example**

```plaintext
MAIN
  OPTIONS INPUT WRAP
  ...
END MAIN
```

**Related concepts**

[Defining field tabbing order method](#) on page 561

**Defining field tabbing order method**

**Syntax**

```plaintext
OPTIONS FIELD ORDER ↓ CONSTRAINED ↓ UNCONSTRAINED ↓ FORM ↓
```

**Usage**

Tabbing order is used in interactive instructions such as `INPUT`, `INPUT ARRAY`, or `CONSTRUCT`, where individual fields can get the focus.

The FIELD ORDER runtime option defines the default behavior when moving from field to field with the TAB and SHIFT-TAB keys in GUI mode, and with the Up / Down arrow keys in TUI mode.

**Note:** The OPTIONS FIELD ORDER defines the global field order mode. The field order mode can also be defined at the dialog level, with the FIELD ORDER dialog attribute.
By default, the tabbing order is defined by the list of fields used by the program instruction. This corresponds to the FIELD ORDER CONSTRAINED option, which is the default.

When using the FIELD ORDER UNCONSTRAINED in TUI mode, the Up and Down arrow keys will move the cursor to the field above or below the current field, respectively. When using the default FIELD ORDER CONSTRAINED option, the Up and Down arrow keys move the cursor to the previous or next field, respectively. If FIELD ORDER UNCONSTRAINED is used, the Dialog.fieldOrder FGLPROFILE entry is ignored.

The FIELD ORDER UNCONSTRAINED option can only be supported in TUI mode, with a simple form layout. It is not recommended to use this option in GUI mode.

The FIELD ORDER FORM option instructs interactive instructions to use the tabbing order defined by the TABINDEX attributes of the current form fields. With this option, tabbing order can be defined in the layout of the form, independently from the program instruction. This is the preferred way in GUI mode. When FIELD ORDER FORM is used, the Dialog.fieldOrder FGLPROFILE entry is ignored.

Example

Form "form1.per":

```plaintext
LAYOUT
GRID
{
  First name: [f001] Last name: [f002]
  Address: [f003]
}
END
END

ATTRIBUTES
EDIT f001 = FORMONLY.fname, TABINDEX = 2;
EDIT f002 = FORMONLY.lname, TABINDEX = 1;
EDIT f003 = FORMONLY.address, TABINDEX = 0;
END

Module "main.4gl":

```plaintext
MAIN
   DEFINE rec RECORD
       fname VARCHAR(20),
       lname VARCHAR(20),
       address VARCHAR(50)
   END RECORD
   OPTIONS INPUT WRAP
   OPEN FORM f1 FROM "form1"
   DISPLAY FORM f1
   OPTIONS FIELD ORDER CONSTRAINED
   INPUT BY NAME rec.*
   OPTIONS FIELD ORDER UNCONSTRAINED
   INPUT BY NAME rec.*
   OPTIONS FIELD ORDER FORM
   INPUT BY NAME rec.*
END MAIN
```

Related concepts

Defining the tabbing order on page 1729
Control the order of tabbing through the fields with the TABINDEX attribute.

**Application termination**
The OPTIONS TERMINATE SIGNAL defines a callback function in case of SIGTERM signal.

**Syntax**

```plaintext
OPTIONS ON TERMINATE SIGNAL CALL function
```

**Usage**
The OPTIONS ON TERMINATE SIGNAL CALL function defines the function that must be called when the application receives the SIGTERM signal.

With this option, you can control program termination. If this statement is not called, the program is stopped with an exit value of SIGTERM (15).

**Important:** This feature is only supported on UNIX-like operating systems.

Use the OPTIONS ON TERMINATE SIGNAL CALL function instruction with care, and do not execute complex code in the callback function. The code is expected to contain only simple and short cleanup operations; any interactive instruction must be avoided.

**Front-end termination**
The OPTIONS CLOSE APPLICATION instruction defines the callback function in case of front-end termination.

**Syntax**

```plaintext
OPTIONS ON CLOSE APPLICATION CALL function
```

**Usage**
The OPTIONS ON CLOSE APPLICATION CALL function can be used to execute specific code when the front-end stops. For example, when the front-end program is stopped, when the user workstation session is ended, or when the workstation is shut down.

Before stopping, the front-end sends an internal event that is trapped by the runtime system. When a callback function is specified with this program option command, the application code that was executing is canceled, and the callback function is executed before the program stops.

Use the OPTIONS ON CLOSE APPLICATION CALL function instruction with care, and do not execute complex code in the callback function. The code is expected to contain only simple and short cleanup operations; any interactive instruction must be avoided.

**Important:** A front-end program crash or network failure is not detected and cannot be handled by the ON CLOSE APPLICATION option.

**Related concepts**

GUI front-end connection on page 1116
This section explains runtime to front-end connection in its simplest form.

**Defining the message file**
The OPTIONS HELP FILE instruction defines the name of the message file.

**Syntax**

```plaintext
OPTIONS HELP FILE filename
```
Usage

The OPTIONS HELP FILE instruction specifies an expression that returns the file name of a help file. This file name can also include a path name.

Messages in this file can be referenced by number in form-related statements, and are displayed at runtime when the user presses the Help key.

Message files are found in the directories as described in the FGLRESOURCEPATH reference topic.

Defining control keys

The OPTIONS logical-key KEY physical-key instruction defines physical keys for logical keys (TUI mode).

Syntax

```
OPTIONS logical-key KEY physical-key
```

where logical-key is one of:

```
\[ ACCEPT | HELP | INSERT | DELETE | NEXT | PREVIOUS \]
```

1. logical-key identifies a logical key corresponding to a dialog action or operation.
2. physical-key defines the keyboard accelerator (F1, CTRL-Z), to be associated with the logical-key.

Usage

The OPTIONS logical-key KEY physical-key instruction can be used to associate physical keys (F2) for logical keys corresponding to a dialog operation (screen record page up/down) or dialog action (such as insert), to be triggered in the current interactive instruction.

This form of the OPTIONS instruction is provided for backward compatibility with the TUI mode: In GUI mode, use the action configuration to define accelerator keys for actions.

Table 160: Description of OPTIONS logical keys

<table>
<thead>
<tr>
<th>OPTIONS clause</th>
<th>Corresponding action / dialog function</th>
<th>Default Physical Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPTIONS ACCEPT KEY</td>
<td>The accept action, that validates the current dialog.</td>
<td>ESCAPE</td>
</tr>
<tr>
<td>OPTIONS HELP KEY</td>
<td>The help action, to display help messages.</td>
<td>CONTROL-W</td>
</tr>
<tr>
<td>OPTIONS INSERT KEY</td>
<td>The insert action, to insert a new row in an INPUT ARRAY.</td>
<td>F1</td>
</tr>
<tr>
<td>OPTIONS DELETE KEY</td>
<td>The delete action, to deletes the current row in an INPUT ARRAY.</td>
<td>F2</td>
</tr>
<tr>
<td>OPTIONS NEXT KEY</td>
<td>Scrolls to the next screen record page in an INPUT ARRAY or DISPLAY ARRAY.</td>
<td>F3</td>
</tr>
<tr>
<td>OPTIONS PREVIOUS KEY</td>
<td>Scrolls to the previous screen record page in an INPUT ARRAY or DISPLAY ARRAY.</td>
<td>F4</td>
</tr>
</tbody>
</table>

Note: In TUI mode, the key to cancel a dialog is the interruption key CTRL-C. In this context, the key to cancel dialogs can only be configured with the TTY settings, to raise an interruption signal (see stty command).

You can specify the following keywords for the physical key names:
Table 161: Keywords for physical key names

<table>
<thead>
<tr>
<th>Key Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESC or ESCAPE</td>
<td>The ESC key (not recommended, use ACCEPT instead).</td>
</tr>
<tr>
<td>INTERRUPT</td>
<td>The interruption key (on UNIX™, interruption signal).</td>
</tr>
<tr>
<td>TAB</td>
<td>The TAB key (not recommended).</td>
</tr>
<tr>
<td>CONTROL-char</td>
<td>A control key where char can be any character except A, D, H, I, J, K, L, M, R, or X</td>
</tr>
<tr>
<td>F1 through F255</td>
<td>A function key.</td>
</tr>
<tr>
<td>LEFT</td>
<td>The left arrow key.</td>
</tr>
<tr>
<td>RETURN or ENTER</td>
<td>The return key.</td>
</tr>
<tr>
<td>RIGHT</td>
<td>The right arrow key.</td>
</tr>
<tr>
<td>DOWN</td>
<td>The down arrow key.</td>
</tr>
<tr>
<td>UP</td>
<td>The up arrow key.</td>
</tr>
<tr>
<td>PREVIOUS or PREVPAGE</td>
<td>The previous page key.</td>
</tr>
<tr>
<td>NEXT or NEXTPAGE</td>
<td>The next page key.</td>
</tr>
</tbody>
</table>

You might not be able to use other keys that have special meaning to your version of the operating system. For example, CONTROL-C, CONTROL-Q, and CONTROL-S specify the Interrupt, XON, and XOFF signals on many UNIX™ systems.

**Related concepts**

Configuring actions on page 1744
Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with action attributes.

**Setting default screen modes for sub-programs**
The OPTIONS RUN IN instruction defines the TTY mode to run sub-programs.

**Syntax**

```plaintext
OPTIONS RUN IN \FORM\LINE\ MODE
```

**Usage**

This instruction defines the default mode to run child applications with the RUN command.

Details about the line mode and form mode are available in the reference topic of the RUN on page 490 instruction.

**Enabling/disabling SQL interruption**
The OPTIONS SQL INTERRUPT instruction enables or disables SQL statement interruption.

**Syntax**

```plaintext
OPTIONS SQL INTERRUPT \ ON \ OFF
```

**Usage**

The OPTIONS SQL INTERRUPT instruction controls interruption event detection during the execution of long running SQL statements.

By default, SQL interruption is off.
Pay attention to the fact that not all database servers support SQL interruption.

Depending on the type of database server and client, SQL interruption handling may require additional overhead. Consider enabling SQL interruption only for SQL queries that can take a while to execute or can be blocked for a long time because of concurrent access (locks).

If an SQL statement is interrupted, SQLCA.SQLCODE is set to -213 and the runtime system raises an error. Use a TRY/CATCH block or WHENEVER ERROR CONTINUE to trap this SQL error.

**Example**

```
MAIN
 DEFINE cnt INTEGER
 DEFER INTERRUPT -- Do not stop if interrupt signal is caught
 CONNECT TO "mydb"
 WHENEVER ERROR CONTINUE -- Continue in case of SQL interrupt error
 OPTIONS SQL INTERRUPT ON -- Enable SQL interruption
 SELECT COUNT(*) INTO cnt FROM stock -- Long running query
 OPTIONS SQL INTERRUPT OFF -- Disable SQL interruption
 WHENEVER ERROR STOP -- Reset default exception handler
 IF SQLCA.SQLCODE == -213 THEN
   DISPLAY "SQL Statement interrupted by user"
 END IF
END MAIN
```

**Related concepts**

* Using SQL interruption on page 601
  Interrupt long running SQL queries, or interrupt queries waiting for locked data.

**Runtime configuration in FGLPROFILE**

The behavior of the runtime system can be controlled with FGLPROFILE configuration parameters.

**Responding to CTRL_LOGOFF_EVENT**

FGLPROFILE fglrun.ignoreLogoffEvent controls program behavior in case of logoff events on Windows™ platforms.

**Syntax**

```
fglrn.ignoreLogoffEvent = true
```

**Usage**

On Windows™ platforms, when the user disconnects, the system sends a CTRL_LOGOFF_EVENT event to all console applications. When the runtime system receives this event, it stops immediately.

On a Windows™ Terminal Server, if an Administrator user closes his session, a CTRL_LOGOFF_EVENT is sent to all console applications started by ANY user connected to the machine (even if these applications were not started by the administrator).

To prevent the runtime system from stopping on a logoff event, you can use the fglrun.ignoreLogoffEvent entry in the FGLPROFILE configuration file. If this entry is set to true, the CTRL_LOGOFF_EVENT event is ignored by the runtime system.

As a result, when the administrator user disconnects on a Windows™ Terminal Server, programs started by remote users would not stop.

**Related concepts**

* The FGLPROFILE file(s) on page 255
FGLPROFILE environment variable defines Genero BDL configuration files

**DEFER INTERRUPT / QUIT**

The DEFER instruction defines the program behavior when interruption or quit signals are received.

**Syntax**

```
DEFER INTERRUPT | QUIT
```

**Usage**

The DEFER instruction controls the behavior of the program when an interruption or quit signal has been received.

DEFER INTERRUPT and DEFER QUIT instructions should only be used in the MAIN block, to be executed at the beginning of the program.

DEFER INTERRUPT indicates that the program must continue when it receives an interruption signal. By default, the program stops when it receives an interrupt signal.

Once deferred, you cannot reset to the default behavior.

When an interrupt signal is caught by the runtime system and DEFER INTERRUPT is used, the INT_FLAG global variable is set to true by the runtime system.

Interrupt signals are raised on terminal consoles when the user presses a key like CTRL-C, depending on the stty configuration. When a program is displayed through a front-end, no terminal console is used; therefore, users cannot send interrupt signals with the CTRL-C key. To send an interruption request from the front-end, you must define an 'interrupt' action view.

DEFER QUIT indicates that the program must continue when it receives a quit signal. By default, the program stops when it receives a quit signal.

When a quit signal is caught by the runtime system and DEFER QUIT is used, the QUIT_FLAG global variable is set to true by the runtime system.

**Related concepts**

User interruption handling on page 1711
Allow the end user to cancel the execution of a procedure in the program.

**Program registers**

Predefined global registers can be used in programs to detect errors, signals and events.

**STATUS**

STATUS is a predefined variable that contains the execution status of the last instruction.

**Syntax**

```
STATUS
```

**Usage**

STATUS is a predefined variable that contains the execution status of the last program instruction.

STATUS allows diagnostic information about procedural, interactive, and SQL instructions to be obtained.

The data type of STATUS is INTEGER.
Note: While STATUS can be modified by hand, it is not recommended except in specific situations as shown in the STATUS example.

STATUS is typically used with WHENEVER ERROR CONTINUE or WHENEVER ERROR CALL, or TRY/CATCH blocks, to identify the type of error that occurred.

STATUS will be set for expression evaluation errors only when WHENEVER ANY ERROR is used.

After an SQL statement execution, STATUS contains the value of SQLCA.SQLCODE.

STATUS is set to an error code when an instruction produces an error, or it is reset to zero when non-assignment instructions succeed. A typical mistake is to test STATUS after a DISPLAY STATUS instruction, written after an SQL statement:

```
WHENEVER ERROR CONTINUE
DELETE FROM _invalid_table_name_ where col = 1
WHENEVER ERROR STOP
DISPLAY "STATUS: ", STATUS -- this DISPLAY instruction reset STATUS to zero
IF STATUS<0 THEN
   DISPLAY "SQL Error!"
   EXIT PROGRAM 1
END IF
```

Tip: Use SQLCA.SQLCODE for SQL error detection, and use STATUS for other language instructions.

Example

```
MAIN
   DISPLAY is_number(NULL)
   DISPLAY is_number("abc")
   DISPLAY is_number("-12.45")
END MAIN

FUNCTION is_number(s)
   DEFINE s STRING
   DEFINE f FLOAT, l_status INTEGER
   IF length(s)==0 THEN
      RETURN FALSE
   END IF
   WHENEVER ANY ERROR CONTINUE
   LET STATUS=0 # Needed, as STATUS won't be set if succeeds
   LET f = s
   LET l_status = STATUS
   WHENEVER ANY ERROR CONTINUE
   IF l_status == 0 THEN
      RETURN TRUE
   ELSE
      RETURN FALSE
   END IF
END FUNCTION
```

Related concepts
Example 2: SQL error handling with WHENEVER on page 510
WHENEVER instruction on page 503
Use the WHENEVER instruction to define how exceptions must be handled for the rest of the module.

**INT_FLAG**

INT_FLAG is a predefined variable set to TRUE when an interruption event is detected.

**Syntax**

<table>
<thead>
<tr>
<th>INT_FLAG</th>
</tr>
</thead>
</table>

**Usage**

INT_FLAG is set to TRUE by the runtime system when an interruption event is detected by the runtime system. The interruption event is raised when the user presses the interruption key, or when the graphical front-end sends an interruption event while the program is running in a procedure or SQL query.

INT_FLAG must be used with the DEFER INTERRUPT configuration instruction. If the DEFER INTERRUPT instruction is not specified, and interruption signal will stop the program execution.

When an interruption event occurs during a procedural instruction (FOR loop), the runtime system sets INT_FLAG to TRUE. It is up to the program to check the INT_FLAG variable.

When an interruption event occurs during an interactive instruction (INPUT, CONSTRUCT), the runtime system sets INT_FLAG to TRUE and exits from the interactive instruction. It is recommended that you test INT_FLAG after an interactive instruction to check whether the input has been canceled.

Once INT_FLAG is set to TRUE, it must be reset to FALSE in order to detect a new interruption event.

INT_FLAG will also be used by the runtime system as diagnostic flag for predefined action block execution such as ON INSERT in DISPLAY ARRAY.

**Example**

```plaintext
MAIN
   DEFER INTERRUPT
   LET INT_FLAG = FALSE
   INPUT BY NAME ... 
   AFTER INPUT
      IF INT_FLAG THEN
         MESSAGE "The input is canceled."
      END IF 
   ... 
END INPUT 
... 
END MAIN
```

**Related concepts**

QUIT_FLAG on page 569
QUIT_FLAG is a predefined variable set to TRUE when a quit event is detected.

**QUIT_FLAG**

QUIT_FLAG is a predefined variable set to TRUE when a quit event is detected.

**Syntax**

<table>
<thead>
<tr>
<th>QUIT_FLAG</th>
</tr>
</thead>
</table>
**Usage**

QUIT_FLAG is set to TRUE when a quit event is detected by the runtime system. The quit event is raised when the user presses the quit signal key (\[Ctrl\]+[Backslash]), or when another process sends the quit signal to the runtime system process.

QUIT_FLAG must be used with the DEFER QUIT configuration instruction. If the DEFER QUIT instruction is not specified, and quit signal will stop the program execution.

When the quit event arrives during a procedural instruction (FOR loop), the runtime system sets QUIT_FLAG to TRUE and continues the program execution. It is up to the program to check the QUIT_FLAG variable.

When the quit event arrives during an interactive instruction (INPUT, CONSTRUCT), the runtime system sets QUIT_FLAG to TRUE and continues with the execution of the interactive instruction.

Once QUIT_FLAG is set to TRUE, it must be reset to FALSE to detect a new quit event.

**Example**

```plaintext
MAIN
  DEFINE n INTEGER
  DEFER QUIT
  LET QUIT_FLAG = FALSE
  FOR n = 1 TO 1000
      IF QUIT_FLAG THEN EXIT FOR END IF
      ...
  END FOR
END MAIN
```

**Related concepts**

**INT_FLAG on page 569**

INT_FLAG is a predefined variable set to TRUE when an interruption event is detected.

---

**OOP support**

Describes Object Oriented Programming basics in the language.

**Understanding classes and objects**

The Genero language supports basic Object Oriented Programming (OOP) concepts.

Classes are grouped into packages which are: a) built-in and directly usable, and b) available as libraries which need to be imported with the IMPORT instruction.

It is not possible to define classes with the language.

**Related concepts**

**Built-in packages on page 2378**

These topics cover the built-in classes provided by the Genero Business Development Language.

**Extension packages on page 2726**

Several utility classes and functions are provided in additional packages.

**DEFINE ... package.class**

Object reference variables allow to manipulate class instances.

**Syntax:**

```
DEFINE variable package.class
```

1. `variable` is the variable that references the object.
2. *package* is the name of the package the class comes from.
3. *class* is the name of the class.

For more details about variables referencing objects, see Working with objects on page 571.

**Related concepts**
- DEFINE on page 390

A variable contains volatile information of a specific data type.

**Distinguish class and object methods**

Class methods can be invoked from the class, while object methods can only be invoked from the variable referencing the object.

Methods can be invoked like regular functions, by passing parameters and/or returning values, and can be used in expressions when they return a scalar value.

### Class methods

Class methods are called by using the class identifier as prefix, with the period as separator. The class identifier includes the package name and class name.

```plaintext
package.classname.method( parameter [\ldots] )
```

For example, to call the `refresh()` method of the `Interface` class, which is part of the `ui` package:

```plaintext
CALL ui.Interface.refresh()
```

### Object methods

Object methods are called through the variable referencing the object. To use object methods, the object must exist. Call the object methods by using the object variable as a prefix, with a period as the separator.

```plaintext
object.method( parameter [\ldots] )
```

For example, to call the `setFieldActive()` method of an object of the `Dialog` class, which is part of the `ui` package:

```plaintext
DEFINE d ui.Dialog
LET d = ui.Dialog.getCurrent()
CALL d.setFieldActive("cust_addr", FALSE)
```

**Working with objects**

This topic describes basic object usage in Genero BDL.

**Instantiating objects**

In order to instantiate an object in your program:

1. Define an object variable using the class identifier.
2. Instantiate the object; this is usually done by invoking a class method.

An object variable only contains a reference to the object. For example, when passed to a function, only the reference to the object is copied onto the stack.

In the following code example, the object referenced by the variable `n` is instantiated using the `create()` class method of the `DomDocument` class. The object referenced by the variable `b` is instantiated using the
getDocumentElement() object method of the DomDocument class. This method returns the DomNode object that is the root node of the DomDocument object referenced by n:

```plaintext
DEFINE n om.DomDocument, b DomNode
LET n = om.DomDocument.create("Stock")
LET b = n.getDocumentElement()
```

### Destroying objects

Objects created during program execution do not need to be explicitly destroyed. This is done automatically by the runtime system, based on a reference counter.

```plaintext
MAIN
    DEFINE d om.DomDocument
    LET d = om.DomDocument.create("Stock")  -- Reference counter = 1
END MAIN  -- d is removed, reference counter = 0 => object is destroyed.
```

When an object is referenced by several variables, an internal counter is incremented and decremented:

```plaintext
MAIN
    DEFINE d1, d2 om.DomDocument
    LET d1 = om.DomDocument.create("Stock")  -- Reference counter = 1
    LET d2 = d1                                -- Reference counter = 2
    LET d1 = NULL                             -- Reference counter = 1
    LET d2 = NULL                             -- Reference counter = 0, object is destroyed
END MAIN
```

### Using object references within functions

Object references can be passed to and returned from functions.

In this example, the function creates the object and returns its reference on the stack:

```plaintext
MAIN
    DEFINE x om.DomDocument
    LET x = createStockDomDocument()
END MAIN

FUNCTION createStockDomDocument()
    DEFINE d om.DomDocument
    LET d = om.DomDocument.create("Stock")  -- Reference counter = 1
    RETURN d
END FUNCTION  -- Reference counter is still 1 because d is on the stack
```

Another part of the program can get the result of that function and pass it as a parameter to another function:

```plaintext
MAIN
    DEFINE x om.DomDocument
    LET x = createStockDomDocument()
    CALL writeStockDomDocument( x )
END MAIN

FUNCTION createStockDomDocument()
    DEFINE d om.DomDocument
    LET d = om.DomDocument.create("Stock")
    RETURN d
END FUNCTION

FUNCTION writeStockDomDocument( d )
    DEFINE d om.DomDocument
```
### Invoking class and object methods

Class methods must be invoked with the package and class name:

```genero
DEFINE d om.DomDocument
LET d = om.DomDocument.create("Stock")
```

Object methods are invoked with the variable referencing the object:

```genero
DEFINE ch base.Channel
LET ch = base.Channel.create()
CALL ch.openFile("myfile.txt","r")
```

If a method returns an object reference, it can be directly used to invoke another method of the returned object:

```genero
DEFINE s STRING
LET s = "abc"
LET s = s.subString(1, 10).toLowerCase()
```

### What class packages exist?

A set of utility packages including useful classes are part of the distribution.

- **Built-in packages** such as `ui`, `om` and `base`, are part of the runtime system and can be referenced directly.
- **Extension packages** such as `util`, `os`, `com` and `xml` need to be loaded explicitly with the `IMPORT` instruction, at the beginning of program modules.

Genero supports usage of **Java classes and objects**. Note that using Java from a Genero program will create a Java Virtual Machine (JVM) that will be part of the runtime system process.

### XML support

Introduces to DOM and SAX standards and describes the XML utility classes built-in the language.

These classes are useful to perform basic XML processing and manipulate the abstract user interface tree.

Use the full-featured XML classes provided in the **web services extension** for other needs.

### DOM and SAX standards

DOM and SAX are both programming interfaces that can work with XML.

The **DOM** (Document Object Model) is a programming interface specification being developed by the World Wide Web Consortium (W3C) that lets a programmer create and modify HTML pages and XML documents as full-fledged program objects. DOM is a full-fledged object-oriented, complex but complete API, providing methods to manipulate the full XML document as a whole. DOM is designed for small XML trees manipulation.

The **SAX** (Simple API for XML) is a programming interface for XML, simpler than DOM. SAX is event-driven, streamed-data based, and designed for large trees.

### DOM and SAX built-in classes

The DOM and SAX APIs both contain a set of built-in classes.

The DOM API is composed of:
• The `om.DomDocument` class, that defines the interface to a DOM document. Instances of this class can be used to identify and manipulate an XML tree. `DOMNode` object manipulation methods are provided by this class.
• The `om.DomNode` class, that defines the interface to an DOM node. Instances of this class can be used to identify and manipulate a branch of an XML tree. Child nodes and node attributes management methods are provided by this class.
• The `om.NodeList` class, to handle a list of `DomNode` objects.

The SAX API is composed of:
• The `om.SaxAttributes` class represents a set of element attributes. It is used with an `om.XmlReader` or an `om.XmlWriter` object.
• The `om.XmlReader` class, that is defined to read XML. The XML document processing is based on SAX events.
• The `om.XmlWriter` class, that is defined to write XML. The XML document processing is based on SAX events.
• The `om.SaxDocumentHandler` class, which provides an interface to implement a SAX driver using functions defined in a .4gl module loaded dynamically.

**Limitations of XML built-in classes**
Built-in XML classes have some limitations you must be aware off.

The built-in XML classes are provided for convenience, to help you manipulate XML content easily without loading a complete external XML library such as Java XML classes or a C-based XML libraries.

The features of these built-in classes are limited to basic XML usage. For example, there is no DTD / XML Schema validation done; you can create the same attribute twice or set an invalid attribute value. You must take care to follow the definition of the XML document when using these classes.

For a complete XML support, use the full-featured XML classes provided in the [web services extension](#).

**Exception handling with XML classes**
Errors can occur while using XML built-in classes.

For example, calling methods of a SAX handler in an invalid order raises the runtime error `-8004`.

By default, the program stops in case of exception. XML errors can be trapped with the `WHENEVER ERROR` or `TRY/CATCH` exception handlers of Genero. If an error occurs during a method call of an XML class, the runtime system sets the `STATUS` variable.

This code example shows the trapping of XML classes errors.

```plaintext
MAIN
DEFINE w om.SaxDocumentHandler
LET w = om.SaxDocumentHandler.createFileWriter("sample.xml")
TRY
  CALL w.endDocument()
CATCH
  DISPLAY "ERROR: ", STATUS
END TRY
END
```

**Related concepts**
The `SaxDocumentHandler` class on page 2617
The `om.SaxDocumentHandler` class provides an interface to write an XML filter with events.

Exceptions on page 502
Describes exception (error) handling in the programs.

**Controlling the UI with XML classes**

The User Interface of a Genero application can be manipulated with the build-in XML API.

The runtime system represents the user interface of a program with a DOM tree.

User interface elements can be manipulated with the DOM and SAX built-in classes.

However, you must pay attention when modifying the AUI tree directly through the use of these classes. Invalid node or attribute creation can lead to unpredictable results.

**Related concepts**

The abstract user interface tree on page 1106

The abstract user interface tree is the XML representation of the application forms displayed to the end user.

DOM and SAX built-in classes on page 573

The DOM and SAX APIs both contain a set of built-in classes.

**JSON support**

Genero BDL supports JSON data manipulation.

**Related concepts**

Records on page 408

Records allow structured program variables definitions.

Arrays on page 418

Arrays (static or dynamic) allow you to handle an ordered collection of elements.

**What is JSON?**

JSON (JavaScript Object Notation) is a well known lightweight data-interchange format for JavaScript.

A JSON string (or object) is a comma-separated list of name/value pairs, with a : colon separating the key and the value. The list of name/value pairs is enclosed in {} curly brackets.

The names are delimited by double-quotes.

The value can be a single numeric value, a double-quotes string, an array, or a sub-element.

Arrays are defined by a comma-separated list of values enclosed in [] square brackets.

Sub-elements are defined inside {} curly brackets and define name/value pairs.

For example:

```json
{
   "cust_num":865234,
   "cust_name":"McCarlson",
   "order_ids":[234,3456,24656,34561],
   "address":{
      "street":"34, Sunset Bld",
      "city":"Los Angeles",
      "state":"CA"
   }
}
```

For more details, see [http://www.json.org](http://www.json.org).
JSON utility classes

Genero BDL provides utility classes to manipulate JSON formatted data.

JSON classes

The following JSON utility classes are provided as an extension:

- The util.JSON class on page 2766 implements basic conversion methods.
- The util.JSONObject class on page 2773 implements detail JSON Object control.
- The util.JSONArray class on page 2783 implements detail JSON Array control.

BDL/JSON conversion basics

When the data structures and member names match, Genero BDL variables can be converted from/to JSON data with the util.JSON* utility classes.

Matching BDL and JSON data structures

In order to convert a BDL variable to/from a JSON string, the program RECORD or DYNAMIC ARRAY and the JSON data string must have the same structure.

JSON object elements and BDL RECORD member are associated by name, not by position. Elements in the JSON string and in the BDL variable can be at a different ordinal position.

JSON array elements and BDL DYNAMIC ARRAY elements are associated by position.

A JSON object can also be converted to a BDL DICTIONARY, when the JSON object is a list of named elements, using the same structure as the dictionary.

Example of BDL data structure:

```sql
DEFINE rec RECORD
  pkey INT,
  name VARCHAR(50),
  arr DYNAMIC ARRAY OF STRING,
  dic DICTIONARY OF DECIMAL
END RECORD

LET rec.pkey = 999
LET rec.name = "Tim Birton"
LET rec.arr[1] = "item1"
LET rec.arr[2] = "item2"
LET rec.dic["abc"] = 14.45
LET rec.dic["def"] = 18.11
```

JSON equivalent:

```
{
  "pkey": 999,
  "name": "Tim Birton",
  "arr": ["item1","item2"],
  "dic": {
    "def": 18.11,
    "abc": 14.45
  }
}
```

BDL to JSON conversion

BDL variables can be converted to JSON strings for example with the util.JSON.stringify() method.
The JSON elements get the same names of the record members, as defined in the program source. For more details about BDL to JSON names handling, see BDL names and JSON element names on page 577.

Program array members in the record are converted to JSON arrays delimited by square brackets (\[\]). Special consideration needs to be taken regarding empty dynamic arrays records where all elements are null. The Genero JSON API provides options to control the production of JSON elements for empty records and array. For more details, see NULLs and empty structures on page 578.

For details about BDL to JSON data type conversion rules, see BDL to JSON type conversion rules on page 580.

**JSON to BDL conversion**

When conversion from JSON to BDL, elements in the JSON string that do not match a Genero BDL record member are ignored; no error is thrown if there is no corresponding Genero BDL member.

Genero BDL record members that have no matching JSON element are initialized to NULL.

The JSON value must match the data format of the destination member. If the value does not correspond to the type (for example, if the JSON value is a character string while the target record member is defined with a numeric type), the target member will be set to NULL.

JSON arrays delimited by square brackets are used to fill a program array of the destination record. The destination array should be a dynamic array. If the array is defined as static, the additional elements of the source JSON array will be discarded, while missing elements will be initialized to NULL.

The JSON source string must follow the JSON format specification. It can contain multilevel structured data. If the source string is not well formatted, the runtime system will throw error -8109.

For details about JSON to BDL data type conversion rules, see JSON to BDL type conversion rules on page 581.

**BDL names and JSON element names**

To identify elements, JSON standards use different format as Genero BDL variable names.

As in many other programming languages, Genero BDL variables are named with simple identifiers. These identifiers are case-insensitive, must start with a letter or underscore, and cannot contain special characters such as spaces.

By default, JSON/BDL element name matching is case-insensitive. For example, if the Genero BDL record member is defined as CustNo, and the JSON data string contains the "custno":999 name/value pair, the value will be assigned. However, since Genero BDL record member names are used as-is to write JSON data, it is strongly recommended to define the Genero BDL records with the exact names used in JSON data string. Since JSON is case-sensitive, make sure the names of the Genero BDL record members match exactly the names expected in the resulting JSON data string: CustNo will be different from custNo.

**Important:** JSON specifications allow you to define element names with characters that cannot be used in Genero BDL identifiers. For example, a JSON element name can be "customer.name" or "customer:name". To work around this issue, use the json_name attribute when defining the BDL variable.

If the JSON element name cannot be defined as a Genero BDL variable identifier, it is possible to define the BDL variable with the json_name attribute, to specify the exact name of the corresponding JSON element.

In the next example, the BDL variable cust_name will be mapped to the JSON element "Customer Name":

```
DEFINE cust_name INTEGER ATTRIBUTES(json_name="Customer Name")
```

**Note:** When converting JSON to BDL structures, elements in the JSON string that do not match an Genero BDL record member are ignored; no error is thrown if there is no corresponding Genero BDL member.

**Related concepts**

Definition attributes on page 398
Variables can be defined with meta-data information.

**NULLs and empty structures**

Unlike Genero BDL, JSON distinguishes NULL, empty and undefined elements.

**JSON notation for NULL, empty and undefined**

In JSON notation, a NULL element is defined with the `null` keyword. In the next example, the "name" element is null:

```json
{ "key":345, "name":null }
```

Empty JSON objects are represented with an opening followed by a closing curly brace:

```json
{ "key":345, "address":{} }
```

Empty JSON arrays are represented with an opening followed by a closing square brace:

```json
{ "key":345, "list":[] }
```

Undefined elements do not appear in the JSON string representation. In the next example, the corresponding BDL record could have a "key" and "name" member. In the JSON notation, the "name" element is just omitted:

```json
{ "key":345 }
```

**States of Genero BDL variables**

Genero BDL variables defined with a primitive type such as INTEGER can have a value or can be NULL, but they cannot be undefined as in JSON.

RECORD variables cannot be NULL. However, all elements of a RECORD can be NULL and thus the record can be considered as empty or null.

A DYNAMIC ARRAY containing zero elements can be considered as empty or null.

**Controlling JSON serialization from BDL**

By default, the `util.JSON.stringify()` method writes all elements of the BDL variable.

Sub-records and dynamic arrays are written to the JSON output:

```bdl
DEFINE rec RECORD
  field1 INTEGER,
  subarr DYNAMIC ARRAY OF INTEGER
END RECORD
INITIALIZE rec.* TO NULL
LET rec.field1 = 999
DISPLAY util.JSON.stringify(rec)
```

Produces following output:

```json
{"field1":999,"subarr":[]}
```

If you want to omit all empty RECORD or DYNAMIC ARRAY elements in the JSON string, use the `util.JSON.stringifyOmitNulls()` method:

```bdl
IMPORT util
MAIN
DEFINE rec RECORD
```
field1 INTEGER,
subarr DYNAMIC ARRAY OF INTEGER
END RECORD
INITIALIZE rec.* TO NULL
LET rec.field1 = 999
DISPLAY util.JSON.stringifyOmitNulls(rec)
END MAIN

Produces following output (the subarr element is omitted because it is empty):

{"field1":999}

If you want to have a fine-grained control on the JSON serialization of null values and empty records and dynamic arrays, use the `util.JSON.stringify()` method, in conjunction with the `json_null` attribute when defining the BDL variable. The `json_null` variable definition attribute can be set to "null" or "undefined".

**Using the json_null="null" attribute**

When defining a BDL variable with the `json_null="null"` attribute, the following BDL elements will be represented with the `null` keyword in the resulting JSON string:

1. Simple primitive variables that are NULL,
2. RECORD variables where all members are NULL,
3. Empty DYNAMIC ARRAY that contain zero elements.

In the next example, the sub-record and sub-array are written to the JSON output as nulls:

```plaintext
DEFINE rec RECORD
    field1 INTEGER ATTRIBUTES(json_null="null"),
    field2 CHAR(1) ATTRIBUTES(json_null="null"),
    subrec1 RECORD ATTRIBUTES(json_null="null")
        field11 INTEGER,
        field12 VARCHAR(30)
    END RECORD,
    subarr1 DYNAMIC ARRAY ATTRIBUTES(json_null="null") OF INTEGER
END RECORD
INITIALIZE rec.* TO NULL
DISPLAY util.JSON.stringify(rec)
```

Produces following output:

{"field1":null,"field2":null,"subrec1":null,"subarr1":null}

**Using the json_null="undefined" attribute**

When defining a BDL variable with the `json_null="undefined"` attribute, the following BDL elements will be omitted in the resulting JSON string:

1. Simple primitive variables that are NULL (in fact, `json_null="undefined"` is the default for primitives),
2. RECORD variables where all members are NULL,
3. Empty DYNAMIC ARRAY that contain zero elements.

In the next example, all elements of the record (except field1) are omitted in the resulting JSON output:

```plaintext
DEFINE rec RECORD
    field1 INTEGER ATTRIBUTES(json_null="undefined"),
    field2 CHAR(1) ATTRIBUTES(json_null="undefined"),
    subrec1 RECORD ATTRIBUTES(json_null="undefined")
        field11 INTEGER,
        field12 VARCHAR(30)
    END RECORD,
END RECORD
```
**Advanced features**

---

**580**

**subarr1 DYNAMIC ARRAY ATTRIBUTES(json_null="undefined") OF INTEGER**

END RECORD

INITIALIZE rec.* TO NULL

LET rec.field1 = 999

DISPLAY util.JSON.stringify(rec)

Produces following output:

```
{"field1":999}
```

---

**Exception of NULL dynamic array elements**

If elements of a dynamic array are NULL, the JSON serialization class will always produce a `null` keyword, even if you specify the `json_null="undefined"` attribute at the primitive type level.

This is mandatory because JSON requires `null` keyword for undefined array elements.

See the following example:

```
DEFINE rec RECORD
    subarr DYNAMIC ARRAY OF INTEGER ATTRIBUTES(json_null="undefined")
END RECORD

INITIALIZE rec.* TO NULL

LET rec.subarr[3] = 999

DISPLAY util.JSON.stringify(rec)
```

Produces this JSON output:

```
{"subarr":null,null,999}
```

---

**BDL to JSON type conversion rules**

Specific type conversion rules apply when converting a BDL variable to JSON.

**Table 162: Genero BDL to JSON type conversion rules**

<table>
<thead>
<tr>
<th>Source Genero BDL type</th>
<th>JSON result string</th>
</tr>
</thead>
<tbody>
<tr>
<td>RECORD .. END RECORD</td>
<td>The JSON string will be a JSON object in the form:</td>
</tr>
<tr>
<td></td>
<td>{ &quot;record-element-name&quot; : json-value [, ...] }</td>
</tr>
<tr>
<td>DYNAMIC ARRAY OF ...</td>
<td>The JSON string will be a JSON array in the form:</td>
</tr>
<tr>
<td></td>
<td>[ json-value [, ...] ]</td>
</tr>
<tr>
<td>DICTIONARY OF ...</td>
<td>The JSON string will be a JSON object in the form:</td>
</tr>
<tr>
<td></td>
<td>{ &quot;dictionary-key&quot; : json-value [, ...] }</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>Will be serialized with the JSON values true or false.</td>
</tr>
</tbody>
</table>
Source Genero BDL type | JSON result string
---|---
TINYINT, SMALLINT, INTEGER, BIGINT, SMALLFLOAT, FLOAT, DECIMAL, MONEY | Any numeric type will be serialized to this form: an optional minus sign (-), a sequence of digits (0–9), containing a optional decimal separator (.), followed by an optional exponent. The exponent has the form (e) followed by an optional minus sign and an sequence of digits. The representation of numeric values does not depend from the current locale. The decimal separator is always a dot (.). MONEY values will be represented like DECIMAL values: the currency symbol will be omitted.

DATE | The date value will be formatted as "YYYY-MM-DD" (with double quotes)

DATETIME | The date-time value will be formatted as "YYYY-MM-DD hh:mm:ss.ffffff" (with double quotes), based on the date-time type definition. For example, a DATETIME HOUR TO MINUTE will produce "hh:mm" formatted values.

Note that the YYYY-MM-DD hh:mm:ss.ffffff format is used to represent the local time. When exchanging date-time values in communications across different time zones, consider converting date-time values to Coordinated Universal Time (UTC), by using the util.Datetime methods.

INTERVAL | The interval value will be formatted as "YYYY-MM" or "DD hh:mm:ss.ffffff" (with double quotes), depending on the interval type definition.

BYTE | Will be serialized to a Base64 encoded double quoted string. The Base64 encoding is described in [RFC4648].

TEXT, CHAR, VARCHAR, STRING | Character string data will be serialized as a double quoted string with backslash escaping.

List of characters requiring escaping:

<table>
<thead>
<tr>
<th>Character</th>
<th>Unicode</th>
</tr>
</thead>
<tbody>
<tr>
<td>\</td>
<td>U+005C</td>
</tr>
<tr>
<td>&quot;</td>
<td>U+0022</td>
</tr>
<tr>
<td>\b</td>
<td>U+0008</td>
</tr>
<tr>
<td>\f</td>
<td>U+000C</td>
</tr>
<tr>
<td>\n</td>
<td>U+000A</td>
</tr>
<tr>
<td>\r</td>
<td>U+000D</td>
</tr>
<tr>
<td>\t</td>
<td>U+0009</td>
</tr>
</tbody>
</table>

Other | Any other type will be serialized as a double quoted (") string.

### JSON to BDL type conversion rules

Specific type conversion rules apply when parsing a JSON string to fill a BDL variable.

**Table 163: JSON to Genero BDL type conversion rules**

<table>
<thead>
<tr>
<th>JSON source string</th>
<th>Target Genero BDL type</th>
</tr>
</thead>
<tbody>
<tr>
<td>The JSON string must be a JSON object in the form:</td>
<td>RECORD ... END RECORD</td>
</tr>
</tbody>
</table>
| ```json
{ "record-element-name" : json-value [, ...] }
``` | |
| The JSON string must be a JSON array in the form: | DYNAMIC ARRAY OF ... |
| ```json
[ json-value [, ...] ]
``` | |
<table>
<thead>
<tr>
<th>JSON source string</th>
<th>Target Genero BDL type</th>
</tr>
</thead>
<tbody>
<tr>
<td>The JSON string must be a JSON object in the form:</td>
<td>DICTIONARY OF ...</td>
</tr>
<tr>
<td><code>{ &quot;dictionary-key&quot; : json-value ... }</code></td>
<td>BOOLEAN</td>
</tr>
</tbody>
</table>

The JSON value should be `null`, `true` or `false`. If the JSON value is a number or a string, the language conversion rules from number/string to `BOOLEAN` apply.

A JSON number. The JSON number can be assigned to any language numeric type. The limits of the target type cause potential overflows errors. On error the target variable will be initialized to `NULL`, the parser continues without an error.

The JSON value must be a string formatted as "YYYY-MM-DD".

A JSON string representing a datetime or a JSON number.

If the value is a JSON string, it must be formatted as "YYYY-MM-DD hh:mm:ss.ffffff", or represent as an ISO 8601 formatted date-time, in UTC (with Z indicator) or with a timezone offset (+/-hh:mm). For example: "2013-02-21T15:18:44.456Z", "2013-02-21T20:18:44.456+02:00".

If the value is a JSON number, it is interpreted as UNIX™ time (seconds since the Epoch 00:00:00 UTC, January 1, 1970).

Note that the YYYY-MM-DD hh:mm:ss.ffffff format is used to represent the local time. When exchanging date-time values in communications across different time zones, consider converting date-time values to Coordinated Universal Time (UTC), by using the `util.Datetime` methods.

**Note:** The `util.JSON` parser also supports Microsoft JSON AJAX old-style datetime values: `/\Date\((nnn)(Z|\+[\-]hhmm))\/`, where `nnn` is a number of milliseconds since epoch. `Z` indicates UTC and `+-hhmm` defines a timezone offset. Note that in the JSON representation of the AJAX date value, backslashes need to be escaped: "\\Date(76523465)\\".

The JSON value must be a string formatted as "YYYY-MM" or "DD hh:mm:ss.ffffff", depending on the interval class of the target variable.

The JSON string value must be encoded in Base64.

The Base64 encoding is described in [RFC4648].

A JSON string or number.

If the JSON value is a number, the resulting BDL string value uses the locale specific decimal point.

If the JSON value is a string: Any character in the Basic Multilingual Plane (U+0000 through U+FFFF) may be escaped: \u followed by exactly 4 hexadecimal digits ([0-9a-fA-F]). The hexadecimal digits encode the code point. Characters outside the Basic Multilingual Plane may be escaped by there UTF-16 surrogate pairs. For example, the representation of the G clef character (U+1D11E) is "\uD834\uDD1E".

**Note:** When parsing a JSON string to fill a `TEXT` or `BYTE` variable, if the data storage for the LOB variable has not been defined with the `LOCATE` instruction, the JSON methods will automatically locate the `TEXT` or `BYTE` in memory. This applies also to `TEXT` and `BYTE` elements of records, arrays and dictionaries.
Front calls

Front call functions execute on the platform where the front-end is installed.

Related concepts

Built-in front calls on page 2635
This section contains the description of all built-in front calls.

User-defined front calls on page 2249
Front-ends can be extended with custom functions to access specific features.

Understanding front calls

Front calls execute a native function on the front-end platform.

In your Genero program, use the `ui.Interface.frontCall()` class method to invoke front-end functions.

When calling a user function from programs, specify a module name and a function name. Input and output parameters can be passed/returned in order to transmit/receive values to/from the front-end. A typical example is an "open file" dialog window that allows you to select a file from the front-end workstation file system.

**Important**: Front calls can be specific to the platform or front-end technology and may not be supported. For example, it is not possible to execute a shell command (`shellexec`) with the Web Browser front-end.

A set of front-end functions is built-in by default in front-ends. However, it is possible to write your own functions in order to extend the front-end possibilities.

**Tip**: While you can use DDE/OLE APIs to manipulate Microsoft™ Office documents, there are freeware alternatives such as the Apache POI Java library which can be used with the Java Interface. For an example, see Java Interface: Example 2.

Related concepts

The abstract user interface tree on page 1106
The abstract user interface tree is the XML representation of the application forms displayed to the end user.

**ui.Interface.frontCall**

`ui.Interface.frontCall` performs a function call to the current front-end.

Syntax

```
ui.Interface.frontCall(
    moduleName STRING,
    functionName STRING,
    [ valueList ],
    [ variableList ]
)
```

1. `moduleName` defines the shared library or classpath where the function is implemented.
2. `functionName` defines the name of the function to be called.
3. `valueList` is a list of input parameters.
4. `variableList` is a list of output parameters.

**Important**: The variables for output parameters are passed by reference to the `frontCall()` method.

Usage

The `ui.Interface.frontCall()` class method can be used to execute a procedure on the front-end workstation through the front-end software component. You can for example launch a front-end specific application like a browser or a text editor, or manage the clipboard content.

**Important**: When calling the `ui.Interface.frontCall()` method, the connection to the front-end is initiated, if it is not yet established. Consider avoiding front calls in batch programs and interactive programs using
the text mode. This is also important to consider in graphical mode, if no interactive instruction was issued before the front call. Furthermore, each front call will sync the AUI tree with the front end.

The method takes four parameters:

1. The module, identifying the shared library (.so or .DLL) or the Java class (GMA) implementing the front call function.
2. The function of the module to be executed.
3. The list of input parameters, using the square brace notation.
4. The list of output parameters, using the square brace notation.

Input and output parameters are provided as a variable list of parameters, by using the square brackets notation ([param1, param2, ...]):

- Input and output parameters can be of any simple type like INTEGER, a RECORD or a DYNAMIC ARRAY.
- An empty list of input or output parameters is specified with [].
- Input parameters can be an expression such as (10 * var).
- Output parameters must be variables only, to receive the returning values.
- Output parameters are optional. If the front call returns values, these values will be ignored by the runtime system, if no output parameters are provided to receive these values.

**Simple front call example:**

```plaintext
FUNCTION call()
  DEFINE info STRING
  CALL ui.Interface.frontCall( "standard", "feInfo", ["feName"], [info] )
END FUNCTION
```

Some front calls need a file path as parameter. File paths must follow the syntax of the front end workstation file system. You may need to escape backslash characters in such parameters. The following example shows how to pass a file path with a space in a directory name to a front-end running on a Microsoft® Windows® workstation:

```plaintext
FUNCTION call()
  DEFINE path STRING, res INTEGER
  LET path = "\c:\work dir\my report.doc"
  -- This is: "c:\work dir\my report.doc"
  CALL ui.Interface.frontCall( "standard", "shellExec", [path], [res] )
END FUNCTION
```

When using RECORD and DYNAMIC ARRAY as front call input or output parameters, the runtime system will use JSON serialization, to pass and return such structured data to/from the front-end. This is important to know when implementing your own custom front calls. Note that one can use the `json_null` and `json_name` variable definition attributes to control JSON serialization:

```plaintext
DEFINE optrec RECORD
  mode INTEGER ATTRIBUTES(json_null="null"),
  filter STRING ATTRIBUTES(json_name="Data Filter")
END RECORD
DEFINE flags DYNAMIC ARRAY OF INTEGER ATTRIBUTES(json_null="undefined")
DEFINE result_list DYNAMIC ARRAY OF STRING
LET optrec.mode = 999
LET optrec.filter = "*A*
LET flags[1] = 111
LET flags[3] = 333
CALL ui.Interface.frontCall( "m1", "fc1", [optrec, flags], [result_list] )
```

**Front call cost**

A front call is a remote procedure call requiring a full network round trip between the server app and the front-end.
Depending on the current network speed, this may result in delays in the millisecond to sub second range.

**Note:** In mobile application development or `runOnServer` mode, the execution time of a front can be much slower when running the app on the server, compared to embedded apps.

**Front call error handling**

Exception handling instructions can be used to check the execution status of a front call. Both `WHENEVER ERROR` directives or `TRY/CATCH` blocks can surround the front call to avoid program stopping in case of error, and to check the error number returned in the `STATUS` variable.

**Note:** There is no need to surround front calls with exception handlers such as `TRY/CATCH`, if the front call is always supposed to execute without error. For example, the `feInfo` front call will never produce an exception.

Example of front call error handling with a `TRY/CATCH` block:

```gen
FUNCTION takePhoto()
    DEFINE path STRING
    TRY -- This front call may fail if the front-end is not a mobile device:
        CALL ui.Interface.frontCall( "mobile", "takePhoto", [], [path] )
    CATCH
        MESSAGE "Cannot take photo: ", STATUS, " ", err_get(STATUS)
        LET path = NULL
    END TRY
    RETURN path
END FUNCTION
```

If the front call module name or the function name is invalid, the errors -6331 or -6332 will be raised, respectively.

If the front call execution fails for some reason, the error -6333 will be raised. The description of the problem can be found in the second part of the error message, returned by a call to the `ERR_GET()` function.

The error -6334 can be raised in case of input or output parameter mismatch. The control of the number of input and output parameters is in the hands of the front-end. Most of the standard front calls have optional returning parameters and will not raise error -6334, if the output parameter list is left empty. However, front-end specific extensions or user-defined front-end functions may return an invalid execution status in case of input or output parameter mismatch, raising error -6334. If the front-end sends a call execution status of zero (OK), and the number of returned values does not match the number of program variables, the runtime system will set unmatched program variables to `NULL`. As a general rule, it is recommended that the program provides the expected input and output parameters as specified in the documentation.

**Related concepts**

- Front calls on page 583
- Front call functions execute on the platform where the front-end is installed.

**User-defined front calls**

Extend the Genero language possibilities by implementing your own front-end functions.

For more details, see User-defined front calls on page 2249.
Optimization

Programming tips and tricks to make your programs run faster.

Runtime system basics

This section contains topics about Genero BDL runtime system basic concepts.

Dynamic module loading

Type of Genero programs

A Genero Business Development Language program is made of several .42m modules.

A program can be created by linking modules together (to get a .42r file), or by defining the dependency between modules with IMPORT FGL (and have only .42m program files); in this case no linking is required.

When are .42m modules loaded?

As a general rule, .42m program modules are loaded at runtime on demand, when a module element (i.e symbol) is required by the caller.

For example, when executing a CALL instruction, the runtime system checks if the module implementing the function is already in memory. If not, the module is first loaded, then module variables are initialized, and then the function is called.

Note: When using the debugger, all modules are loaded implicitly when starting the debugger, for both linked programs and programs using imported modules.

Overwriting .42m modules in production environments

Modules loaded by running programs are not affected by file replacements: Programs continue to run with an image of the module file that was originally loaded.

However, replacing program modules during execution should be used with care: Since .42m modules are loaded on demand (when a symbol of the module is referenced), some modules of a running program may not yet be loaded.

When replacing a module while programs are running, invalid symbol errors can occur if the module to be loaded does not correspond to the rest of the program modules that were loaded before the file replacements.

See following scenario:

1. Program starts with V1 of main.42m, needing V1 of module libutil.42m (loaded later on demand).
2. Administrator upgrades application and installs main.42m and libutil.42m version V2.
3. Program running with V1 copy of main.42m calls a function from libutil.42m: runtime loads V2 of that module, while V1 is expected.

When live application updates are mandatory, consider installing new program and resource files (V2) in a different directory as the currently running version (V1), and use the FGLLDPATH and FGLRESOURCEPATH environment variables to point to the new files when starting a new (V2) program instance.

Note that on Windows™ platforms, program files currently in use cannot be overwritten, because of Windows™ OS memory mapping limitations. You need to turn off memory mapping with the FGLPROFILE entry fglrun.mmapDisable.

Elements shared by multiple programs

The (.42m) p-code module instructions and other elements such as constants are shared among several programs running on the same machine.

Localized string resource files (.42s) are also shared among all fglrun processes running on a computer.
These files are loaded with the system memory mapping facility, which allows multiple processes to access the same unique memory area.

**Elements shared by multiple modules**

By definition, global variables are visible to all modules of a program, and thus shared among all modules of the program. While global variables are an easy way to share data among multiple modules, it is not recommended that you use too many global variables.

The data type definitions are only defined once in memory and shared by all modules of a program instance. By data type definition we mean the type descriptions, not the data itself. This applies only to the equivalent data types used in different modules.

**Objects private to a program**

Program objects such as global variables, module variables as well as resources used by the user interface and SQL connections and cursors, are private to a program.

This implies that each of these objects requires private memory to be allocated. If memory is an issue, do not allocate unnecessary resources. For example, don't create windows / load forms or declare / prepare cursors until these are really needed by the program. When the resource is not longer needed, consider freeing them (CLOSE WINDOW, FREE cursor).

**Optimize your programs**

This section contains programming tips to optimize the execution of your application.

**Finding program bottlenecks**

The best way to find out why a program is slow (and also, to optimize an already fast-running program), is to use the profiler.

This tool is included in the runtime system, and generates a report that shows what function in your program is the most time-consuming.

Additionally, you might want to identify part of code of your programs that are never executed, or executed very often, by using the coverage tool.

**Optimizing SQL statements**

SQL statement execution is often the code part of the program that consumes a lot of processor, disk and network resources. Therefore, it is critical to pay attention to SQL execution.

Advice for this can be found in SQL Programming.

**Passing CHAR parameters to functions**

Simple data types such as CHAR are passed by value to functions (this means the value of the caller variable is copied on the stack, and then copied back into a local variable of the called function.) When large data types are used, this can introduce a performance issue.

For example, the following code defines a logging function that takes a CHAR(2000) as parameter:

```sql
FUNCTION log_msg( msg )  
    DEFINE msg CHAR(2000)  
    CALL myLogChannel.writeLine(msg)  
END FUNCTION
```

The function is then typically used by passing a log message as parameter:

```sql
CALL log_msg( "Start processing..." )
```

When performing this call, the runtime system copies 19 characters on the stack, calls the function, and then copies the value into the local variable. Since the values in CHAR variables must always have a length matching the variable
definition size, the runtime system fills the remaining 1981 positions with blanks. As result, each time you call this function, a 2000 character-long variable is created on the stack.

By using a `VARCHAR(2000)` (or a `STRING`) data type in this function, you optimize the execution, because no trailing blanks need to be added for these types.

Consider also using `VARCHAR` instead of `CHAR` for large database columns. Due to the Informix® SQL history and the Informix® `VARCHAR(255)` limit, there may be `CHAR()` columns when the storage size required is more than 255 bytes. For more details, see Trailing blanks in CHAR/VARCHAR on page 620.

**Related concepts**
Manipulating character strings on page 588

**Manipulating character strings**

When parsing, concatenating or accessing parts of a character string contained in `CHAR`, `VARCHAR` and `STRING` variables, the runtime system must follow the BDL language semantics, which can result in performance issues with very large character strings.

When heavy string manipulation is to be done, consider using the `base.StringBuffer` class instead of `CHAR/VARCHAR/STRING` variables: The `base.StringBuffer` class is designed for character string manipulation.

Furthermore, execution time of character iterations (to find a specific sub-string for example or get the position of a character in a string), can be affected when using the UTF-8 locale and `CHAR length semantics usage`: The runtime needs to compute the number of bytes representing a character when iterating through the string.

**Related concepts**

Passing CHAR parameters to functions on page 587

**Passing records by reference to functions**

When using the `record-name.*` notation, RECORD structures are passed to functions by copying all record members on the stack.

When the record is big, this can lead to performances issues when the function is called many times, or when chaining function calls passing complete records to each other. This is even more resource consuming, when using `CHAR()` types.

The next code example shows the execution time improvement when passing a record by reference:

```plaintext
TYPE t_customer RECORD
    cust_id INTEGER,
    cust_name CHAR(100),
    cust_addr CHAR(100),
    cust_comm CHAR(500)
END RECORD

MAIN
DEFINE x INTEGER
DEFINE ts DATETIME HOUR TO FRACTION(5)
DEFINE cust t_customer

LET cust.cust_id = 101
LET cust.cust_name = "Scott Pilgrim"
LET cust.cust_addr = "5 Market Place"
LET cust.cust_comm = "..."

LET ts = CURRENT
FOR x = 1 TO 1000000
    CALL func1( cust.* )
END FOR
DISPLAY CURRENT HOUR TO FRACTION(5) - ts
```
LET ts = CURRENT
FOR x = 1 TO 1000000
    CALL func2( cust )
END FOR
DISPLAY CURRENT HOUR TO FRACTION(5) - ts
END MAIN

FUNCTION func1( rec_t_customer )
    CALL func11( rec.* )
END FUNCTION

FUNCTION func11( rec_t_customer )
    CALL func111( rec.* )
END FUNCTION

FUNCTION func111( rec_t_customer )
END FUNCTION

FUNCTION func2( rec_t_customer INOUT )
    CALL func21( rec )
END FUNCTION

FUNCTION func21( rec_t_customer INOUT )
    CALL func211( rec )
END FUNCTION

FUNCTION func211( rec_t_customer INOUT )
END FUNCTION

The above program produces following output:

0:00:03.70940
0:00:00.57024

Note: Records passed by reference can be modified by the function. For more details, see Passing records by reference on page 446

Compiler removes unused variables

When declaring a large static array without any reference to that variable in the rest of the module, you will not see the memory grow at runtime. The compiler has removed its definition from the 42m module.

To get the defined variable in the .42m module, you must at least use it once in the source (for example, with a LET statement). Note that memory might only be allocated when reaching the lines using the variable.

Saving memory by splitting modules

Program modules (.42m) are loaded dynamically on demand. If a program only needs some independent functions of a given module, all module resources will be allocated just to call these functions. By independent, we mean functions that do not use module objects such as variables defined outside function or SQL cursors. To avoid unnecessary resource allocation, you can extract these independent functions into another module and save a lot of memory at runtime.

If you are using .42x libraries, it is recommended that you create libraries with the 42m modules that belong to the same functionality group. For example, group all accounting modules together in an accounting library. By doing this, programmers using the .42x libraries are not dependent from module reorganizations.

Libraries are supported for backward compatibility, it is recommended that you consider using the IMPORT FGL instruction to define module dependency and get modules loaded dynamically when needed.

Saving memory by using STRING variables

The CHAR and VARCHAR data types are provided to hold string data from a database column. When you define a CHAR or VARCHAR variable with a length of 1000, the runtime system must allocate the entire size, to be able to fetch SQL data directly into the internal string buffer.
For character string data that is not stored in the database, consider using the `STRING` data type. The `STRING` type is similar to `VARCHAR`, except that you don't need to specify a maximum length and the internal string buffer is allocated dynamically as needed. Thus, by default, a `STRING` variable initially requires just a bunch of bytes, and grows during the program life time, with a limitation of 65534 bytes.

Use of a `STRING` variable is typically recommended when building SQL statements dynamically, for example from a `CONSTRUCT` instruction. You may also use the `STRING` type for utility function parameters, to hold file names for example.

After a large `STRING` variable is used, it should be cleared with a `LET` or an `INITIALIZE TO NULL` instruction. However, this is only needed for `STRING` variables declared as global or module variables. The variables defined in functions will be automatically destroyed when the program returns from the function.

Use of the `base.StringBuffer` build-in class is recommended when a great deal of string manipulation and modification is required. String data is not copied on the stack when an object of this class is passed to a function, or when the string is modified with class methods. This can have a big impact on performance when very large strings are processed.

### Saving memory by using dynamic arrays

The language supports both static arrays and dynamic arrays. For compatibility reasons, static arrays must be allocated in their entirety. This can result in huge memory usage when big structures are declared, such as:

```plaintext
DEFINE arr_cust ARRAY[5000] OF RECORD
  id INTEGER,
  name CHAR(50),
  address CHAR(200),
  comment CHAR(2000)
END RECORD
```

If possible, replace such static arrays with dynamic arrays:

```plaintext
DEFINE arr_cust DYNAMIC ARRAY OF RECORD
  id INTEGER,
  name CHAR(50),
  address CHAR(200),
  comment CHAR(2000)
END RECORD
```

However, be aware that dynamic arrays have a slightly different behavior than static arrays.

### SQL support

These topics cover SQL support in the Genero Business Development Language.

### SQL programming

Covers topics about interacting with a database server using SQL.

### SQL basics

This section contains fundamental information to know about SQL programming with Genero BDL.

#### Database and application locales

Choose the right locale for your database and application programs.

When designing a database and the application programs that connect to this database, it is important to consider what character set will be used in the database server, and the character set used by application programs.
Most database servers support character set conversion options, thus it is possible to have different configurations, for example:

1. Database uses ISO88591, application programs use ISO88591
2. Database uses UTF-8, application programs use ISO88591
3. Database uses UTF-8, application programs use UTF-8

**Important:** The locale definition of the database client software must match the application locale defined for Genero. For more details, see Database client settings on page 526.

Consider also the length semantics to be used (is CHAR(10) 10 bytes or 10 characters?).

For a detailed discussion about character sets and length semantics, see CHAR and VARCHAR types on page 616.

**Related concepts**

*Application locale* on page 512

The *application locale* defines the language and codeset for your application.

**SQL execution diagnostics**

If an SQL statement execution fails, error description can be found in the SQLCA.SQLCODE, SQLSTATE, STATUS and SQLERRMESSAGE predefined registers.

**Trapping SQL errors**

By default, SQL errors stop program execution and display the error message to the standard output. Most SQL statements executed by a program should not return an error and thus do not require error trapping. However, in some cases, a program must keep the control when an SQL error occurs. For example, when connecting to the database, the user might enter an invalid password that will raise a login denied error. The program must trap such SQL connection error in order to return to the login dialog and let the user enter a new login and password.

To trap potential SQL errors, surround the SQL statements to be checked either with a WHENEVER ERROR exception handler or with a TRY / CATCH block:

```sql
-- WHENEVER ERROR handler
WHENEVER ERROR CONTINUE
    INSERT INTO orders VALUES ( rec_ord. * )
    IF SQLCA.SQLCODE = -75623 THEN
        ...
    END IF
WHENEVER ERROR STOP -- restore the default

-- TRY/CATCH block
TRY
    INSERT INTO orders VALUES ( rec_ord. * )
CATCH
    IF SQLCA.SQLCODE = -75623 THEN
        ...
    END IF
END TRY
```

**Using SQLCA.SQLCODE**

SQL error codes are provided in the SQLCA.SQLCODE register. This register always contains an IBM® Informix® error code, even when connected to a database that is different from IBM® Informix®.

STATUS is the global error code register that can be set for any kind of error (even non-SQL). When an SQL error occurs, both SQLCA.SQLCODE and STATUS hold the SQL error code.

Use SQLCA.SQLCODE for SQL error management, and use STATUS to detect errors with other language instructions.
When connecting to a database that is different from IBM® Informix®, the database driver tries to convert the native SQL error to an IBM® Informix® error which will be copied into the SQLCA.SQLCODE and STATUS registers. If the native SQL error cannot be converted, SQLCA.SQLCODE and STATUS will be set to -6372 (a general SQL error). You can then check the native SQL error in SQLCA.SQLERRD[2]. The native SQL error code is always available in SQLCA.SQLERRD[2], even if it cannot be converted to an IBM® Informix® error.

Using SQLSTATE

SQLSTATE is a register that contains an error code following ISO standards. However, not all database servers support this standard.

Preferably use SQLSTATE for SQL error checking, as long as the target databases support this feature.

Note: The SQLSTATE codes are defined by the ISO standards. However, not all database types support this standard.

Table 164: SQLSTATE error codes supported per database server type

<table>
<thead>
<tr>
<th>Database Server Type</th>
<th>Supports SQLSTATE errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM® DB2® LUW (UNIX™)</td>
<td>Yes, since version 7.1</td>
</tr>
<tr>
<td>IBM® Informix®</td>
<td>Yes, since IDS 10</td>
</tr>
<tr>
<td>IBM® Netezza®</td>
<td>Yes</td>
</tr>
<tr>
<td>Microsoft™ SQL Server</td>
<td>Yes, since version 8 (2000)</td>
</tr>
<tr>
<td>Oracle® MySQL</td>
<td>Yes</td>
</tr>
<tr>
<td>Oracle® Database Server</td>
<td>Not in version 10.2</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>Yes, since version 7.4</td>
</tr>
<tr>
<td>SAP® ASE</td>
<td>Yes</td>
</tr>
<tr>
<td>SAP HANA®</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Centralize SQL error checking

SQL error identification sometimes requires complex code to check several RDBMS-specific error numbers. Therefore, it is strongly recommended that you centralize SQL error identification in a function. When needed, this allows you to write the RDBMS-specific code only once.

For maximum SQL portability, centralize SQL error checking in functions, to test either SQLCA.SQLCODE or SQLSTATE, depending on the database server type. Furthermore, consider defining error identifiers with constants:

```
CONSTANT SQLERR_INVALID_DATABASE = -1001,
   SQLERR_INVALID_USER = -1002,
   ...

FUNCTION do_connect()
   DEFINE uname, upswd VARCHAR(100)
   WHILE TRUE
      CALL login() RETURNING uname, upswd
      TRY
         CONNECT TO "stores" USER uname USING upswd
      CATCH
         CASE check_sql_error()
            WHEN SQLERR_INVALID_DATABASE
               DISPLAY SQLERRMESSAGE
               EXIT PROGRAM 1 -- Fatal error: Stop!
            WHEN SQLERR_INVALID_USER
               ERROR "Invalid login, try again"
```
SQL error messages

SQLERRMESSAGE is a register that contains the database-specific error message. These messages are different for every database type.

Only use SQLERRMESSAGE to print or log SQL execution diagnostics.

SQL warnings

Some SQL instructions can produce SQL warnings. Unlike SQL errors, SQL warnings indicate a minor issue that can often be ignored. For example, when connecting to an IBM® Informix® database, a warning is returned to indicate that a database connection was opened. Another warning may also be returned if the database supports transactions. None of these facts are critical, but that information may help with program execution.

If an SQL warning is raised, SQLCA.SQLCODE / STATUS remain zero, and the program flow continues. To detect if an SQL warning has occurred, the SQLCA.SQLAWARN register must be checked. SQLCA.SQLAWARN is defined as a CHAR(7) variable. If SQLCA.SQLAWARN[1] contains the W letter, it means that the last SQL instruction returned a warning. The other character positions (SQLCA.SQLAWARN[2-8]) may contain W letters too, depending on the database server type and the type of SQL instruction that was executed.

If SQLCA.SQLAWARN is set, you can also check the SQLSTATE and SQLCA.SQLERRD[2] registers to get more details about the warning. The SQLERRMESSAGE register might also contain the warning description.

In this example, the program connects to a database and displays the content of the SQLCA.SQLAWARN register:

```plaintext
MAIN
   DATABASE stores
   DISPLAY "[", sqlca.sqlawarn, "]"
END MAIN
```

When connecting to an IBM® Informix® database with transactions, the program will display the following:

```
[WW W ]
```

By default, SQL warnings do not stop the program execution. To trap SQL warnings with an exception handler, use the WHENEVER WARNING instruction, as shown in this example:

```plaintext
MAIN
   DEFINE cust_name VARCHAR(50)
   DATABASE stores
   WHENEVER WARNING STOP
      SELECT cust_lname, cust_address INTO cust_name
         FROM customer WHERE cust_id = 101
      WHENEVER WARNING CONTINUE
END MAIN
```

The SELECT statement in the above example uses two columns in the select list, while only one INTO variable is provided. This is legal and does not raise an SQL error. However, SQLCA.SQLAWARN is set to indicate that the number of target variables does not match the select-list items.

See also WHENEVER WARNING exception.
Display detailed debug information in case of internal driver error

If an unexpected problem happens within the database driver, it will return the error -6319 (indicating an internal error in the database driver). When this SQL error occurs, set the FGLSQLDEBUG environment variable to get more details about the internal error.

Determine the number of processed rows

SQL statements such as UPDATE and DELETE can affect zero, one or several rows.

If you need to check how many rows are processed by an SQL statement, check the SQLCA.SQLERRD[3] register.

In the next code example, the SQLCA.SQLCODE and SQLCA.SQLERRD[3] registers are used to diagnose the execution of an UPDATE statement:

```java
TYPE t_country RECORD
  c_code CHAR(5),
  c_desc VARCHAR(50)
END RECORD

FUNCTION update_country_description(rec t_country) RETURNS INTEGER
  UPDATE country SET desc = rec.c_desc WHERE code = rec.c_code
  CASE
    WHEN SQLCA.SQLCODE < 0
      MESSAGE SFMT("SQL error:%1 [%2]", SQLCA.SQLCODE, SQLERRMESSAGE)
      RETURN -2
    WHEN SQLCA.SQLCODE == 0
      IF SQLCA.SQLERRD[3] == 0 THEN
        MESSAGE SFMT("No country was found with code: %1", rec.c_code)
        RETURN -1
      ELSE
        MESSAGE SFMT("Updated country with code: %1", rec.c_code)
      END IF
  END CASE
  RETURN 0
END FUNCTION
```

For more details see The SQLCA diagnostic record on page 594.

Related concepts
SQL error identification on page 595

Identify SQL exceptions in your programs with SQLCA.SQLCODE.

The SQLCA diagnostic record
The SQLCA variable is a predefined record containing SQL statement execution information.

The SQLCA record definition
The SQLCA record is defined as follows:

```java
DEFINE SQLCA RECORD
  SQLCODE INTEGER,
  SQLERRM VARCHAR(71),
  SQLERRP CHAR(7),
  SQLERRD ARRAY[6] OF INTEGER,
  SQLAWARN CHAR(7)
END RECORD
```

1. SQLCODE contains the SQL execution code ( 0 = OK, 100 = not row found, <0 = error ).
2. SQLERRM contains the error message parameter.
3. SQLERRP is not used at this time.
4. SQLERRD[1] is not used at this time.
5. SQLERRD[2] contains the last SERIAL or the native SQL error code.
6. SQLERRD[3] contains the number of rows processed in the last statement (server dependent).
9. SQLERRD[6] contains the ROWID of the last row that was processed (server dependent).
10. SQLWARN contains the ANSI warning represented by a W character at a given position in the string.
11. SQLWARN[1] is set to W when any of the other warning characters have been set to W.
12. SQLWARN[2–7] have specific meanings, see database server documentation for more details.

Usage

The "SQLCA" acronym stands for the SQL Communication Area variable.

The SQLCA record can be used to get an SQL execution diagnostic. Error and warning information can be found in this structure.

The SQLCA record is filled after each SQL statement execution.

SQLCA is not designed to be modified by user code. It must be used as a read-only record.

Portability

SQLCA.SQLCODE will be set to a specific IBM® Informix® SQL error code, provided that the database driver can convert the native SQL error to an IBM® Informix® SQL error. In case of error, SQLCA.SQLERRD[2] will hold the native SQL error produced by the database server.

Other SQLCA record members are specific to IBM® Informix®. For example, after inserting a row in a table with a SERIAL column, SQLCA.SQLERRD[2] will contain the new generated serial number. After an SQL error occurred, SQLCA.SQLERRD[2] will contain the native SQL error. Furthermore, the SQLCA.SQLERRD[3] member may be set with the number of processed rows, as long as the database client supports this feature. Other SQLCA.SQLERRD[n] members must be considered as non portable.

Example

```
MAIN
  WHENEVER ERROR CONTINUE
  DATABASE stores
  SELECT COUNT(*) FROM foo   -- Table should not exist!
  DISPLAY SQLCA.SQLCODE, SQLCA.SQLERRD[2]
END MAIN
```

Related concepts

SQL error identification on page 595
Identify SQL exceptions in your programs with SQLCA.SQLCODE.

SQL execution diagnostics on page 591
If an SQL statement execution fails, error description can be found in the SQLCA.SQLCODE, SQLSTATE, STATUS and SQLERRMESSAGE predefined registers.

SQL errors on PREPARE on page 958
The BIGSERIAL / SERIAL8 data types on page 749

SQL error identification

Identify SQL exceptions in your programs with SQLCA.SQLCODE.

Every database type has its own set of error numbers. Portable SQL code must take care of this when checking for SQL errors in programs.

The IBM® Informix® compatible error code is stored in the SQLCA.SQLCODE register. This aims at simplifying migration to another database type. Existing code based on Informix® error numbers does not need to be modified.
Database drivers map native SQL errors to Informix® SQL errors, as listed in the following table:

**Table 165: Native SQL error to Informix® SQL error mapping**

<table>
<thead>
<tr>
<th>Informix® SQL</th>
<th>Oracle® DB</th>
<th>SQL Server</th>
<th>IBM® DB2</th>
<th>PostgreSQL</th>
<th>Oracle® (SQLSTATE)</th>
<th>SAP® ASE</th>
<th>SAP HANA®</th>
<th>SQLite</th>
<th>IBM® Netezza</th>
</tr>
</thead>
<tbody>
<tr>
<td>-204</td>
<td>N/A 3016</td>
<td>-103</td>
<td>N/A N/A N/A N/A N/A N/A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-206</td>
<td>903, 942</td>
<td>3701, 4004</td>
<td>-204</td>
<td>42P01 1146, 1051</td>
<td>207, 208, -259</td>
<td>N/A</td>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-217</td>
<td>904</td>
<td>4005</td>
<td>-205, -206</td>
<td>42703 1054</td>
<td>N/A -260 N/A 31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-236</td>
<td>913, 947 1200</td>
<td>-117</td>
<td>N/A N/A N/A N/A N/A N/A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-244</td>
<td>N/A 1222</td>
<td>N/A N/A N/A 12205</td>
<td>-131</td>
<td>N/A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-251</td>
<td>N/A N/A</td>
<td>-125</td>
<td>N/A N/A N/A N/A N/A N/A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-253</td>
<td>972</td>
<td>2014</td>
<td>-107</td>
<td>N/A N/A N/A N/A N/A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-254</td>
<td>1008, 1475</td>
<td>N/A N/A</td>
<td>N/A N/A N/A N/A -270 N/A N/A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-255</td>
<td>N/A N/A N/A</td>
<td>25P01 N/A N/A N/A N/A N/A N/A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-257</td>
<td>1000 N/A N/A N/A N/A N/A N/A N/A N/A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-263</td>
<td>54 N/A N/A</td>
<td>N/A N/A N/A N/A N/A N/A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-268</td>
<td>1 2601, 2627</td>
<td>-803</td>
<td>23505 1062</td>
<td>2601 -301 N/A N/A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-280</td>
<td>N/A N/A</td>
<td>-102</td>
<td>N/A N/A N/A N/A -274</td>
<td>N/A N/A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-282</td>
<td>N/A N/A</td>
<td>-105</td>
<td>N/A N/A N/A N/A N/A N/A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-294</td>
<td>937 N/A</td>
<td>-119, -122</td>
<td>N/A N/A N/A N/A -277 N/A N/A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-316</td>
<td>N/A N/A</td>
<td>-605</td>
<td>N/A N/A N/A N/A -289 N/A N/A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-324</td>
<td>960 N/A</td>
<td>-203</td>
<td>N/A N/A N/A N/A N/A N/A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-350</td>
<td>1408 N/A N/A</td>
<td>N/A N/A N/A N/A N/A N/A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-360</td>
<td>N/A N/A</td>
<td>-118</td>
<td>N/A N/A N/A N/A N/A N/A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Informix® Oracle®</td>
<td>SQL DB2®</td>
<td>PostgreSQL® (SQLSTATE)</td>
<td>Oracle® MySQL®</td>
<td>SAP® ASE</td>
<td>SAP HANA®</td>
<td>SQLite</td>
<td>IBM® Netezza</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------------------</td>
<td>----------</td>
<td>------------------------</td>
<td>-----------------</td>
<td>----------</td>
<td>-----------</td>
<td>--------</td>
<td>-------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-371</td>
<td>1452</td>
<td>N/A</td>
<td>-603, -673</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-382</td>
<td>1756</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-387</td>
<td>1017, 1045</td>
<td>715, 4002, 4003, 4008</td>
<td>-1403, -1404</td>
<td>N/A</td>
<td>1045</td>
<td>N/A</td>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-388</td>
<td>1536</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-391</td>
<td>1400, 1407</td>
<td>N/A</td>
<td>-407</td>
<td>22004, 23502</td>
<td>N/A</td>
<td>N/A</td>
<td>-287</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-400</td>
<td>1002</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-517</td>
<td>N/A</td>
<td>N/A</td>
<td>-602</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-530</td>
<td>2290</td>
<td>N/A</td>
<td>-193</td>
<td>23514</td>
<td>N/A</td>
<td>548</td>
<td>-677</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-551</td>
<td>N/A</td>
<td>N/A</td>
<td>-613</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-674</td>
<td>N/A</td>
<td>2812</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>14216</td>
<td>-328</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-681</td>
<td>N/A</td>
<td>2812</td>
<td>-121</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>-272</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-691</td>
<td>2291</td>
<td>547</td>
<td>-530</td>
<td>23503</td>
<td>1452</td>
<td>546</td>
<td>-461</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-743</td>
<td>955</td>
<td>6000, 6006, 6008</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>2714</td>
<td>-288</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-930</td>
<td>1033, 1034, 12154, 12203, 12224, 12500, 12560</td>
<td>11, 17, 708, 709, 711, 4014, 17142</td>
<td>-1013</td>
<td>08000, 08001, 08004, 08006, 08007, 08000</td>
<td>1044</td>
<td>4002</td>
<td>-10, -10709</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-942</td>
<td>N/A</td>
<td>N/A</td>
<td>-903</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1202</td>
<td>N/A</td>
<td>N/A</td>
<td>-801</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>-304</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1218</td>
<td>N/A</td>
<td>3048, 3049, 3050</td>
<td>-180, -181</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1260</td>
<td>932</td>
<td>N/A</td>
<td>-190</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>-339</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1279</td>
<td>1401</td>
<td>N/A</td>
<td>-433, -99998</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1349</td>
<td>1722</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sometimes the native error code of the database cannot be converted to an Informix® error code. In such case, the SQLCA.SQLCODE register will be set to -6372. To properly identify an SQL error, the native SQL error code is also provided in the SQLCA.SQLERRD[2] register.

Centralize SQL error identification in a function:

```
-- sqlerr.4gl module
```
public constants:
- SQLERRTYPE_FATAL = -1
- SQLERRTYPE_LOCK = -2
- SQLERRTYPE_CONN = -3
- SQLERRTYPE_UNDEF = -999

FUNCTION lastSqlErrorType()
    CASE
    WHEN SQLCA.SQLCODE == -201
        OR SQLCA.SQLErrorD[2] == ...
        RETURN SQLERRTYPE_FATAL
    WHEN SQLCA.SQLCODE == -263
        OR SQLCA.SQLCODE == -244
        OR SQLCA.SQLErrorD[2] == ...
        RETURN SQLERRTYPE_LOCK
    OTHERWISE
        RETURN SQLERRTYPE_UNDEF
    END CASE
END FUNCTION

Then you can then easily use this function after every SQL statement:

import FGL sqlerr
main
    database stores
    whenever error continue
    update customer set cust_address = null
    when cust_name is null
        if lastSqlErrorType() == SQLERRTYPE_LOCK then ...
        end if ...
    end if ...
end main

Related concepts
SQL execution diagnostics on page 591
If an SQL statement execution fails, error description can be found in the SQLCA.SQLCODE, SQLSTATE, STATUS
and SQLERRMESSAGE predefined registers.

Debugging SQL statements on page 603
The runtime system can display debug information for SQL statements executed by the program.

Cursors and connections
How to use database cursors across connections?

Several database connections can be opened simultaneously with the CONNECT TO instruction. Once connected, you can DECLARE cursors or PREPARE statements to be used in parallel within different connection contexts. This section describes how to use SQL cursors and SQL statements in a multiple-connection program.

When you DECLARE a cursor or when you PREPARE a statement, you actually create an SQL statement handle; the runtime system allocates resources for that statement handle before sending the SQL text to the database server via the database driver.

The SQL statement handle is created in the context of the current connection, and must be used in that context, until it is freed or recreated with another DECLARE or PREPARE statement. Using an SQL statement handle in a different connection context than the one for which it was created will produce a runtime error.

The SET CONNECTION instruction changes the connection context. Connections are identified by a name. The AS clause of the CONNECT TO instruction allows you to specify a connection name. If the AS clause is omitted, the connection gets a default name based on the data source name.
This small program example illustrates the use of two cursors with two different connections:

```
MAIN
  CONNECT TO "db1" AS "s1"
  CONNECT TO "db2" AS "s2"
  SET CONNECTION "s1"
  DECLARE c1 CURSOR FOR SELECT tab1.* FROM tab1
  SET CONNECTION "s2"
  DECLARE c2 CURSOR FOR SELECT tab1.* FROM tab1
  SET CONNECTION "s1"
  OPEN c1
  SET CONNECTION "s2"
  OPEN c2
  ...
END MAIN
```

The `DECLARE` and `PREPARE` instructions are a type of creator instruction; if an SQL statement handle is recreated in a connection other than the original connection for which it was created, old resources are freed and new resources are allocated in the current connection context.

This allows you to re-execute the same cursor code in different connection contexts, as in this example:

```
MAIN
  CONNECT TO "db1" AS "s1"
  CONNECT TO "db2" AS "s2"
  SET CONNECTION "s1"
  IF checkForOrders() > 0 ...
  SET CONNECTION "s2"
  IF checkForOrders() > 0 ...
  ...
END MAIN

FUNCTION checkForOrders(d)
  DEFINE d DATE, i INTEGER
  DECLARE c1 CURSOR FOR SELECT COUNT(*) FROM orders WHERE ord_date = d
  OPEN c1
  FETCH c1 INTO i
  CLOSE c1
  FREE c1
  RETURN i
END FUNCTION
```

If the SQL statement handle was created in a different connection, the resources used in the old connection context are freed automatically, and new statement handle resources are allocated in the current connection context.

Related concepts

- [Database connections](#) on page 656
  Explains how to manage database connections in a program.

Concurrent data access

Understanding concurrent data access and data consistency.

*Data concurrency* is the simultaneous access of the same data by many users. On the other hand, *data consistency* means that each user sees a consistent view of the database. Without adequate concurrency and consistency controls, data may be changed improperly, compromising data integrity. To write inter-operable applications, you must adapt the program logic to the behavior of the database server regarding concurrency and consistency management. This issue requires good knowledge of multiuser application programming, transactions, locking mechanisms, isolation levels and wait mode. If you are not familiar with these concepts, carefully read the documentation of each database server which covers this subject.

Processes accessing the database can change transaction parameters such as the isolation level. Existing programs might have to be adapted in order to work with this new behavior.
The following is the best configuration to get common behavior with all types of database engines:

- The database must support transactions; this is usually the case.
- Transactions must be as short as possible (under a second is fine, 3 or more seconds is a long transaction).
- The isolation level should be set to COMMITTED READ or CURSOR STABILITY. IBM® Informix® IDS 11 has introduced the LAST COMMITTED option for the COMMITTED READ isolation level, which makes IDS behave like other database servers using row-versioning, by returning the most recently committed version of the row, rather than wait for a lock to be released. This option can also be turned on implicitly with the USELASTCOMMITTED configuration parameter, saving code changes.
- The wait mode for locks must be WAIT or WAIT n (timeout). Wait mode can be adapted to wait for the longest transaction.

Remarks: With this configuration, the locking granularity does not have to be at the row level. To improve performance with IBM® Informix®, you can use the LOCK MODE PAGE locking level, which is the default. However, if the application requires row-level locking, you must use LOCK MODE ROW clause, or define the DEF_TABLES_LOCKMODE configuration parameter to ROW, in the onconfig file of the server.

Table 166: Concurrent data access differences in database engine brands

<table>
<thead>
<tr>
<th>Database Server Type</th>
<th>Concurrency topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM® DB2® LUW (UNIX™)</td>
<td>Concurrency in IBM DB2</td>
</tr>
<tr>
<td>IBM® Informix®</td>
<td>N/A</td>
</tr>
<tr>
<td>IBM® Netezza®</td>
<td>Concurrency in Netezza</td>
</tr>
<tr>
<td>Microsoft™ SQL Server</td>
<td>Concurrency in SQL Server</td>
</tr>
<tr>
<td>Oracle® MySQL</td>
<td>Concurrency in Oracle MySQL</td>
</tr>
<tr>
<td>Oracle® Database Server</td>
<td>Concurrency in Oracle DB</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>Concurrency in PostgreSQL</td>
</tr>
<tr>
<td>SAP® ASE</td>
<td>Concurrency in SAP ASE</td>
</tr>
<tr>
<td>SAP HANA®</td>
<td>Concurrency in SAP HANA</td>
</tr>
<tr>
<td>SQLite</td>
<td>Concurrency in SQLite</td>
</tr>
</tbody>
</table>

Related concepts

Database transactions on page 688
Database transaction concepts and handling.

Database transactions

Database transactions define a set of SQL instructions to be executed as a whole, or rolled back as a whole.

The BDL instructions to define a transaction block are:

- BEGIN WORK
- COMMIT WORK
- ROLLBACK WORK

Additional BDL instructions related to transaction management is available, such as SET LOCK MODE . . . , SET ISOLATION . . . , SAVEPOINT, etc. For a complete description, see Database transactions on page 688

When performing a transaction instruction, the database drivers execute the corresponding native SQL instruction (or database client API call) to begin, commit or rollback a transaction.

In this example a basic transaction block executes inside a TRY/CATCH block to rollback the transaction in case of SQL error:
CONNECT TO ...
TRY
  BEGIN WORK
  UPDATE tab1 SET col1 = 'aaa' WHERE pkey = 123
  UPDATE tab2 SET col2 = 'bbb' WHERE pkey = 456
  ...
  COMMIT WORK
CATCH
  ROLLBACK WORK
END TRY
END MAIN

Related concepts

Concurrent data access on page 599
Understanding concurrent data access and data consistency.

Transaction blocks across connections on page 612
Transaction blocks manage transactions when connected to several database servers.

Handling nested transactions on page 611
You can manage nested transactions in different parts of a program.

Avoiding long transactions on page 652
Long transactions consume resources and decrease concurrent data access.

Performance with transactions on page 652
Commit database changes by blocks of transaction speeds performance with some database servers.

Using SQL interruption
Interrupt long running SQL queries, or interrupt queries waiting for locked data.

If the database server supports SQL interruption, a program can interrupt a long running SQL statement.

Important: Not all database servers support SQL interruption.

SQL interruption is not enabled by default. Use the OPTIONS SQL INTERRUPT ON instruction to enable SQL interruption.

The OPTIONS SQL INTERRUPT ON instruction must be used together with signal handling instructions DEFER INTERRUPT and DEFER QUIT, otherwise the program will stop immediately in case of an interruption event.

Note: Depending on the type of database server and client, SQL interruption handling may require additional overhead. Consider enabling SQL interruption only for SQL queries that can take a while to execute or can be blocked for a long time because of concurrent access (locks).

When the program receives an interruption event (either a SIGINT signal from the system, or an interrupt event from the front-end) and SQL interrupt is enabled with OPTIONS SQL INTERRUPT ON, the following happens:

- The running SQL statement is stopped,
- The INT_FLAG global variable is set to TRUE,
- The SQLCA.SQLCODE is set with error -213.

SQL interruption results in abnormal SQL statement execution and raises a runtime error. Therefore, the SQL statement that can be interrupted must be protected by a WHENEVER ERROR exception handler or TRY/CATCH block.

MAIN
  DEFINE cnt INTEGER
  DEFER INTERRUPT
  DATABASE test1
  WHENEVER ERROR CONTINUE
  OPTIONS SQL INTERRUPT ON
  -- Start long query (self join takes time)
  -- From now on, user can hit CTRL-C in TUI mode to stop the query,
-- or use the special "interrupt" action (button) in GUI mode.

```
SELECT COUNT(*) INTO cnt FROM customers a, customers b
  WHERE a.cust_id <> b.cust_id
OPTIONS SQL INTERRUPT OFF
IF SQLCA.SQLCODE == -213 THEN
  DISPLAY "Statement was interrupted by user..."
  EXIT PROGRAM 1
END IF
WHENEVER ERROR STOP
END MAIN
```

When SQL interruption is supported by a database server type other than IBM® Informix®, the database drivers will return error -213 in case of interruption, to behave as in IBM® Informix®.

**Note:** When FGLSQLDEBUG is set, OPTIONS SQL INTERRUPT ON/OFF instructions are traced.

### Table 167: Database server support of SQL interruption

<table>
<thead>
<tr>
<th>Database Server Type</th>
<th>SQL Interruption API</th>
<th>SQL error code for interrupted query</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM® DB2® LUW (Since version 9.x)</td>
<td>SQLCancel()</td>
<td>Native error -952</td>
</tr>
<tr>
<td>IBM® Informix®</td>
<td>sqlbreak()</td>
<td>Native error -213</td>
</tr>
<tr>
<td>IBM® Netezza®</td>
<td>SQLCancel()</td>
<td>Native error 46</td>
</tr>
<tr>
<td>Microsoft™ SQL Server (Only 2005+ with SNC driver)</td>
<td>SQLCancel()</td>
<td>SQLSTATE HY008</td>
</tr>
<tr>
<td>Oracle® MySQL</td>
<td>KILL QUERY</td>
<td>Native error -1317</td>
</tr>
<tr>
<td>Oracle® Database Server</td>
<td>OCIbreak()</td>
<td>Native error -1013</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>PQCancel()</td>
<td>SQLSTATE 57014</td>
</tr>
<tr>
<td>SAP® ASE</td>
<td>ct_cancel()</td>
<td>SQLSTATE HY008</td>
</tr>
<tr>
<td>SAP HANA®</td>
<td>SQLCancel()</td>
<td>Native error -139</td>
</tr>
<tr>
<td>SQLite</td>
<td>sqlite3_interrupt()</td>
<td>Native error SQLITE_ABORT</td>
</tr>
</tbody>
</table>

### Related concepts

- **Concurrent data access** on page 599
  Understanding concurrent data access and data consistency.

- **The SQLCA diagnostic record** on page 594
  The SQLCA variable is a predefined record containing SQL statement execution information.

- **TRY - CATCH block** on page 506
  Use TRY / CATCH blocks to trap runtime exceptions in a delimited code block.

- **WHENEVER instruction** on page 503
  Use the WHENEVER instruction to define how exceptions must be handled for the rest of the module.

### Implicit database connection

An implicit database connection is made with the DATABASE instruction used before MAIN; use SCHEMA to avoid the implicit connection.

The DATABASE instruction can be used in two distinct ways, depending on the context of the statement within its source module:

- To specify a default database.
Typically used in a GLOBALS module, to define variables with the DEFINE ... LIKE, but it is also used for the INITIALIZE and VALIDATE statements. Using the DATABASE statement in this way results in that database being opened automatically at run time.

- To specify a current database.

  In MAIN or in a FUNCTION, used to connect to a database. A variable can be used in this context (DATABASE varname).

A default database is almost always used, because many programs contain DEFINE ... LIKE statements. A problem occurs when the production database name differs from the development database name, because the default database specification will result in an automatic connection (just after MAIN):

```sql
DATABASE stock_dev  -- Default database, used at compile time
DEFINE
  p_cust RECORD LIKE customer.*
MAIN -- Connection to default database occurs at MAIN
  DEFINE dbname CHAR(30)
  LET dbname = "stock1"
  DATABASE dbname -- Real database used in production
...  
END MAIN
```

In order to avoid the implicit connection, you can use the SCHEMA instruction instead of DATABASE:

```sql
SCHEMA stock_dev -- Schema specification only
DEFINE
  p_cust RECORD LIKE customer.*
MAIN -- No default connection occurs...
  DEFINE dbname CHAR(30)
  LET dbname = "stock1"
  DATABASE dbname
END MAIN
```

This instruction will define the database schema for compilation only, and will not make an implicit connection at runtime.

**Related concepts**

Database schema on page 467

Defines database table structures with column type information to be reused in program variable definitions.

**Debugging SQL statements**

The runtime system can display debug information for SQL statements executed by the program.

SQL debug information is printed by the runtime system, when using a value different from zero in the FGLSQLDEBUG environment variable or with the fgl_sqldebug() built-in function.

The debug messages are sent to the standard error stream (stderr). Consider to redirect the standard error output into a file, to inspect the debug log after program execution.

UNIX™ (shell) example:

```bash
FGLSQLDEBUG=3
export FGLSQLDEBUG
fglrun myprog 2>sqldbg.txt
```

When the debug level is set with a positive value, the runtime system prints an SQL debug message for each SQL statement executed by the program.

When the debug level is set to -1, the runtime system only prints debug information for SQL statements that produce an error.
SQL debug messages show detailed information about the SQL statement execution, for example:

```
SQL: DATABASE
   4gl source : c.4gl line=2
   loading driver : [/opt/fgl/dbdrivers/dbmdefault]
   db driver type : ifx
   sqlcode : 0
   curr driver : ident='dbmdefault'
   curr connection : ident='_1' (dbspec=[test1])
   Execution time : 0 00:00:00.02689

SQL: DELETE FROM mytable WHERE pkey IS NULL
   4gl source : c.4gl line=4
   sqlcode : -206
   sqlstate : 42000
   sqlerrd2 : -111
   sql message : The specified table (mytable) is not in the database.
   sql msg param : mytable
   curr driver : ident='dbmdefault'
   curr connection : ident='_1' (dbspec=[test1])
   Execution time : 0 00:00:00.00035
```

**Note:** The most important information is the SQL error code and the source code line where the SQL statement failed. For performance tuning, the execution time can be used to identify slow SQL statements.

If the SQL debug log generates too many messages, define the debug level to -1, in order to produce an SQL debug message only for the SQL statements that produce an SQL error. Once the SQL statement producing the error is identified, you can find the source code line where the error occurred, to reproduce the problem. Before reproducing the error, set the debug level to a high positive level, to get detailed debug information.

**Note:** When the debug level is set to a positive value, an SQL debug header is printed before executing the underlying database driver code. If the driver code crashes or raises an assertion, you can easily find the last SQL instruction that was executed by the program, and report to your support center. When the debug level is set to -1, the SQL debug header is not printed before executing the database driver code, because the SQL execution status (SQLCA.SQLCODE) is not known before executing the statement. If the driver code crashes, no error message will be printed. If you experience database driver crashes, use a positive debug level in order to identify the problem.

It is also possible to use the `fgl_sqldebug()` built-in function to set the SQL debug level by program:

```
MAIN
   DATABASE stores
   DELETE FROM mytable   -- SQL debug output if FGLSQLDEBUG is set
   CALL fgl_sqldebug(3)
   DELETE FROM mytable   -- SQL debug output is enabled by program
   CALL fgl_sqldebug(0)
   DELETE FROM mytable   -- SQL debug output if FGLSQLDEBUG is set
   CALL fgl_sqldebug(-1)
   DELETE FROM undefined -- SQL debug output only in case of error
END MAIN
```

**Related concepts**

- The SQLCA diagnostic record on page 594
- The SQLCA variable is a predefined record containing SQL statement execution information.
- SQL execution diagnostics on page 591
- If an SQL statement execution fails, error description can be found in the SQLCA.SQLCODE, SQLSTATE, STATUS and SQLERRMESSAGE predefined registers.
- The database utility library
- The `fgldbutl.4gl` library provides several database-related utility functions.

You find this library in the `$FGLDIR/src` directory.
The DB utility library implements helpers for the following areas:

- Database type identification
- Sequence number generation
- Nested transaction control

See fgldbutl: Database utility module on page 2307 for more details.

**SQL security**

It is important to consider security aspects when programming with SQL.

**Authenticating DB users**

Understanding how users are authenticated to the database server.

When connecting to a database server, the user must be identified by the server. Once connected, the current user is authenticated and identified by the db server, and the database system can then apply specific privileges, audit user activity, and so on.

Database user authentication is typically achieved by specifying a login and password in the `CONNECT TO` instruction. However, most database servers support additional user authentication methods, such as OS user authentication, trusted connections, LDAP authentication, Single Sign-On authentication, and even specific pluggable authentication methods.

Follow these simple security patterns to avoid basic user authentication problems:

- Make sure that application files installed on your production server have the appropriate file system permissions set. For regular users it is recommended that they have read-only access to program and resource files. If any OS user can replace a program file with another program, it could harm your database or retrieve sensitive private data.
- Each physical end user must have a specific database account. If several end users connect as the same db application account, they cannot be distinguished in the security and auditing system.
- For normal application users, always use database accounts with the minimum database privileges required to achieve the daily work (GRANT/REVOKE). For example, regular users are not given permission to execute Data Definition Language statements (drop tables).
- Instead of asking a name and password in a login dialog when an application starts, some applications hard code the db user names and passwords in the program code, in scripts or configurations files such as FGLPROFILE. This is not a good practice and must be avoided. If a login dialog is not appropriate, you must set up another user authentication method supported by the database server, such as Single Sign-On.

**Related concepts**

- Database user authentication on page 679

Different database user authentication methods exist.

- The FGLPROFILE file(s) on page 255

FGLPROFILE environment variable defines Genero BDL configuration files

**Avoiding SQL injection**

Prevent SQL injection attacks in your programs.

SQL injection is a well-known attack that started to appear with Web applications, where the end user enters SQL statement fragments in form fields that are normally designed to hold simple data. When the entered text is used to complete an SQL statement without further checking, there is a risk of SQL statements being injected by the user to intentionally harm the database.

To illustrate the problem, see the following code:

```plaintext
MAIN
    DEFINE sql CHAR(200), cn CHAR(50), n INTEGER
    OPEN FORM f FROM "custform"
    DISPLAY FORM f
    INPUT BY NAME cn
```
LET sql = "SELECT COUNT(*) FROM customers WHERE custname = ", cn, ",""
PREPARE stmt FROM sql
EXECUTE stmt INTO n
DISPLAY "Count = ", n
END MAIN

If the end user enters for example:

[xxx' ; delete from customers ]

The resulting SQL statement will contain an additional DELETE command that will drop all rows of the customers table:

SELECT COUNT(*) FROM customers WHERE custname = 'xxx'; DELETE FROM customers

In some applications, you may also want to let the end user choose sort columns to be added in an ORDER BY clause. The recommendation is that code for such a feature controls the user input. For example, by providing a list of columns to choose from, instead of allowing free text input that will be added to the ORDER BY clause.

To avoid SQL injection attacks, do not build SQL instructions dynamically by concatenating user input that is not checked. Instead of basic concatenation, use static SQL statements with program variables (if dynamic SQL is not needed), use parameterized queries (with ? parameter placeholders), or use the CONSTRUCT instruction to implement a query by example form.

Simple static SQL example:

MAIN
  DEFINE cn CHAR(50), n INTEGER
  OPEN FORM f FROM "custform"
  DISPLAY FORM f
  INPUT BY NAME cn
  SELECT COUNT(*) INTO n FROM customers WHERE custname = cn
  DISPLAY "Count = ", n
END MAIN

Parameterized query example:

MAIN
  DEFINE sql CHAR(200), cn CHAR(50), n INTEGER
  OPEN FORM f FROM "custform"
  DISPLAY FORM f
  INPUT BY NAME cn
  LET sql = "SELECT COUNT(*) FROM customers WHERE custname = ?"
  PREPARE stmt FROM sql
  EXECUTE stmt USING cn INTO n
  DISPLAY "Count = ", n
END MAIN

CONSTRUCT example:

MAIN
  DEFINE sql CHAR(200), cond CHAR(50), n INTEGER
  OPEN FORM f FROM "custform"
  DISPLAY FORM f
  CONSTRUCT BY NAME cond ON custname
  LET sql = "SELECT COUNT(*) FROM customers WHERE ", cond
  PREPARE stmt FROM sql
  EXECUTE stmt INTO n
  DISPLAY "Count = ", n
END MAIN
Related concepts
Query by example (CONSTRUCT) on page 1565
The CONSTRUCT instruction implements database query criteria input in an application form.
PREPARE (SQL statement) on page 710
Prepares an SQL statement for execution.

SQL portability
Writing portable SQL is mandatory, to support different kind of database servers.
This section provides hints to solve SQL incompatibility problems in your programs.
In addition to this SQL portability guide, read carefully the database-specific guides which contain database specific information about SQL compatibility issues.
To easily detect SQL statements with specific syntax, you can use the \texttt{-W stdsql} option of fglcomp:

\begin{quote}
\verb|$ fglcomp -W stdsql orders.4gl module.4gl:15: SQL Statement or language instruction with specific SQL syntax.$
\end{quote}

This compiler option can only detect non-portable SQL syntax in static SQL statements.

Database entities
The database entity concept across different database engines.
Most database servers can handle multiple database entities (you can create multiple ‘databases’), but this is not possible with all engines:

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
Database Server Type & Multiple Database support \\
\hline
IBM® DB2® LUW & Yes, see details \\
IBM® Informix® & Yes, see details \\
IBM® Netezza® & Yes, see details \\
Microsoft™ SQL Server & Yes, see details \\
Oracle® MySQL / Mariadb & Yes, see details \\
Oracle® Database Server & Yes, see details \\
PostgreSQL & Yes, see details \\
SAP® ASE & Yes, see details \\
SAP HANA® & Yes, see details \\
SQLite & Yes, see details \\
\hline
\end{tabular}
\end{table}

When using a database server that does not support multiple database entities, you can emulate different databases with schema entities, but this requires you to check for the database user definition. Each database user must have privileges to access any schema, and to see any table of any schema without needing to set a schema prefix before table names in SQL statements.

Some database drivers allow to select a specific schema at connection with the following FGLPROFILE entry:

\begin{quote}
\verb|dbi.database.dbname.dbtype.schema = "schema-name"|
\end{quote}

Some databases also allow you to define a default schema for each database user. When the user connects to the database, the default schema is automatically selected.
**Related concepts**
The FGLPROFILE file(s) on page 255
FGLPROFILE environment variable defines Genero BDL configuration files

**Database users and security**
Properly identifying database users allows to use database security and audit features.

To get the benefit of the database server security features, you should identify each physical user as a database user.

Some applications use a single database user for different end users, to avoid user management and connection issues in the database. This is not good practice because all user-related features of the database are unusable. Furthermore, the single db user often has all database privileges and thus can lead in security issues.

According to the type of server, you must do this steps to create a database user:

1. Define the user as an operating system user.
2. Declare the user in the database server.
3. Grant database access privileges.

Each database server has its specific users management and data access privilege mechanisms. Check the vendor documentation for security features and make sure you can define the users, groups, and privileges in all database servers you want to use.

**Table 169: Database user definitions by database engine brands**

<table>
<thead>
<tr>
<th>Database Server Type</th>
<th>DB Users topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM® DB2® LUW (UNIX™)</td>
<td>DB Users in IBM DB2 LUW</td>
</tr>
<tr>
<td>IBM® Informix®</td>
<td>DB Users in IBM Informix</td>
</tr>
<tr>
<td>IBM® Netezza®</td>
<td>DB Users in IBM Netezza</td>
</tr>
<tr>
<td>Microsoft™ SQL Server</td>
<td>DB Users in SQL Server</td>
</tr>
<tr>
<td>Oracle® MySQL / Mariadb</td>
<td>DB Users in Oracle MySQL</td>
</tr>
<tr>
<td>Oracle® Database Server</td>
<td>DB Users in Oracle DB</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>DB Users in PostgreSQL</td>
</tr>
<tr>
<td>SAP® ASE</td>
<td>DB Users in SAP ASE</td>
</tr>
<tr>
<td>SAP HANA®</td>
<td>DB Users in SAP HANA</td>
</tr>
<tr>
<td>SQLite</td>
<td>DB Users in SQLite</td>
</tr>
</tbody>
</table>

**Related concepts**
Database user authentication on page 679
Different database user authentication methods exist.

**Creating a database from programs**
Creating a database from within a program requires special consideration.

**Understanding database creation statements**
The Genero language syntax supports database creation statements such as:

```
CREATE DATABASE mydb WITH BUFFERED LOG
```

Such instruction performs an implicit connection to the database server (this means that no CONNECT TO or DATABASE is required before a CREATE DATABASE), which leads to a default connection.
Creating a database in a database server

When using a database server engine, the creation of a database entity is not a trivial operation. The process usually requires additional tasks such as data storage configuration, database user creation, data access policy, and so on. These tasks are typically left to the database administrator.

Database creation statements such as CREATE DATABASE, CREATE DBSPACE, and DROP DATABASE can be used in programs connected to an IBM® Informix® server, but these statements are not portable. Use database creation statements only for development or testing purpose.

Creating a database on mobile devices (SQLite)

Mobile applications usually create their database at first execution. Database creation on a mobile device is a much simpler operation than database creation on a database server. For example, with SQLite, creating a database only requires creating an empty file.

The SQLite database file must be created in the application sandbox, in a writable directory. This directory is specific to the type of mobile device, and can be found in programs with the os.Path.pwd on page 2806 method.

To build the full path to the database file, get the current working directory (os.Path.pwd()) and add this path to the database file name. This defines the source specification in the database connection parameters, to build the string used for the CONNECT instruction.

Note: Backslashes (\) in the parameters of the connection string must be escaped, before using the connection string in a CONNECT TO or DATABASE instruction. In the next code example, when executing on a Windows® platform, the file path to the SQLite database file will contain backslashes, that need to be doubled. Note that a backslash in a string literal of the BDL source is also doubled: "\" represents in fact a single backslash.

```plaintext
IMPORT os
...
DEFINE dbfile, source, connstr VARCHAR(256)

FUNCTION init_connection_strings()
    LET dbfile = "contacts.dbs"
    LET source = os.Path.join(os.Path.pwd(), dbfile)
    LET connstr = SFMT("contacts+source='%1', escape_backslashes(source) )
    IF NOT base.Application.isMobile() THEN
        -- Add db driver spec when in development mode
        LET connstr = connstr, ",driver='dbmsqt'"
    END IF
END FUNCTION

FUNCTION escape_backslashes(str)
    DEFINE str STRING
    DEFINE buf base.StringBuffer
    LET buf = base.StringBuffer.create()
    CALL buf.append(str)
    CALL buf.replace("\\","\\\",0)
    RETURN buf.toString()
END FUNCTION
```

If not specified, the source connection parameter (i.e., the path to the database file) defaults to the database name specification in the CONNECT instruction. Thus, the source='dbpath' parameter is usually omitted, and dbpath is specified directly as the database name. In this case, however, the identifier of the database connection is the complete path to the SQLite database file. For more details about database connection parameters, see Database connections on page 656.

Before executing the CONNECT instruction, check if the database file already exists with os.Path.exists(source). Create the database file and tables only if needed:

```plaintext
IMPORT os
...
```
CALL init_connection_strings()
IF os.Path.exists(source) THEN
    CONNECT TO connstr AS "c1"
ELSE
    CALL create_empty_file(source)
    CONNECT TO connstr AS "c1"
    CALL create_database_tables()
END IF
...

FUNCTION create_empty_file(fn)
DEFINE fn STRING
DEFINE ch base.Channel
LET ch = base.Channel.create()
CALL ch.openFile(fn,"w")
CALL ch.close()
END FUNCTION

Instead of creating an empty database file, it is also possible to prepare a template (pre-configured) SQLite database file on the development platform, deploy the template database with the other program files, and copy the template file from the program files directory (base.Application.getProgramDir on page 2408) into the working directory (os.Path.pwd on page 2806) on the first application execution (that is when the database file in the working directory does not yet exist):

IMPORT os
...
    CALL init_connection_strings()
    IF NOT prepare_database("template.dbs", source) THEN
        ERROR "Could not prepare database"
        EXIT PROGRAM 1
    END IF
    CONNECT TO connstr AS "c1"
    ...
...

FUNCTION prepare_database(template, target)
DEFINE template, target STRING
DEFINE tplpath STRING
IF os.Path.exists(target) THEN
    RETURN TRUE
END IF
LET tplpath = os.Path.join(base.Application.getProgramDir(), template)
IF NOT os.Path.exists(tplpath) THEN
    ERROR "Database template file not found"
    RETURN FALSE
END IF
RETURN os.Path.copy(tplpath, target)
END FUNCTION

Important: When creating an initial database file in the working directory from a template file deployed in the program files directory, different file names should be used for the template and actual database file, as folders pointed to by base.Application.getProgramDir on page 2408 and os.Path.pwd on page 2806 may be the same on some devices.

Related concepts
Directory structure for GMA apps on page 3589
Platform-specific rules need to be considered when deploying on Android™ devices (GMA).
Directory structure for GMI apps on page 3604
Platform-specific rules need to be considered when deploying on iOS devices (GMI).

**Handling nested transactions**
You can manage nested transactions in different parts of a program.

A program can become very complex if it contains a lot of nested functions calls, doing SQL processing within transactions. You may want to centralize transaction control commands in wrapper functions. The fgldbutl.4gl library contains special functions to manage the beginning and the end of a transaction with an internal counter, in order to implement nested function calls inside a unique transaction.

```plaintext
MAIN
  IF a() <> 0 THEN
    ERROR "..."
  END IF
  IF b() <> 0 THEN
    ERROR "..."
  END IF
END MAIN

FUNCTION a()
  DEFINE s INTEGER
  LET s = db_start_transaction()
  UPDATE ...
  LET s = SQLCA.SQLCODE
  IF s = 0 THEN
    LET s = b()
  END IF
  LET s = db_finish_transaction((s==0))
  RETURN s
END FUNCTION

FUNCTION b()
  DEFINE s INTEGER
  LET s = db_start_transaction()
  UPDATE ...
  LET s = SQLCA.SQLCODE
  LET s = db_finish_transaction((s==0))
  RETURN s
END FUNCTION
```

In this example, you see in the **MAIN** block that both functions **a()** and **b()** can be called separately. However, the transaction SQL commands will be used only if needed: When function **a()** is called, it starts the transaction, then calls **b()**, which does not start the transaction since it was already started by **a()**. When function **b()** is called directly, it starts the transaction.

The function **db_finish_transaction()** is called with the expression **(s==0)**, which is evaluated before the call. This allows you to write in one line the equivalent of the following **IF** statement:

```plaintext
IF s==0 THEN
  LET s = db_finish_transaction(1)
ELSE
  LET s = db_finish_transaction(0)
END IF
```

**Related concepts**
The **MAIN block / function** on page 464
The **MAIN block** is the starting point of the program.

**Database transactions** on page 688
Database transaction concepts and handling.

**Transaction blocks across connections**
Transaction blocks manage transactions when connected to several database servers.

In some cases, you need to copy data from a database to another. Database vendor export / import tools exist for this task and their use is preferred when large data transfer is needed. However, it is also possible to achieve this with a BDL program connected to both databases, reading data from the source database and inserting rows into the target database.

If the rows created in the target database need to be committed as a whole, you must open a transaction with the `BEGIN WORK` instruction, use `SET CONNECTION` to switch between the connections to read/write rows, and terminate the transaction with a `COMMIT WORK`.

In order to keep a transaction open when switching to another database connection, the connection must be initiated with the `WITH CONCURRENT TRANSACTION` clause. If this option is not used, databases servers might raise an error when changing the connection context. For example IBM® Informix® will return the SQL error -1801: Multiple-server transaction not supported.

The example opens two database connections, reads rows from a table of the first database, and uses a transaction to insert rows in a table of the second database:

```sql
MAIN
   DEFINE rec RECORD
       pk INTEGER,
       name VARCHAR(50)
   END RECORD
   CONNECT TO "test1+driver='dbmifx'" AS "s1"
   USER "ifxuser" USING "fourjs"
   WITH CONCURRENT TRANSACTION
   CREATE TEMP TABLE tt1 ( pk INT, name VARCHAR(50) )
   INSERT INTO tt1 VALUES ( 1, "Item 1" )
   INSERT INTO tt1 VALUES ( 2, "Item 2" )
   CONNECT TO "test1+driver='dbmmys'" AS "s2"
   USER "mysuser" USING "fourjs"
   WITH CONCURRENT TRANSACTION
   CREATE TEMP TABLE tt2 ( pk INT, name VARCHAR(50) )
   SET CONNECTION "s1"
   DECLARE c1 CURSOR FOR SELECT * FROM tt1
   SET CONNECTION "s2"
   BEGIN WORK
   SET CONNECTION "s1"
   FOREACH c1 INTO rec.*
     SET CONNECTION "s2"
     INSERT INTO tt2 VALUES ( rec.* )
   SET CONNECTION "s1"
   END FOREACH
   SET CONNECTION "s2"
   COMMIT WORK
END MAIN
```

**Related concepts**
- Multi-session mode connection instructions on page 683
- Opening and closing a database for a unique session.
- Database transactions on page 688
Database transaction concepts and handling.

**Transaction savepoints**
Using transaction savepoints with different database engines.

IBM® Informix® IDS 11.50 introduced transaction savepoints, following the ANSI SQL standards. While most recent database servers support savepoints, you must pay attention and avoid Informix® specific features. For example, Oracle® (11), SQL Server (2008 R2), SAP® ASE (15.5) do not support the RELEASE SAVEPOINT instruction. The UNIQUE clause of SAVEPOINT is only supported by IBM® Informix® and IBM® DB2® LUW.

**Table 170: Transaction savepoint instructions by database server brand**

<table>
<thead>
<tr>
<th>Database Server Type</th>
<th>SAVEPOINT &amp; ROLLBACK WORK TO SAVEPOINT</th>
<th>RELEASE SAVEPOINT</th>
<th>SAVEPOINT UNIQUE</th>
<th>Related topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM® DB2® LUW</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>See details</td>
</tr>
<tr>
<td>IBM® Informix®</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>See details</td>
</tr>
<tr>
<td>IBM® Netezza®</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>See details</td>
</tr>
<tr>
<td>Microsoft™ SQL Server (Only 2005+ with SNC driver)</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>See details</td>
</tr>
<tr>
<td>Oracle® MySQL / Mariadb</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>See details</td>
</tr>
<tr>
<td>Oracle® Database Server</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>See details</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>See details</td>
</tr>
<tr>
<td>SAP® ASE</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>See details</td>
</tr>
<tr>
<td>SAP HANA®</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>See details</td>
</tr>
<tr>
<td>SQLite</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>See details</td>
</tr>
</tbody>
</table>

**Related concepts**

**SAVEPOINT** on page 690
Defines or resets the position of a rollback point in the current transaction.

**Data definition statements**

It is recommended to avoid use of DDL in programs.

When using Data Definition Statements like CREATE TABLE, ALTER TABLE, DROP TABLE, only a limited SQL syntax works on all database servers. Most databases support NOT NULL, CHECK, PRIMARY KEY, UNIQUE, FOREIGN KEY constraints, but the syntax for naming constraints is different.

The following statement works with most database servers and creates a table with equivalent properties in all cases:

```sql
CREATE TABLE customer  (
    cust_id INTEGER NOT NULL,
    cust_name CHAR(50) NOT NULL,
    cust_lastorder DATE NOT NULL,
    cust_group INTEGER,
    PRIMARY KEY (cust_id),
    UNIQUE (cust_name),
    FOREIGN KEY (cust_group) REFERENCES group (group_id)
)```
Some engines like SQL Server have a different default behavior for NULL columns when you create a table. You may need to set up database properties to make sure that a column allows nulls if the NOT NULL constraint is not specified.

When you want to create tables in programs using non-standard clauses (for example to define storage options), you must use dynamic SQL and adapt the statement to the target database server.

**Related concepts**

Dynamic SQL management on page 709
Explains how to execute and manage SQL statements at runtime.

**Using portable data types**

Only a limited set of data types are really portable across several database engines.

The ANSI SQL specification defines standard data types, but for historical reasons most databases vendors have implemented native (non-standard) data types. You can usually use a synonym for ANSI types, but the database server will uses the native types behind the scenes. For example, when you create a table with an INTEGER column in Oracle, the native NUMBER data type is used.

In your programs, avoid data types that do not have a native equivalent in the target database. This includes simple types like floating point numbers, as well as complex data types like INTERVAL. Numbers may cause rounding or overflow problems, because the values stored in the database have different limits. For the DECIMAL types, always use the same precision and scale for the program variables and the database columns.

To write portable applications, we strongly recommend using the following data types only:

- CHAR (n)
- VARCHAR(n)
- BIGINT
- INTEGER
- SMALLINT
- DECIMAL(p, s)
- DATE
- DATETIME HOUR TO MINUTE
- DATETIME HOUR TO SECOND
- DATETIME HOUR TO FRACTION(n)
- DATETIME YEAR TO MINUTE
- DATETIME YEAR TO SECOND
- DATETIME YEAR TO FRACTION(n)
- TEXT/BYTE (for LOBs)

**Table 171: Data type differences in database engine brands**

<table>
<thead>
<tr>
<th>Database Server Type</th>
<th>Data type topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM® DB2® LUW (UNIX™)</td>
<td>Data type conversion table: Informix to DB2 on page 766</td>
</tr>
<tr>
<td>IBM® Informix®</td>
<td>Genero BDL is base on Informix SQL data types...</td>
</tr>
<tr>
<td>IBM® Netezza®</td>
<td>Data type conversion table: Informix to Netezza on page 802</td>
</tr>
<tr>
<td>Microsoft™ SQL Server</td>
<td>Data type conversion table: Informix to SQL Server on page 837</td>
</tr>
<tr>
<td>Oracle® MySQL / MariadbDB</td>
<td>Data type conversion table: Informix to MySQL on page 882</td>
</tr>
<tr>
<td>Oracle® Database Server</td>
<td>Data type conversion table: Informix to Oracle on page 920</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>Data type conversion table: Informix to PostgreSQL on page 971</td>
</tr>
<tr>
<td>SAP® ASE</td>
<td>Data type conversion table: Informix to SAP ASE on page 1040</td>
</tr>
<tr>
<td>SAP HANA®</td>
<td>Data type conversion table: Informix to SAP HANA on page 1075</td>
</tr>
</tbody>
</table>
**Database Server Type** | **Data type topic**
--- | ---
SQLite | Data type conversion table: Informix to SQLite on page 1009

**Related concepts**

Primitive Data types on page 289

Selecting the correct data type assists you in the input, storage, and display of your data.

**Data manipulation statements**

Make sure that SQL statement syntaxes are supported by all target database engines.

Several SQL syntaxes for the INSERT, UPDATE and DELETE statements are supported by the compiler. Some of the syntaxes are IBM® Informix® specific, but will be converted to standard SQL at compile time.

The following statements are standard SQL and work with all database servers:

1. `INSERT INTO table (column-list) VALUES (value-list)`
2. `UPDATE table SET column = value, ... [WHERE condition]`
3. `DELETE FROM table [WHERE condition]`

The next statements are not standard SQL, but are converted by the compiler to standard SQL, working with all database servers:

4. `INSERT INTO table VALUES record.*` -- where `record` is defined LIKE a table from db schema
5. `UPDATE table SET (column-list) = (value-list) [WHERE condition]`
6. `UPDATE table SET [[table.]*](column-list) = record.* [WHERE condition]` -- where `record` is defined LIKE a table from db schema
7. `UPDATE table SET [[table.]*] = (value-list) [WHERE condition]`

For maximum SQL portability, INSERT statements should be reviewed to ensure the SERIAL column is excluded from the value list.

**Note:** You can easily search for non-portable SQL statements in your sources by compiling with the `-W stdsql fglcomp` option.

For example, the following statement:

```sql
INSERT INTO tab (col1, col2, ...) VALUES ( 0, p_value2, ... )
```

should be converted to:

```sql
INSERT INTO tab (col2, ...) VALUES ( p_value2, ... )
```

A static SQL INSERT statement using records defined from the schema file should also be reviewed:

```sql
DEFINE rec LIKE tab.*
INSERT INTO tab VALUES ( rec.* ) -- will use the serial column
```

should be converted to:

```sql
INSERT INTO tab VALUES rec.* -- without parentheses, serial column is removed
```

**Note:** Using the `record.*` notation in static INSERT and UPDATE syntax may not be compatible with database-specific features, where some automatically-assigned columns must not be set or modified by the statement. For example, with Microsoft SQL Server temporal tables, timestamp columns are automatically assigned and must not be changed by INSERT or UPDATE statements. In such case, it is mandatory to explicitly list all modifiable columns and corresponding program variables.
Related concepts
Static SQL statements on page 694
Describes static SQL statements supported in the language.

CHAR and VARCHAR types
Using the CHAR and VARCHAR data types with different databases.

The CHAR and VARCHAR types are designed to store character strings, but all database servers do not have the same semantics for these types.

Important: The maximum size, supported characters sets and length semantics of CHAR and VARCHAR types can be very different from one database system to another. Consider using character types and sizes that are common across all the database systems you target.

The behavior of database servers may differ in the following areas related to CHAR/VARCHAR types.

- Byte or Character Length semantics? on page 616
- SQL character type for Unicode/UTF-8 on page 617
- Size limits for CHAR/VARCHAR columns on page 618
- Empty strings and NULLs on page 619
- Trailing blanks in CHAR/VARCHAR on page 620
- What should you do? on page 622

Table 172: Character data type differences by database engine brands

<table>
<thead>
<tr>
<th>Database Server Type</th>
<th>Character types topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM® DB2® LUW (UNIX™)</td>
<td>CHAR/VARCHAR in IBM DB2</td>
</tr>
<tr>
<td>IBM® Informix®</td>
<td>CHAR/VARCHAR in IBM Informix</td>
</tr>
<tr>
<td>IBM® Netezza®</td>
<td>CHAR/VARCHAR in IBM Netezza</td>
</tr>
<tr>
<td>Microsoft™ SQL Server</td>
<td>CHAR/VARCHAR in SQL Server</td>
</tr>
<tr>
<td>Oracle® MySQL / Mariadb</td>
<td>CHAR/VARCHAR in Oracle MySQL</td>
</tr>
<tr>
<td>Oracle® Database Server</td>
<td>CHAR/VARCHAR in Oracle DB</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>CHAR/VARCHAR in PostgreSQL</td>
</tr>
<tr>
<td>SAP® ASE</td>
<td>CHAR/VARCHAR in SAP ASE</td>
</tr>
<tr>
<td>SAP HANA®</td>
<td>CHAR/VARCHAR in SAP HANA</td>
</tr>
<tr>
<td>SQLite</td>
<td>CHAR/VARCHAR in SQLite</td>
</tr>
</tbody>
</table>

Byte or Character Length semantics?
Length Semantics defines the unit used to express the length of a character string, the position of a given character, and the size of a character data type.

When defining a CHAR/VARCHAR database column or program variable, you must specify a size. When using a multibyte character set, the unit of this size matters: it can be specified in bytes or characters.

In programs, the size unit of CHAR/VARCHAR variables depends on the length semantics defined by the FGL_LENGTH_SEMANTICS environment variable.

In databases, the size unit of the CHAR/VARCHAR columns can be expressed in bytes or characters, depending on the database server and its configuration.

Related concepts
Length semantics settings on page 521
**SQL character type for Unicode/UTF-8**

This section explains database server specifics regarding Unicode / UTF-8 support with character string SQL types.

All database servers can store UNICODE data in character strings types, but there are some specifics you must be aware of. Genero BDL programs typically use the CHAR and VARCHAR types to store UTF-8 strings. But the corresponding SQL type may have a different name, depending on the database server type. Use the correct SQL type when creating your database tables. When the database uses a different UNICODE codeset as UTF-8 to store the character string data, the database client or the Genero database driver take care of the codeset conversion, as long as the runtime system and database client locale are properly defined.

**Table 173: Database server character types for Unicode / UTF-8 data**

<table>
<thead>
<tr>
<th>Database Server Type</th>
<th>Char types to be used for Unicode/UTF-8</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM® DB2® LUW</td>
<td>CHAR / VARCHAR if the database was created with UTF-8 codeset. Otherwise, you must use GRAPHIC, VARGRAPHIC types. For more details, see the topic about character data type usage with IBM DB2.</td>
</tr>
<tr>
<td>IBM® Informix®</td>
<td>CHAR / VARCHAR, the database must be created with UTF-8 locale. NCHAR / NVARCHAR (data always stored in UTF-8). For more details, see the topic about character data type usage with IBM Netezza.</td>
</tr>
<tr>
<td>IBM® Netezza</td>
<td>NCHAR / NVARCHAR, to store UTF-16 data (drivers make the conversion for application codeset UTF-8) For more details, see the topic about character data type usage with IBM Netezza.</td>
</tr>
<tr>
<td>Microsoft™ SQL Server</td>
<td>The CHAR/VARCHAR types can only store non-unicode data. For more details, see the topic about character data type usage with Microsoft SQL Server.</td>
</tr>
<tr>
<td>Oracle® MySQL</td>
<td>NCHAR / NVARCHAR if the database have to use the national character set. For more details, see the topic about character data type usage with Oracle MySQL.</td>
</tr>
<tr>
<td>Oracle Database Server</td>
<td>NCHAR / NVARCHAR2 if the database locale is UTF-8. For more details, see the topic about character data type usage with Oracle DB.</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>CHAR / VARCHAR if the database locale is UTF-8. For more details, see the topic about character data type usage with PostgreSQL.</td>
</tr>
<tr>
<td>SAP® ASE</td>
<td>NCHAR / NVARCHAR or UNICHAR / UNIVARCHAR if you need to use the national character set. For more details, see the topic about character data type usage with SAP ASE.</td>
</tr>
</tbody>
</table>
**Related concepts**

Language and character set settings on page 519

Database client settings on page 526

This section describes the settings defining the locale for the database client.

### Size limits for CHAR/VARCHAR columns

Each database brand defines its own limits for CHAR/VARCHAR SQL types.

#### CHAR and VARCHAR SQL types

To write portable SQL, you must use CHAR and VARCHAR type sizes that do not exceed the limits of the target database brands (Informix, Oracle, SQL Server, etc).

The CHAR type is a blank-padded type: The string "abc" in a CHAR(200) column will in fact get 197 additional trailing blancs. The VARCHAR type is designed to store variable-length character strings. Thus, the CHAR(N) type should be used for fixed-length string data (such as "ABC-123" codes), while VARCHAR(N) should be used for variable-length string data (such as names and descriptions). Furthermore, with some database brands, in VARCHAR values, trailing spaces are semantically significant, while trailing blanks of CHAR SQL values are not significant.

With Informix, the VARCHAR SQL type was designed with a limit of 255 bytes. On the other hand, the CHAR type has a much larger limit of 32767 bytes. In fact, it should have been the other way around, since the blank-padded CHAR type is rather for small fix-length data.

For large variable-length character strings, Informix has introduced the LVARCHAR type, first with a limit of 2048 bytes and in younger IDS versions with a limit of 32739 bytes.

Most database brands supported by Genero have a higher limit for VARCHAR than the CHAR type. For example, Oracle DB CHAR has a limit of 2000, while VARCHAR2 has a limit of 4000 (bytes or chars, depending on the length semantics).

**Note:** The Genero BDL VARCHAR type size limit is 65524 (like the CHAR BDL type), and thus can store Informix LVARCHAR or any database brand VARCHAR column data.

#### What CHAR length should be used?

To store short and fix-length character data, use a CHAR(N) column, where n will typically be 10 to 100 (bytes or characters): All database brands supported by Genero can support such CHAR size.

#### What VARCHAR length should be used?

To store variable-length character strings that do not exceed 255 bytes, use the VARCHAR(N) type: All database brands supported by Genero can support such VARCHAR size.

To store variable-length character strings that can exceed 255 bytes:

- If you target only non-Informix databases, use VARCHAR(2048) type in the CREATE TABLE / ALTER TABLE statements in BDL code: That will compile (no Informix 255 limit) and the SQL will also execute.
• If you want to support all databases including Informix, use LVARCHAR(N) type in CREATE TABLE / ALTER TABLE statements in BDL code: The LVARCHAR Informix-specific type will be converted to VARCHAR(N) with other database brands.

Related concepts
Length semantics settings on page 521

Empty strings and NULLs
Depending on the context, an empty string ("") can be considered as NULL or NOT NULL.

Most SQL databases distinguish "" empty strings from NULL (with some exceptions like Oracle® DB).

With Genero BDL character string variables, an empty string is the equivalent to NULL:

```BDL
MAIN
  DEFINE vc10 VARCHAR(10)
  LET vc10 = ''
  DISPLAY (vc10 IS NULL)   -- shows 1 / TRUE
END MAIN
```

When inserting an empty string SQL literal into a DB row, and fetching the value back into a variable, the result might be null or non-null, depending on the database type and target variable type, since CHAR values are blank-padded.

See for example the following code:

```BDL
DEFINE rec RECORD
  pk INTEGER,
  c10 CHAR(10),
  vc10 VARCHAR(10)
END RECORD

MAIN
  -- CONNECT TO ... (your connection instruction goes here)
  DISPLAY "DB driver: ", fgl_db_driver_type()
  CREATE TEMP TABLE tt1 ( pk INT, c10 CHAR(10), vc10 VARCHAR(10) )
  INSERT INTO tt1 VALUES ( 101, ' ', ' ' )
  INSERT INTO tt1 VALUES ( 102, NULL, NULL )
  INSERT INTO tt1 VALUES ( 103, '', '' )
  SELECT tt1.* INTO rec.* FROM tt1 WHERE pk=101
  CALL show_record()
  SELECT tt1.* INTO rec.* FROM tt1 WHERE pk=102
  CALL show_record()
  SELECT tt1.* INTO rec.* FROM tt1 WHERE pk=103
  CALL show_record()
END MAIN

FUNCTION show_record()
  DISPLAY rec.pk, " [", NVL(rec.c10,"<NULL>"),","]",
    COLUMN 30, "[",NVL(rec.vc10,"<NULL>"),","]
END FUNCTION
```

Result with Informix IDS 14, PostgreSQ 12, MySQL 8.0, DB2 11.5:

```
DB driver: ifx
  101 [ ] [ ]
```
Result with Oracle DB 19c:

<table>
<thead>
<tr>
<th></th>
<th>[&lt;NULL&gt;]</th>
<th>[&lt;NULL&gt;]</th>
</tr>
</thead>
<tbody>
<tr>
<td>102</td>
<td>[</td>
<td>[</td>
</tr>
<tr>
<td>103</td>
<td>[&lt;NULL&gt;]</td>
<td>[&lt;NULL&gt;]</td>
</tr>
</tbody>
</table>

Tip: Mixing empty strings and NULL values in database rows makes queries more difficult: Is an empty string considered as null or not null by your application? To avoid empty strings in database columns, always use program variables to insert data into SQL tables: If a CHAR or VARCHAR program variable is assigned with an empty string, it will be set to NULL, and inserted as NULL into the database.

### Trailing blanks in CHAR/VARCHAR

How to cope with trailing blanks in CHAR (N) and VARCHAR (N) SQL columns and program variables?

#### Trailing blanks in CHAR/VARCHAR database columns

With all kinds of databases servers, CHAR columns are always filled with blanks up to the size of the column (this is called blank padding).

With IBM® Informix®, trailing blanks are not significant in comparisons:

```
CHAR('abc ') = CHAR('abc')
```

With other database servers, trailing blanks are significant when comparing VARCHAR values:

```
VARCHAR('abc ') != VARCHAR('abc')
```

This is a major issue if you mix CHAR and VARCHAR columns and variables in your SQL statements, because the result of an SQL query can be different depending on whether you are using IBM® Informix® or another database server.

Furthermore, the semantics of the SQL LIKE operator regarding trailing blanks and CHAR/VARCHAR types can differ from database to database. For example, try the following expressions with your database, with a CHAR(5) column containing a row with the value 'abc':

```
CREATE TABLE t1 ( k INT, c CHAR(5), vc VARCHAR(5) )
INSERT INTO t1 VALUES ( 1, 'abc', 'abc' )
SELECT * FROM t1 WHERE c LIKE 'ab_'
SELECT * FROM t1 WHERE vc LIKE 'ab_'
SELECT * FROM t1 WHERE RTRIM(c) LIKE 'ab_'
SELECT * FROM t1 WHERE c LIKE '%c'
SELECT * FROM t1 WHERE vc LIKE '%c'
SELECT * FROM t1 WHERE RTRIM(c) LIKE '%c'
```

See discussion about MATCHES and LIKE operators in adaption guides for more details.

### CHAR blank padding versus VARCHAR

In all database engines, CHAR (N) data is blank padded. This means that the database engine fills the column value with trailing blanks when needed.

Because of the IBM® Informix® SQL VARCHAR() limit of 255 bytes, there may be legacy code using the CHAR data type for larger sizes, such as CHAR(400) or CHAR(2000).

The problem with large CHAR() columns is blank padding. For example, with CHAR(500), if you store and pass the value "abc" in program variables, SQL parameters, and SQL statements, you pass abc + 497 blanks, because CHAR values are blank-padded.
On the other hand, VARCHAR types only store the actual value that was provided ('abc' in our case), and trailing blanks only if explicitly provided ('abc ').

In order to optimize your application, consider replacing large CHAR() columns by VARCHAR(). The CHAR() type can be used to store small, fixed-size, character string values (such as phone or credit card numbers).

Informix® IDS supports the LVARCHAR() type with a larger limit. Genero BDL supports this Informix® SQL type.

See also Passing CHAR parameters to functions on page 587.

**Trailing blanks in CHAR/VARCHAR program variables**

In programs, CHAR variables are filled with blanks, even if the value used does not contain all spaces.

The following example:

```plaintext
DEFINE c CHAR(5)
LET c = "abc"
DISPLAY c || "."
```

shows the value "abc ." (5 chars + dot).

VARCHAR variables are assigned with the exact value specified, with significant trailing blanks.

For example, this code:

```plaintext
DEFINE v VARCHAR(5)
LET v = "abc ".
DISPLAY v || "."
```

shows the value "abc ." (4 chars + dot).

Assigning an empty string to a CHAR or VARCHAR variable will set the variable to NULL:

```plaintext
DEFINE v VARCHAR(5)
LET v = ""
IF v IS NULL THEN
  DISPLAY "is null" -- will be displayed
END IF
```

When comparing CHAR or VARCHAR variables in an expression, the trailing blanks are not significant:

```plaintext
DEFINE c CHAR(5)
DEFINE v1, v2 VARCHAR(5)
LET c = "abc"
LET v1 = "abc ".
LET v2 = "abc  ".
IF c == v1 THEN
  DISPLAY "c==v1"
END IF
IF c == v2 THEN
  DISPLAY "c==v2"
END IF
IF v1 == v2 THEN
  DISPLAY "v1==v2"
END IF
```

All three messages are shown.

Additionally, when you assign a VARCHAR variable from a CHAR, the target variable gets the trailing blanks of the CHAR variable:

```plaintext
DEFINE pc CHAR(50)
```
DEFINE pv VARCHAR(50)
LET pc = "abc"
LET pv = pc
DISPLAY pv || "."

"abc <47 spaces>. " (50 chars + dot) is shown.

To avoid this, use the CLIPPED operator:

LET pv = pc CLIPPED

### Trailing blanks in SQL statement parameters

When you insert a row containing a CHAR variable into a CHAR or VARCHAR column, the database interface removes the trailing blanks to avoid overflow problems, (insert CHAR(100) into CHAR(20) when value is "abc" must work).

In this example:

```sql
DEFINE c CHAR(5)
LET c = "abc"
CREATE TABLE t ( v1 CHAR(10), v2 VARCHAR(10) )
INSERT INTO tab VALUES ( c, c )
```

The value in column v1 and v2 would be "abc" (3 chars in both columns).

When you insert a row containing a VARCHAR variable into a VARCHAR column, the VARCHAR value in the database gets the trailing blanks as set in the variable. When the column is a CHAR(N), the database server fills the value with blanks so that the size of the string is N characters.

In this example:

```sql
DEFINE vc VARCHAR(5)
LET vc = "abc  " -- note 2 spaces at end of string
CREATE TABLE t ( v1 CHAR(10), v2 VARCHAR(10) )
INSERT INTO tab VALUES ( vc, vc )
```

The value in column v1 would be "abc " (10 chars) and v2 would be "abc " (5 chars).

### Related concepts

**MATCHES and LIKE operators** on page 645

Use the standard LIKE operator instead of the MATCHES operator.

**CHAR(size)** on page 293

The CHAR data type is a fixed-length character string data type.

**VARCHAR(size)** on page 309

The VARCHAR data type is a variable-length character string data type, with a maximum size.

**CLIPPED** on page 358

The CLIPPED operator removes trailing blank spaces (ASCII 32) of a string expression.

### What should you do?

This section contains facts and tips to consider regarding character data types and locale settings.

Make sure that you have correctly defined the locale and length semantics for your character string data types.

When designing your database tables, consider using CHAR(N) for fixed-length string data (such as codes) and VARCHAR(N) for variable-length string data, such as names, address and comments.

Use VARCHAR variables for VARCHAR columns, and CHAR variables for CHAR columns, to achieve portability across all kinds of database servers.
Note: Note that with SQL Server, the column types can be NCHAR/NVARCHAR National Character types, to Unicode character types to be combined with CHAR/VARCHAR program variables. SQL Server 2019 supports UTF-8 collation for CHAR/VARCHAR columns.

Avoid storing empty strings in VARCHAR columns, or make sure that your program is prepared to get nulls while the database stores empty strings.

Using byte or character length semantics depends mainly on the character set of your application:

- When using a single-byte character set, keep the default byte length semantics.
- When using a multibyte character set such as UTF-8, use character length semantics in both the database and the programs.

The database column definition and the program variable definition must match, this can be simplified by using a database schema.

Related concepts
Defining the application locale on page 519
This section describes the settings defining the application locale, changing the behavior of the compilers and runtime system.

Database schema on page 467
Defines database table structures with column type information to be reused in program variable definitions.

The LENGTH() function in SQL on page 647
The semantics of the LENGTH() SQL function differs according to the database engine.

Scrollable cursors
How scrollable cursors can be supported on different databases.

Scrollable cursors can be used to go forward and backward in an SQL query result set:

```
DEFINE cust_rec RECORD LIKE customer.*
DECLARE sc SCROLL CURSOR
   FOR SELECT * FROM customer
OPEN sc
FETCH NEXT sc INTO cust_rec.*
FETCH LAST sc INTO cust_rec.*
FETCH FIRST sc INTO cust_rec.*
CLOSE sc
```

This is a useful feature to implement record set navigation in applications. Scrollable cursors are typically implemented in the database server. But not all database servers support scrollable cursors.

When scrollable cursors are not supported by the target database server, the database driver will emulate it with temporary files.

The temporary files are created in a temporary directory, that can be defined with the DBTEMP environment variable. If DBTEMP is not defined, the default temporary directory dependents from the platform used.

It is recommended that you avoid scrollable cursor usage if the target database does not support this feature:

With emulated scrollable cursors, when scrolling to the last row, all rows will be fetched into the temporary file. This can generate a lot of network traffic and can produce a large temporary file if the result-set contains a lot of rows. Additionally, programs are dependent on the file system resource allocated to the OS user (ulimit).

Some databases do not support to use large objects data types (TEXT/BYTE) with scrollable cursors: The OPEN statement will produce SQL error -611. To write portable SQL, use only simple data types in the result set of the scrollable cursor, and use the primary key column to fetch TEXT/BYTE data in a secondary SELECT statement.

The following table lists the native scrollable cursor availability for each supported database:
Table 174: Database server support for scrollable cursors

<table>
<thead>
<tr>
<th>Database Server Type</th>
<th>Scrollable cursors support</th>
<th>TEXT/BYTE support (with scrollable cursors)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM® DB2® LUW</td>
<td>Yes, see details</td>
<td>No</td>
</tr>
<tr>
<td>IBM® Informix®</td>
<td>Yes, native SQL feature</td>
<td>No</td>
</tr>
<tr>
<td>IBM® Netezza</td>
<td>Emulated, see details</td>
<td>No (N/A)</td>
</tr>
<tr>
<td>Microsoft® SQL Server</td>
<td>Yes, see details</td>
<td>Yes</td>
</tr>
<tr>
<td>Oracle® MySQL / Mariadb</td>
<td>Emulated, see details</td>
<td>No</td>
</tr>
<tr>
<td>Oracle® Database Server</td>
<td>Yes, see details</td>
<td>Yes</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>Yes, see details</td>
<td>Yes</td>
</tr>
<tr>
<td>SAP® ASE</td>
<td>Yes, see details</td>
<td>No</td>
</tr>
<tr>
<td>SAP HANA®</td>
<td>Emulated, see details</td>
<td>No</td>
</tr>
<tr>
<td>SQLite</td>
<td>Emulated, see details</td>
<td>No</td>
</tr>
</tbody>
</table>

Related concepts

DECLARE (result set cursor) on page 717
Associates a database cursor with an SQL statement producing a result set.

DBTEMP on page 273
Defines the directory for temporary files.

Optimizing scrollable cursors on page 654
A programming pattern to get fresh data from scrollable cursors.

Optimistic locking

Implementing optimistic locking to handle access concurrently to the same database records.

This section describes how to implement optimistic locking in applications. Optimistic locking is a portable solution to control simultaneous modification of the same record by multiple users.

Traditional IBM® Informix® applications use a cursor declared with SELECT FOR UPDATE to set a lock on the row to be edited by the user. This is called pessimistic locking. The SELECT FOR UPDATE cursor is executed before the interactive part of the code, as described in here:

1. When the end user chooses to modify a record, the program declares and opens a cursor with a SELECT FOR UPDATE. At this point, an SQL error might be raised if the record is already locked by another process. Otherwise, the lock is acquired and user can modify the record.
2. The user edits the current record in the input form.
3. The user validates the dialog.
4. The UPDATE SQL instruction is executed.
5. The transaction is committed or the SELECT FOR UPDATE cursor is closed. The lock is released.

If the IBM® Informix® database was created with transaction logging, you must either start a transaction or define the SELECT FOR UPDATE cursor WITH HOLD option.

Unfortunately, this is not a portable solution. The lock wait mode should preferably be WAIT for portability reasons. Pessimistic locking is based on a NOT WAIT mode to return control to the program if a record is already locked by another process. Therefore, following the portable concurrency model, the pessimistic locking mechanisms must be replaced by the optimistic locking technique.

Basically, instead of locking the row before the user starts to modify the record data, the optimistic locking technique makes a copy of the current values (i.e. before modification values (BVM)), allows the user to edit the record,
and when it’s time to write data into the database, checks if the BMVs still correspond to the current values in the database:

1. A `SELECT` is executed to fill the record variable used by the interactive instruction for modifications.
2. The record variable is copied into a backup record to keep Before Modification Values.
3. The user enters modifications in the input form; this updates the values in the modification record.
4. The user validates the dialog.
5. A transaction is started with `BEGIN WORK`.
6. Declare a cursor with a `SELECT FOR UPDATE`, to select the row to be updated.
7. Open the `SELECT FOR UPDATE` cursor and fetch the row into the temporary record.
8. If `SQLCA.SQLCODE` == `NOTFOUND`, the row has been deleted by another process, and the transaction can stop with `ROLLBACK WORK`.
9. If the row is found, the program compares the temporary record values with the backup record values with the `(rec1.*==rec2.*)` notation.
10. If these values have changed, the row has been modified by another user. At this stage, you can let the end user choose to ignore the changes done by another user and continue with the next step, or stop the transaction with `ROLLBACK WORK` and show a message indicating that the row cannot be updated because it would overwrite changes done by another user.
11. If the values in the database have not changed, the `UPDATE` statement is executed to save the changes done by the current user.
12. The transaction is committed with a `COMMIT WORK`.

To compare 2 records (with NULL checking), simply write:

```sql
IF new_record.* != bmv_record.* THEN
   LET values_have_changed = TRUE
END IF
```

The optimistic locking technique can be implemented with a unique SQL instruction: an `UPDATE` can compare the column values to the BMVs directly (`UPDATE ... WHERE kcol = kvar AND col1 = bmv.var1 AND ...`). But, this is not possible when BMVs can be NULL. The database engine always evaluates conditional expressions such as "col=NULL" to FALSE. Therefore, you must use "col IS NULL" when the BMV is NULL. This means dynamic SQL statement generation based on the DMV values. Additionally, to use the same number of SQL parameters ( markers), you would have to use "col=?" when the BMV is not null and "col IS NULL and ? IS NULL" when the BMV is null. Unfortunately, the expression " ? IS [NOT] NULL " is not supported by all database servers (DB2® raises error SQL0418N).

If you are designing a new database application from scratch, you can also use the row versioning method. Each table of the database must have a column that identifies the current version of the row. The column can be a simple `INTEGER` (to hold a row version number) or it can be a timestamp (`DATETIME YEAR TO FRACTION(5)` for example). To guaranty that the version or timestamp column is updated each time the row is updated, it is recommended that you implement a trigger to increment the version or set the timestamp when an `UPDATE` statement is issued. If this is in place, you just need to check that the row version or timestamp has not changed since the user modifications started, instead of testing all field of the BMV record. If you are only using one specific database type, you may check if the server supports a versioning column natively. For example, IBM® Informix® IDS 11.50.xC1 introduced the `ALTER TABLE ... ADD VERCOLS` option to get a version + checksum column to a table, you can then query the table with the `ifx_insert_checksum` and `ifx_row_version` columns.

**Related concepts**

- Database transactions on page 688
- Database transaction concepts and handling.
- Records on page 408
Records allow structured program variables definitions.

**Auto-incremented columns (serials)**
How to implement automatic record keys.

IBM® Informix® provides the SERIAL, BIGSERIAL or SERIAL8 data types which can be emulated with database drivers for most non-Informix® database engines by using native sequence generators (when "ifxemul.serial" FGLPROFILE setting is true).

But, this requires additional configuration and maintenance tasks. If you plan to review the programming pattern of sequences, it is recommended that you use a portable implementation instead of the serial emulation provided by the database drivers.

This section describes different solutions to implement **auto-incremented fields**. The preferred implementation is the solution using SEQUENCES.

**Table 175: SERIAL emulation by database server brand**

<table>
<thead>
<tr>
<th>Database Server Type</th>
<th>Serial type support</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM® DB2® LUW</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>IBM® Informix®</td>
<td>Yes, this is a native Informix feature</td>
</tr>
<tr>
<td>IBM® Netezza</td>
<td>No, see details</td>
</tr>
<tr>
<td>Microsoft™ SQL Server</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>Oracle® MySQL / Mariadb</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>Oracle® Database Server</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>SAP® ASE</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>SAP HANA®</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>SQLite</td>
<td>Emulated, see details</td>
</tr>
</tbody>
</table>

**Solution 1: Use database specific serial generators**

**Principle**

In accordance with the target database, you must use the appropriate native serial generation method. Get the database type with the fgl_db_driver_type() built-in function and use the appropriate SQL statements to insert rows with serial generation.

This solution uses the native auto-increment feature of the target database and is fast at execution time, but is not very convenient as it requires different code for each database type to be written. Solutions for auto-incremented columns are vendor-specific. It is of course not realistic to use this solution in a large application with hundreds of tables.

**Implementation**

1. Create the database objects required for serial generation in the target database (for example, create tables with SERIAL columns in IBM® Informix®, tables with IDENTITY columns in SQL Server, and SEQUENCE database objects in Oracle).
2. Adapt your programs to use the native sequence generators in accordance with the database type.

**Example**

```
DEFINE t1rec RECORD
  id    INTEGER,
  name  CHAR(50),
```
As you can see in this example, this solution requires database engine specific coding. Querying the last generated serial can be centralized in a function, but the insert statements would still need to be specific to the type of database.

**Related concepts**

*fgl_db_driver_type()* on page 2275

Returns the 3-letter identifier/code of the current database driver.

**Solution 2: Generate serial numbers from your own sequence table**

**Purpose**

The goal is to generate unique INTEGER or BIGINT numbers. These numbers will usually be used for primary keys.

**Prerequisites**

1. The database must use transactions. This is usually the case with non-Informix databases, but IBM® Informix® databases default to auto-commit mode. Make sure your IBM® Informix® database allows transactions.
2. The sequence generation must be called inside a transaction (BEGIN WORK/COMMIT WORK).
3. The transaction isolation level must guarantee that a row UPDATED in a transaction cannot be read or written by other db sessions until the transaction has ended (typically, COMMITTED READ is ok, but some db servers require a higher isolation level)
4. The lock wait mode must be WAIT. This is usually the case in non-Informix databases, but Informix defaults to NOT WAIT. You must change the lock wait mode with "SET LOCK MODE TO WAIT" or "WAIT seconds" when using IBM® Informix®.
5. Other applications or stored procedures must implement the same technique when inserting records in tables with auto-incremented columns.

**Principle**

A dedicated table named "SEQREG" is used to register sequence numbers. The key is the name of the sequence. This name will usually be the table name the sequence is generated for. In short, this table contains a primary key that identifies the sequence and a column containing the last generated number.

The uniqueness is granted by the concurrency management of the database server. The first executed instruction is an UPDATE that sets an exclusive lock on the SEQREG record. When two processes try to get a sequence at the same time, one will wait for the other until its transaction is finished.
Implementation
The "fgldbutl.4gl" utility library implements a function called "db_get_sequence()" which generates a new sequence. You must create the SEQREG table as described in the fgldbutl.4gl source found in FGLDIR/src, and make sure that every user has the privileges to access and modify this table.

In order to guarantee the uniqueness of the generated number, the call to db_get_sequence() must be done inside a transaction block that includes the INSERT statement. Concurrent db sessions must wait for each other in case of conflict and the transaction isolation level must be high enough to make sure that the row of the sequence table will not be read or written by other db sessions until the transaction end.

Example

```
IMPORT FGL fgldbutl
DEFINE rec RECORD
     id    INTEGER,
     name  CHAR(100)
END RECORD
...
BEGIN WORK
LET rec.id = db_get_sequence( "CUSTID" )
INSERT INTO CUSTOMER ( CUSTID, CUSTNAME ) VALUES ( rec.* )
COMMIT WORK
```

Related concepts
- Database transactions on page 688
  Database transaction concepts and handling.
- SET LOCK MODE on page 693
  Defines the behavior of the program that tries to access a locked row or table.
- SET ISOLATION on page 692
  Defines the transaction isolation level for the current connection.

Related reference
- fgldbutl: Database utility module on page 2307

Solution 3: Use native SEQUENCE database objects

Principle
Most recent database engines support SEQUENCE database objects. If all database server types you want to use support sequences, it is recommended that you use this solution.

Implementation
1. Create a SEQUENCE object for each table that previously used a SERIAL column in the IBM® Informix® database.
2. In database creation scripts (CREATE TABLE), replace all SERIAL types by INTEGER (or BIGINT if you need large integers).
3. Adapt your programs to retrieve a new sequence before inserting a new row. Consider writing a function to retrieve a new sequence number, using dynamic SQL to pass the name of the sequence as parameter, and adapt to the target database specifics to retrieve a single row (see example below).

Note: For databases not supporting SEQUENCE objects, the code uses SELECT MAX(col) +1 to get a new number, assuming that it's a database like SQLite used by a single process, where duplicates cannot occur. In a multi-user/process case, for databases that do not support sequences, consider using the user-made sequence table solution.

Example

```
MAIN
     DEFINE item_rec RECORD
```
CREATE TABLE item (  
    item_num BIGINT NOT NULL PRIMARY KEY,  
    item_name VARCHAR(50)  
)  
CALL sequence_create("item","item_num")  
FOR x=1 TO 5  
    LET item_rec.item_num = sequence_next("item","item_num")  
    DISPLAY "New sequence: ", item_rec.item_num  
    LET item_rec.item_name = "Item#" || item_rec.item_num  
    INSERT INTO item VALUES ( item_rec.* )  
END FOR  
DROP TABLE item  
CALL sequence_drop("item")  
END MAIN
CASE
  WHEN NOT _supports_sequences()
    -- WARNING: Duplicates can be created in multi-user databases,
    must
    -- be controlled with PRIMARY KEY or UNIQUE constraint!
    RETURN _get_next_pkey(tabname,colname)
  WHEN fgl_db_driver_type()=='pgs'
    LET sql = SFMT("SELECT nextval('%1')%2",seqname,_unique_row_condition())
  WHEN fgl_db_driver_type()=='esm'
    OR fgl_db_driver_type()=='ftm'
    OR fgl_db_driver_type()=='snc'
    OR fgl_db_driver_type()=='ntz'
    LET sql = SFMT("SELECT NEXT VALUE FOR %1",seqname)
  OTHERWISE
    LET sql = SFMT("SELECT %1.nextval%2",seqname,_unique_row_condition())
END CASE
TRY
  PREPARE seq FROM sql
  EXECUTE seq INTO val
CATCH
  DISPLAY SFMT("ERROR: Could not produce new sequence for %1", tabname)
  EXIT PROGRAM 1
  RETURN NULL
END TRY
RETURN val
END FUNCTION

Related concepts

INTEGER on page 300
The INTEGER data type is used for storing large whole numbers.

BIGINT on page 290
The BIGINT data type is used for storing very large whole numbers.

IBM® Informix® SQL ANSI Mode
Understanding the impact of the SQL ANSI mode of IBM® Informix®.

IBM® Informix® allows you to create databases in ANSI mode, which is supposed to be closer to ANSI standard behavior. Other databases like ORACLE and DB2® are 'ANSI' by default.

If you are not using the ANSI mode with IBM® Informix®, we suggest you keep the database as is, because turning an IBM® Informix® database into ANSI mode can result in unexpected behavior of the programs.

Here are some ANSI mode issues extracted from the IBM® Informix® books:

- Some actions, like CREATE INDEX will generate a warning but will not be forbidden.
- Buffered logging is not allowed to enforce data recovery. (Buffered logging provides better performance)
- The table-naming scheme allows different users to create tables without having to worry about name conflicts.
- Owner specification is required in database object names (SELECT ... FROM "owner").table). You must quote the owner name to prevent automatic translation of the owner name into uppercase: SELECT ... FROM owner.table becomes SELECT ... FROM owner.table and thus, the table is not found in the database.
- Default privileges differ: When creating a table, the server grants privileges to the table owner and the DBA only. The same thing happens for the 'Execute' privilege when creating stored procedures.
- Default isolation level is REPEATABLE READ.
- An error is generated if any character field is filled with a value that is longer than the field width.
- DECIMAL (p) (floating point decimals) are automatically converted to DECIMAL (p, 0) (fixed point decimals).
- Closing a closed cursor generates an SQL error.
It will take more time to adapt the programs to the IBM® Informix® ANSI mode than using the database interface to simulate the native mode of IBM® Informix®.

**Positioned UPDATE/DELETE**
Using positioned updates/deletes with named database cursors.

The "WHERE CURRENT OF cursor-name" clause in UPDATE and DELETE statements is not supported by all database engines.

**Table 176: Database server support of WHERE CURRENT OF**

<table>
<thead>
<tr>
<th>Database Server Type</th>
<th>WHERE CURRENT OF support</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM® DB2® LUW</td>
<td>Yes, see details</td>
</tr>
<tr>
<td>IBM® Informix®</td>
<td>Yes, see details</td>
</tr>
<tr>
<td>IBM® Netezza</td>
<td>No, see details</td>
</tr>
<tr>
<td>Microsoft™ SQL Server</td>
<td>Yes, see details</td>
</tr>
<tr>
<td>Oracle® MySQL / Mariadb</td>
<td>No, see details</td>
</tr>
<tr>
<td>Oracle Database Server</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>Yes, see details</td>
</tr>
<tr>
<td>SAP® ASE</td>
<td>Yes, see details</td>
</tr>
<tr>
<td>SAP HANA®</td>
<td>No, see details</td>
</tr>
<tr>
<td>SQLite</td>
<td>No, see details</td>
</tr>
</tbody>
</table>

Some database drivers can emulate WHERE CURRENT OF mechanisms by using rowids, but this requires additional processing. You should review the code to disable this option.

The standard SQL solution is to use primary keys in all tables and write UPDATE / DELETE statements with a WHERE clause based on the primary key:

```sql
DEFINE rec RECORD
  id    INTEGER,
  name  CHAR(100)
END RECORD
BEGIN WORK
  UPDATE CUSTOMER SET CUSTNAME = rec.name
  WHERE CUSTID = rec.id
  ...
  COMMIT WORK
```

**Cursors WITH HOLD**
Programming WITH HOLD cursors using SELECT with and without FOR UPDATE clause.

**Type of WITH HOLD cursors**
Make sure to distinguish cursors declared WITH HOLD for regular SELECT statements, and WITH HOLD cursors declared for a SELECT statements using the FOR UPDATE clause.

**Regular WITH HOLD cursors (without FOR UPDATE clause in SELECT)**
Most database brands allow cursors to remain open across transactions, as long as the SELECT statement is not defined with a FOR UPDATE clause.

```sql
DECLARE c1 CURSOR WITH HOLD FOR SELECT * FROM customers
```
Therefore, a cursor declared WITH HOLD with a SELECT statement not using the FOR UPDATE clause can be used with most databases. However, some databases close any cursor (even declared without FOR UPDATE), when a transaction is rolled back, or when the cursor is opened inside the transaction block.

**Table 177: Database server support of WITH HOLD cursors (with no FOR UPDATE clause)**

<table>
<thead>
<tr>
<th>Database Server Type</th>
<th>WITH HOLD support</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM® DB2® LUW</td>
<td>Partial, see details</td>
</tr>
<tr>
<td>IBM® Informix®</td>
<td>Yes, native SQL feature</td>
</tr>
<tr>
<td>IBM® Netezza</td>
<td>Yes, see details</td>
</tr>
<tr>
<td>Microsoft™ SQL Server</td>
<td>Yes, see details</td>
</tr>
<tr>
<td>Oracle® MySQL / Mariadb</td>
<td>Yes, see details</td>
</tr>
<tr>
<td>Oracle® Database Server</td>
<td>Yes, see details</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>Partial, see details</td>
</tr>
<tr>
<td>SAP® ASE</td>
<td>Yes, see details</td>
</tr>
<tr>
<td>SAP HANA®</td>
<td>Yes, see details</td>
</tr>
<tr>
<td>SQLite</td>
<td>Yes, see details</td>
</tr>
</tbody>
</table>

**Cursors WITH HOLD using SELECT FOR UPDATE**

IBM® Informix® supports WITH HOLD cursors using the FOR UPDATE clause:

```
DECLARE c1 CURSOR WITH HOLD FOR SELECT * FROM customers FOR UPDATE
```

Such cursors can remain open across transactions (when using FOR UPDATE, locks are released at the end of a transaction, but the WITH HOLD cursor is not closed).

Not all database brands allow "for update" cursors to remain open across transactions: The SQL standards recommend closing FOR UPDATE cursors and release locks at the end of a transaction.

Most database servers close FOR UPDATE cursors when a COMMIT WORK or ROLLBACK WORK is done.

All database servers release locks when a transaction ends.

**Table 178: Database server support of WITH HOLD FOR UPDATE**

<table>
<thead>
<tr>
<th>Database Server Type</th>
<th>WITH HOLD FOR UPDATE support</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM® DB2® LUW</td>
<td>No, see details</td>
</tr>
<tr>
<td>IBM® Informix®</td>
<td>Yes, native SQL feature</td>
</tr>
<tr>
<td>IBM® Netezza</td>
<td>No, see details</td>
</tr>
<tr>
<td>Microsoft™ SQL Server</td>
<td>No, see details</td>
</tr>
<tr>
<td>Oracle® MySQL / Mariadb</td>
<td>No, see details</td>
</tr>
<tr>
<td>Oracle® Database Server</td>
<td>No, see details</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>No, see details</td>
</tr>
<tr>
<td>SAP® ASE</td>
<td>No, see details</td>
</tr>
<tr>
<td>SAP HANA®</td>
<td>No, see details</td>
</tr>
<tr>
<td>SQLite</td>
<td>No, see details</td>
</tr>
</tbody>
</table>
It is mandatory to review code using WITH HOLD cursors with a SELECT statement having the FOR UPDATE clause.

The standard SQL solution is to declare a FOR UPDATE cursor (without WITH HOLD clause) outside the transaction and open the cursor inside the transaction:

```sql
DECLARE c1 CURSOR FOR SELECT ... FOR UPDATE
BEGIN WORK
  OPEN c1
  FETCH c1 INTO ...
  UPDATE ...
END WORK
```

If you need to process a large result set by doing updates of master and detail rows, first fetch the primary keys of all master rows into a program array, declare a cursor with the SELECT FOR UPDATE, for each row in the array, start a transaction and perform the UPDATE WHERE CURRENT OF for the current master record and the UPDATE for detail rows, then commit the transaction and continue with the next master record:

```sql
DEFINE x, mkeys DYNAMIC ARRAY OF INTEGER
DECLARE c1 CURSOR FOR SELECT key FROM master ...
FOREACH c1 INTO x
  LET mkeys[mkeys.getLength()+1] = x
END FOREACH
DECLARE c2 CURSOR FOR SELECT * FROM master WHERE key=? FOR UPDATE
FOR x = 1 TO mkeys.getLength()
  BEGIN WORK
    OPEN c2 USING mkeys[x]
    FETCH c2 INTO mrec.*
    IF STATUS==NOTFOUND THEN
      ROLLBACK WORK
      CONTINUE FOREACH
    END IF
    UPDATE master SET ... WHERE CURRENT OF c2
    UPDATE detail SET ... WHERE master_key=mkeys[x]
  COMMIT WORK
END FOR
```

Note: When the underlying database table is defined with a primary key column, consider replacing the WHERE CURRENT OF clause by WHERE pkey-column = ?, which is a more portable syntax. With some types of databases, the WHERE CURRENT OF clause has to be emulated, or is simply unsupported. For more details, see Positioned UPDATE/DELETE on page 631.

Related concepts
DECLARE (SELECT ... FOR UPDATE) on page 728
Associate a database cursor with a SELECT statement to perform positioned updates and deletes

Insert cursors
Using insert cursors with non-Informix databases.

Database cursors defined with "DECLARE cursor-name CURSOR FOR INSERT ..." are designed for IBM® Informix®, to optimize row insertion when a lot of data must be loaded in the table.

This is an IBM® Informix® specific feature. With non-Informix databases, insert cursors are emulated by executing the INSERT

```sql
DEFINE rec RECORD
  id    INTEGER,
  name  CHAR(100)
END RECORD,
DECLARE c1 CURSOR FOR INSERT INTO customer VALUES (?,?)
BEGIN WORK
```
Insert cursors are an IBM® Informix® specific feature. The IBM® Informix® insert cursors buffers the provided rows and flushes blocks of rows into the database after a given number of rows, or when the program explicitly executes a FLUSH or CLOSE. In case of errors, for example when inserting a character string value for a numeric column, the SQL error is returned at "flush time" with Informix®.

With non-Informix databases, the rows are not buffered: insert cursors are emulated in db drivers by executing the INSERT statement on every PUT instruction. As result, this can lead to poor performances, and SQL errors can be returned earlier at PUT time.

Note that the LOAD instruction is based on an insert cursor. The same performance issue applies to the LOAD instruction when using a non-Informix database.

If you need to feed your database with a lot of data, coming for example from external sources, we recommend to use database vendor specific tools to load the data. This option is much more efficient than using a Genero program to load data.

Table 179: INSERT cursors by database brands

<table>
<thead>
<tr>
<th>Database Server Type</th>
<th>INSERT cursor support</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM® DB2® LUW</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>IBM® Informix®</td>
<td>Yes, native SQL feature</td>
</tr>
<tr>
<td>IBM® Netezza</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>Microsoft™ SQL Server</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>Oracle® MySQL / MariadDB</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>Oracle® Database Server</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>SAP® ASE</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>SAP HANA®</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>SQLite</td>
<td>Emulated, see details</td>
</tr>
</tbody>
</table>

String literals in SQL statements

Single quotes is the standard for delimiting string literals in SQL.

Some database servers like IBM® Informix® allow single and double quoted string literals in SQL statements, both are equivalent:

```sql
SELECT COUNT(*) FROM table
WHERE col1 = "abc"'def"ghi"
AND col1 = 'abc''def"ghi"
```

Most database servers do not support this specific feature.
Table 180: Database servers support of double-quoted string literals

<table>
<thead>
<tr>
<th>Database Server Type</th>
<th>Double quoted string literals</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM® DB2® LUW</td>
<td>No</td>
</tr>
<tr>
<td>IBM® Informix®</td>
<td>Yes</td>
</tr>
<tr>
<td>Microsoft™ SQL Server</td>
<td>Yes</td>
</tr>
<tr>
<td>Oracle® MySQL / MariadbDB</td>
<td>No</td>
</tr>
<tr>
<td>Oracle Database Server</td>
<td>No</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>No</td>
</tr>
<tr>
<td>SAP® ASE</td>
<td>No</td>
</tr>
<tr>
<td>SAP HANA®</td>
<td>No</td>
</tr>
<tr>
<td>SQLite</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The ANSI SQL standards define doubles quotes as database object names delimiters, while single quotes are dedicated to string literals:

CREATE TABLE "my table" ( "column 1" CHAR(10) )
SELECT COUNT(*) FROM "my table" WHERE "column 1" = 'abc'

If you want to write a single quote character inside a string literal, you must write 2 single quotes:

... WHERE comment = 'John''s house'

When writing static SQL in your programs, the double quoted string literals as converted to ANSI single quoted string literals by the fglcomp compiler. However, dynamic SQL statements are not parsed by the compiler and therefore need to use single quoted string literals.

We recommend that you always use single quotes for string literals and, if needed, double quotes for database object names.

**Related concepts**

- **Static SQL statements** on page 694
  Describes static SQL statements supported in the language.

- **Dynamic SQL management** on page 709
  Explains how to execute and manage SQL statements at runtime.

**Date/time literals in SQL statements**

Good practices for date and time handling in SQL.

**Date and time strings in SQL Statements**

IBM® Informix® allows you to specify date and time values as a quoted character string in a specific format, depending upon DBDATE and GLS environment variables. For example, if DBDATE=DMY4, the following statement specifies a valid DATE represented by a string literal:

SELECT COUNT(*) FROM table WHERE date_col = '24/12/2005'

Other database servers do support date/time literals as quoted character strings, but the date/time format specification is quite different. The parameter to specify the date/time format can be a database parameter, an environment variable, or a session option.
In order to write portable SQL, use SQL parameters instead of string literals for date-time values:

```
DEFINE cnt INTEGER
DEFINE adate DATE
LET adate = MDY(12,24,2005)
SELECT COUNT(*) INTO cnt FROM table
    WHERE date_col = adate
```

Or, when using dynamic SQL:

```
DEFINE cnt INTEGER
DEFINE adate DATE
LET adate = MDY(12,24,2005)
PREPARE s1 FROM "SELECT COUNT(*) FROM table WHERE date_col = ?"
EXECUTE s1 USING adate INTO cnt
```

Similarly, when fetching rows from the database server into program variables, IBM® Informix® allows string literals (representing date values in DBDATE format) to be fetched into DATE variables:

```
DEFINE adate DATE
SELECT '24/12/2005' INTO adate FROM ...
```

With other database servers, consider casting the original date string to a real date value, to avoid any conversion issue at FETCH time.

As a general rule, always store and handle date values in DATE columns and variables, on both db server or program side.

**Date-time literals**

IBM® Informix® DATETIME and INTERVAL literals are not converted automatically by the SQL translator of the database driver:

```
SELECT COUNT(*) FROM order WHERE ord_when > DATETIME (1999-10-12) YEAR TO DAY
```

Check your code, to detect where you are using such expressions in the SQL statements, and use an SQL parameter instead.

**Informix-specific date/time keywords**

SQL statements using expressions such as TODAY, CURRENT, and EXTEND are specific to Informix® SQL.

Database drivers try to translate date/time constant expressions to native SQL syntax, but this is only provided to simplify migration.

Date/time expression translation can be controlled with the following FGLPROFILE entries:

```
dbi.database.dsname.ifxemul.today = [true | false]
dbi.database.dsname.ifxemul.current = [true | false]
dbi.database.dsname.ifxemul.extend = [true | false]
```

**Important:** To ease migration to a new database type, Informix-specific expressions such as TODAY, CURRENT and EXTEND are converted to native date/time expressions. However, the date/time returned by the native SQL function may use a different timezone / daylight saving time convention.

Check your code, to detect where you are using TODAY/CURRENT/EXTEND expressions in the SQL statements, and consider using SQL parameters with program variables assigned with the TODAY/CURRENT/EXTEND instruction of Genero BDL.
Date-time expressions with parameters

Date-time arithmetic expressions using SQL parameters (USING variables) are not portable.

For example:

```sql
PREPARE s1 FROM "SELECT ... WHERE datecol < ? + 1"
```

Might generate an error with non-Informix databases.

**DATEs as a number of days**

IBM® Informix® can automatically convert integers to a DATE values, as a number of days since 12/31/1899 (1 = 01/01/1900). This is however not supported by other database engines.

Check your code, to detect where you are using integers with DATE columns.

**Related concepts**

DBDATE on page 267

Defines the default display and input format for DATE values.

**Naming database objects**

**SQL identifiers syntax**

Database object naming conventions are different for each database engine.

The table below describes the naming conventions for database objects (tables, sequences, stored procedures):

<table>
<thead>
<tr>
<th>Database Server Type</th>
<th>Naming Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM® DB2® LUW</td>
<td><code>[[database.].owner.].identifier</code></td>
</tr>
<tr>
<td>IBM® Informix®</td>
<td><code>[database@dbservername]:[owner.].identifier</code></td>
</tr>
<tr>
<td>Microsoft™ SQL Server</td>
<td><code>[[server.].database.].owner.].identifier</code></td>
</tr>
<tr>
<td>Oracle® MySQL / Mariadb</td>
<td><code>[[database.].identifier</code></td>
</tr>
<tr>
<td>Oracle® Database Server</td>
<td><code>[[schema.].identifier[@database-link]]</code></td>
</tr>
<tr>
<td>PostgreSQL</td>
<td><code>[[database.].schema.].identifier</code></td>
</tr>
<tr>
<td>SAP® ASE</td>
<td><code>[[database.].identifier</code></td>
</tr>
<tr>
<td>SAP HANA®</td>
<td><code>[[database.].schema.].identifier</code></td>
</tr>
<tr>
<td>SQLite</td>
<td><code>[[database.].identifier</code></td>
</tr>
</tbody>
</table>

As a general rule, use simple unqualified database object names and use database vendor specific configuration setting or SQL instruction to point to the expected schema.
When possible, Genero provides FGLPROFILE entries to define the database schema to be selected after connecting to the database server, with:

```sql
dbi.database.dbname.drivercode.schema = "schema-name"
```

For more details, see Database type specific parameters in FGLPROFILE on page 672.

**SQL identifiers case-sensitivity**

Handling case-sensitivity with different database engines.

In general, database engines use case-sensitive object identifiers. In most cases, when you do not specify identifiers in double quotes, the SQL parser automatically converts names to uppercase or lowercase. With this rule, identifiers will match when the objects are also created without double quoted identifiers.

For example, when creating the following table in ORACLE DB:

```sql
CREATE TABLE Customer ( cust_ID INTEGER )
```

With the above SQL, ORACLE creates a table named "CUSTOMER" with a "CUST_ID" column. Since DB object identifiers are converted to uppercase when executing SQL statements, the next SELECT statement will match the "CUSTOMER" table name and "CUST_ID" column name:

```sql
SELECT * FROM customer WHERE CUST_ID = 219
```

However, the next statements will fail:

```sql
SELECT * FROM "customer" WHERE "CUST_ID" = 219
```

The next table shows the behavior of each database engine regarding case sensitivity and double quoted identifiers:

**Table 182: Database server support of case sensitivity and double-quoted identifiers**

<table>
<thead>
<tr>
<th>Database Server Type</th>
<th>Un-quoted names</th>
<th>Double-quoted names</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM® DB2® LUW</td>
<td>Converts to uppercase</td>
<td>Case sensitive</td>
</tr>
<tr>
<td>IBM® Informix® (1)</td>
<td>Converts to lowercase</td>
<td>Syntax disallowed (non-ANSI mode)</td>
</tr>
<tr>
<td>Microsoft™ SQL Server (2)</td>
<td>Not converted, kept as is</td>
<td>Case sensitive</td>
</tr>
<tr>
<td>Oracle® MySQL / Mariadb</td>
<td>Not converted, kept as is</td>
<td>Syntax disallowed</td>
</tr>
<tr>
<td>Oracle® Database Server</td>
<td>Converts to uppercase</td>
<td>Uppercase</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>Converts to lowercase</td>
<td>Lowercase</td>
</tr>
<tr>
<td>SAP® ASE</td>
<td>Converts to lowercase</td>
<td>Lowercase</td>
</tr>
<tr>
<td>SAP HANA®</td>
<td>Converts to uppercase</td>
<td>Case sensitive</td>
</tr>
<tr>
<td>SQLite</td>
<td>Not converted, kept as is</td>
<td>Case insensitive</td>
</tr>
</tbody>
</table>

(1) If not ANSI database mode.

(2) When case-sensitive charset/collation used.

**Important:**

When DB object identifiers are case sensitive and are not converted to uppercase or lowercase when not using double-quotes, it is possible to create two tables with a similar name (mind the uppercase C in the table name):

```sql
CREATE TABLE customer ( cust_id INTEGER )  -- first table
CREATE TABLE Customer ( cust_id INTEGER )  -- second table
```
For maximum portability, define database object names in lowercase and do not use double quoted identifiers.

**Note:** When using Static SQL statements, the `fglcomp` compiler converts table and column names to lowercase when not using single quotes or double quotes. Check the output of `fglcomp -s` option.

**SQL identifiers size**

Avoid using long database object names.

The maximum size of a table or column name depends on the database server type. Some database engines allow very large names (256c), while others support only short names (30c max). Therefore, using short names is required for writing portable SQL. Short names also simplify SQL programs.

We recommend that you use simple and short (<30c) database object names, without double quotes and without a schema/owner prefix:

```
CREATE TABLE customer ( cust_id INTEGER )
SELECT customer.cust_id FROM table
```

You may need to set the database schema after connection, so that the current database user can see the application tables without specifying the owner/schema prefix each time.

**Tip:** Even if all database engines do not required unique column names for all tables, define column names with a small table prefix (for example, "cust_id" in the "customer" table).

**LOAD and UNLOAD instructions**

The `LOAD` and `UNLOAD` instructions can produce different data formats depending on the database server type.

When executing the `LOAD` and `UNLOAD` instruction from a BDL program, SQL commands are not sent as is to the database server.

In fact, the Genero BDL runtime system implements the `LOAD` and `UNLOAD` instructions, using basic INSERT (for `LOAD`) or SELECT (for `UNLOAD`) SQL commands.

With this implementation, the `LOAD` and `UNLOAD` instruction can be supported with various database servers.

However, `LOAD` and `UNLOAD` require the description of the column types in order to work.

Since each database has its own set of SQL data types, you must pay attention to the BDL type that results from the native database column type.

Depending on the native data type, data formatting may be different from Informix®.

For example, when using Oracle DB, if the table contains a `DATE` column, the `LOAD` and `UNLOAD` instruction will use the date/time format `YYYY-MM-DD hh:mm:ss`. Since the native Oracle `DATE` type can be used to store both Informix® `DATE` or `DATETIME YEAR TO SECOND` values, and as `LOAD` / `UNLOAD` need to make date/time to string conversions when reading from or writing to unload files, it is not possible to select one of the Informix/BDL native date type formats.

**Note:** When using LOAD/UNLOAD, if the target database server provides the exact equivalent date types as the native Informix/BDL `DATE`, the date values will use the `DBDATE` format setting.

### Table 183: Database server support of LOAD and UNLOAD

<table>
<thead>
<tr>
<th>Database Server Type</th>
<th>LOAD/UNLOAD support</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM® DB2® LUW</td>
<td>Yes, see details</td>
</tr>
<tr>
<td>IBM® Informix®</td>
<td>Yes, native SQL feature</td>
</tr>
<tr>
<td>IBM® Netezza</td>
<td>Yes, see details</td>
</tr>
<tr>
<td>Microsoft™ SQL Server</td>
<td>Yes, with some limitations</td>
</tr>
<tr>
<td>Oracle® MySQL / MariadbDB</td>
<td>Yes, see details</td>
</tr>
<tr>
<td>Oracle Database Server</td>
<td>Yes, with some limitations</td>
</tr>
<tr>
<td>Database Server Type</td>
<td>LOAD/UNLOAD support</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>Yes, see details</td>
</tr>
<tr>
<td>SAP® ASE</td>
<td>Yes, see details</td>
</tr>
<tr>
<td>SAP HANA®</td>
<td>Yes, see details</td>
</tr>
<tr>
<td>SQLite</td>
<td>Yes, see details</td>
</tr>
</tbody>
</table>

To export table rows from a database brand that is different from the target database, format the data, to prepare for the native column types of the destination database. For example to unload from Informix and load into Oracle DB, format date/time columns:

```sql
FUNCTION ifx_type_to_expr_for_oracle(tn STRING) RETURNS STRING
CASE
  WHEN tn == "DATE"
    RETURN "TO_CHAR(%1,'YYYY-MM-DD'TH00:00:00')"
  WHEN tn == "DATETIME YEAR TO YEAR"
    RETURN "%1" || '-01-01 00:00:00'
  WHEN tn == "DATETIME YEAR TO MONTH"
    RETURN "%1" || '-01 00:00:00'
  WHEN tn == "DATETIME YEAR TO DAY"
    RETURN "%1" || ' 00:00:00'
  WHEN tn == "DATETIME YEAR TO MINUTE"
    RETURN "%1" || ':00'
  WHEN tn == "DATETIME HOUR TO SECOND"
    RETURN '1900-01-01 '||%1
  WHEN tn == "DATETIME HOUR TO MINUTE"
    RETURN '1900-01-01 '||%1||':00'
  WHEN tn MATCHES "DATETIME HOUR TO FRACTION*"
    RETURN '1900-01-01 '||%1
  WHEN tn MATCHES "INTERVAL YEAR* TO YEAR"
    RETURN "%1-00"%
  WHEN tn MATCHES "INTERVAL YEAR* TO MONTH"
    RETURN "%1"
  WHEN tn MATCHES "INTERVAL HOUR* TO MINUTE"
    RETURN "%1||':00'"
  WHEN tn MATCHES "INTERVAL HOUR* TO SECOND"
    RETURN "%1||%1"
  WHEN tn MATCHES "INTERVAL HOUR* TO FRACTION*"
    RETURN "%1||%1"
  OTHERWISE
    RETURN "%1"
END CASE
END FUNCTION

FUNCTION build_select(tn STRING) RETURNS STRING
DEFINE x INTEGER
DEFINE sqlcmd base.StringBuffer
DEFINE curs base.SqlHandle
LET curs = base.SqlHandle.create()
CALL curs.prepare("SELECT * FROM "||tn)
CALL curs.open()
LET sqlcmd = base.StringBuffer.create()
CALL sqlcmd.append("SELECT 
    FOR x=1 TO curs.getResultCount()
      IF x>1 THEN
        CALL sqlcmd.append( "", " ")
      END IF
      CALL sqlcmd.append( SFMT( ifx_type_to_expr(curs.getResultType(x)),
                                       curs.getResultName(x) ) )
    END FOR
    CALL sqlcmd.append( " FROM " || tn )
RETURN sqlcmd.toString()
END FUNCTION
```

Related concepts
SQL LOAD and UNLOAD on page 736
Describes the instructions to export/import information from/to a database.

**Temporary tables**

Syntax for temporary table creation is not unique across all database engines.

Not all database servers support temporary tables. The engines supporting this feature often provide it with a specific table creation statement:
<table>
<thead>
<tr>
<th>Database Server Type</th>
<th>Native temp table creation syntax</th>
<th>Temp table support</th>
</tr>
</thead>
</table>
| IBM® DB2® LUW              | DECLARE GLOBAL TEMPORARY TABLE tablename (col-defs)  
DECLARE GLOBAL TEMPORARY TABLE tablename AS (SELECT ...) | Emulated, see details            |
|                            | **Note:** DB2 Version 11.1 supports CREATE TEMP TABLE syntax for compatibility with Netezza.      |                                  |
| IBM® Informix®             | CREATE TEMP TABLE tablename (col-defs)  
SELECT ... INTO TEMP tablename | Yes, native SQL feature          |
| IBM® Netezza               | CREATE TEMP TABLE tablename (col-defs)  
SELECT ... INTO TEMP temptab FROM ... | Emulated, see details            |
| Microsoft™ SQL Server      | CREATE TABLE #tablename (col-defs)  
SELECT select-list INTO #tablename FROM ... | Emulated, see details            |
| Oracle® MySQL / MariaDB    | CREATE TEMPORARY TABLE tablename (col-defs)  
CREATE TEMPORARY TABLE tablename LIKE other-table | Emulated, see details            |
| Oracle® Database Server    | CREATE GLOBAL TEMPORARY TABLE tablename (col-defs)  
CREATE GLOBAL TEMPORARY TABLE tablename AS SELECT ... | Emulated, see details            |
|                            | or, since Oracle 18c:  
CREATE PRIVATE TEMPORARY TABLE tablename (col-defs)  
CREATE PRIVATE TEMPORARY TABLE tablename AS SELECT ... | Emulated, see details            |
| PostgreSQL                 | CREATE TEMP TABLE tablename (col-defs)  
SELECT select-list INTO TEMP tablename FROM ... | Emulated, see details            |
| SAP® ASE                   | CREATE TABLE #tablename (col-defs)  
SELECT select-list INTO #tablename FROM ... | Emulated, see details            |
| SAP HANA®                  | CREATE LOCAL TEMPORARY TABLE #tablename (col-defs) | Emulated, see details            |
| SQLite                     | CREATE TEMP TABLE tablename (col-defs) | Emulated, see details            |
The behavior and limitations of temporary tables varies with the type of database server. See database adaptation guides for more details.

**Important:**

Simple Informix-style SQL statement creating temporary tables can be converted to a native SQL equivalent instruction. However, complex SQL statements such as `SELECT . INTO TEMP` with subqueries may fail. In such cases, create a view from the complex query and then create the temp table from the view. Or, disable Informix emulation and use the native SQL syntax to create the temporary table (`EXECUTE IMMEDIATE "/* fglhint_no_ifxemul */ ..."`).

With Informix SQL, if the source table has a column defined as `SERIAL` or `BIGSERIAL`, a `SELECT ... INTO TEMP` will produce a new temp table with an auto-incremented serial column. With the `SELECT ... INTO TEMP` emulation for non-Informix databases, not using the native sequence generators (such as `IDENTITY` columns in SQL Server), the resulting temporary table will get a simple `INTEGER` or `BIGINT` column, instead of an auto-incremented column.

Consider reviewing programs using temporary tables, and adapt the code to create temporary tables with native SQL syntax.

**Outer joins**

Use standard ISO outer join syntax instead of the old IBM® Informix® `OUTER()` syntax.

Old IBM® Informix® SQL outer joins specified with the `OUTER` keyword in the `FROM` part are not standard:

```
SELECT * FROM master, OUTER ( detail )
WHERE master.mid = detail.mid
AND master.cdate IS NOT NULL
```

**Table 185: Informix OUTER JOIN syntax by database brand**

<table>
<thead>
<tr>
<th>Database Server Type</th>
<th>Informix OUTER join support</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM® DB2® LUW</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>IBM® Informix®</td>
<td>Yes, native SQL feature</td>
</tr>
<tr>
<td>IBM® Netezza</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>Microsoft® SQL Server</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>Oracle® MySQL / Mariadb</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>Oracle® Database Server</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>SAP® ASE</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>SAP HANA®</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>SQLite</td>
<td>Emulated, see details</td>
</tr>
</tbody>
</table>

Most recent database servers now support the standard ANSI outer join specification:

```
SELECT * FROM master LEFT OUTER JOIN detail ON (master.mid = detail.mid)
WHERE master.cdate IS NOT NULL
```

it is recommended that you use recent database servers and use ANSI outer joins only.
**Substring expressions**

Handle substrings expressions with different database engines.

Only IBM® Informix® supports substring specification with square brackets:

```
SELECT * FROM item WHERE item_code[1,4] = "XBFG"
```

This syntax is specific to Informix SQL. Other database server types provide a function that extracts substrings from a character string.

**Table 186: Substrings support by database server brand**

<table>
<thead>
<tr>
<th>Database Server Type</th>
<th>Substring function</th>
<th>col[x,y] support</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM® DB2® LUW</td>
<td>SUBSTR(expr,start,length)</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>IBM® Informix®</td>
<td>SUBSTR(expr,start,length)</td>
<td>Yes, native SQL feature</td>
</tr>
<tr>
<td>IBM® Netezza</td>
<td>SUBSTRING(expr,start,length)</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>Microsoft™ SQL Server</td>
<td>SUBSTRING(expr,start,length)</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>Oracle® MySQL / MariadDB</td>
<td>SUBSTR(expr,start,length)</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>Oracle® Database Server</td>
<td>SUBSTRING(expr,start,length)</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>SUBSTRING(expr FROM start FOR length)</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>SAP® ASE</td>
<td>SUBSTRING(expr,start,length)</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>SAP HANA®</td>
<td>SUBSTRING(expr,start,length)</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>SQLite</td>
<td>SUBSTR(expr,start,length)</td>
<td>Emulated, see details</td>
</tr>
</tbody>
</table>

Informix® allows you to update some parts of a CHAR and VARCHAR column by using the substring specification (UPDATE tab SET col[1,2] = 'ab'). This is not possible with other databases.

Review the SQL statements using substring expressions and use the database specific substring function.

You can also create your own SUBSTRING() user function in all databases that do not support this function, to have a common way to extract substrings. In Microsoft™ SQL Server, when you create a user function, you must specify the owner as prefix when using the function. Therefore, it is recommended that you create a SUBSTRING() user function instead of SUBSTR().

**Using ROWID columns**

Automatic ROWID columns is not a common database feature.

Rowids are implicit primary keys generated by the database engine. Not all database servers support rowids:
Table 187: Database server support of rowid

<table>
<thead>
<tr>
<th>Database Server Type</th>
<th>Rowid keyword</th>
<th>Rowid type</th>
<th>ROWID support</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM® DB2® LUW</td>
<td>No</td>
<td>N/A</td>
<td>Yes, see details</td>
</tr>
<tr>
<td>IBM® Informix®</td>
<td>ROWID</td>
<td>INTEGER</td>
<td>Yes, native SQL feature</td>
</tr>
<tr>
<td>IBM® Netezza</td>
<td>No</td>
<td>N/A</td>
<td>Yes, see details</td>
</tr>
<tr>
<td>Microsoft® SQL Server (2)</td>
<td>No</td>
<td>N/A</td>
<td>No, see details</td>
</tr>
<tr>
<td>Oracle® MySQL / Mariadb</td>
<td>No</td>
<td>N/A</td>
<td>No, see details</td>
</tr>
<tr>
<td>Oracle® Database Server</td>
<td>ROWID</td>
<td>CHAR(18)</td>
<td>Yes, see details</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>OID</td>
<td>internal type</td>
<td>Yes with pre-12 versions. No since version 12, see details</td>
</tr>
<tr>
<td>SAP® ASE</td>
<td>No</td>
<td>N/A</td>
<td>No, see details</td>
</tr>
<tr>
<td>SAP HANA®</td>
<td>No</td>
<td>N/A</td>
<td>No, see details</td>
</tr>
<tr>
<td>SQLite</td>
<td>ROWID</td>
<td>BIGINT</td>
<td>Yes, see details</td>
</tr>
</tbody>
</table>

Informix® fills the SQLCA.SQLERRD[6] register with the ROWID of the last updated row. This register is an INTEGER and cannot be filled with rowids having CHAR(size) type.

Search for ROWID and SQLCA.SQLERRD[6] in your code and review the code to remove the usage of rowids.

**Important:** Instead of using ROWID pseudo columns, consider to define PRIMARY KEY constraints in all your database tables, and use the primary key values instead of rowids to identify rows.

**Related concepts**

- The SQLCA diagnostic record on page 594
- The SQLCA variable is a predefined record containing SQL statement execution information.

**MATCHES and LIKE operators**

Use the standard LIKE operator instead of the MATCHES operator.

The MATCHES operator is specific to IBM® Informix® SQL, it allows to compare a character string column to a search pattern:

```sql
SELECT * FROM customer WHERE customer_name MATCHES "A*"
```

**Note:** The Genero language supports a MATCHES operator. Do not confuse the language MATCHES operator (used in BDL instructions such as IF custname MATCHES "S*"); with the SQL MATCHES operator (used in SQL statements). There is no problem in using the MATCHES operator of BDL.

The standard SQL operator for pattern search is LIKE:

```sql
SELECT * FROM customer WHERE customer_name LIKE "A%"
```

When using a non-Informix driver, the MATCHES expressions using a string constant are replaced by a LIKE expression.

**Important:** Only MATCHES expressions with a string constant can be converted to LIKE expressions, if the MATCHES uses a ? SQL parameter place holder, no translation is done.
### Table 188: SQL MATCHES operator support by database brand

<table>
<thead>
<tr>
<th>Database Server Type</th>
<th>SQL MATCHES support</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM® DB2® LUW</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>IBM® Informix®</td>
<td>Yes, native SQL feature</td>
</tr>
<tr>
<td>IBM® Netezza</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>Microsoft™ SQL Server</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>Oracle® MySQL / Mariadb</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>Oracle® Database Server</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>SAP® ASE</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>SAP HANA®</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>SQLite</td>
<td>Emulated, see details</td>
</tr>
</tbody>
</table>

For maximum portability, replace SQL MATCHES expressions by LIKE expression. MATCHES uses * and ? as wildcards. The equivalent wildcards in the LIKE operator are % and _.

**Important:** MATCHES character ranges such as [a-z] cannot be converted for the LIKE operator.

Pay attention to blank padding semantics of the target database when using a program variable in static SQL or a ? SQL parameter place holder after the LIKE operator: If the program variable is defined as a CHAR(N), it is filled by the runtime system with trailing blanks, in order to have a size of N, and the actual pattern for the LIKE operator will contain trailing blanks. For example, when a CHAR(10) variable is assigned with "ABC%", it will in fact contain "ABC% " (with 6 additional blanks). When used as SQL parameter for a LIKE expression, the database server will search for column values matching "ABC"+ some characters + 6 blanks. To avoid this, use a VARCHAR(N) data type instead of CHAR(N) to hold LIKE patterns.

Some database engines have specific semantics for the LIKE operator, especially when using CHAR(N) data types. For example, with Oracle® DB, the expression custname LIKE '%h': If custname is defined as CHAR(30), Oracle® will only find the rows when the custname values end with a 'h' at the last character position (30). Values such as 'Smith' will not match. Similarly, when doing custname LIKE 'ab_', rows where the column type is CHAR(N>3), values such as 'abc' will not match in Oracle®, IBM® DB2® and PostgreSQL, because of the significant trailing blanks.

As a general advice, use the VARCHAR type for variable string data, and leave CHAR usage for fixed-length character string data such as codes.

**Note:** PostgreSQL provides the SIMILAR TO operator, allowing [start-end] character range specification as in MATCHES.

**Related concepts**

MATCHES on page 341

The MATCHES operator returns TRUE if a string matches a given mask.

**GROUP BY clause**

Some databases allow you to specify a column index in the GROUP BY clause:

```sql
SELECT a, b, sum(c) FROM table GROUP BY 1,2
```

This is not possible with all database servers:
Table 189: Database server support of GROUP BY column index

<table>
<thead>
<tr>
<th>Database Server Type</th>
<th>GROUP BY colindex, ... ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM® DB2® LUW</td>
<td>No</td>
</tr>
<tr>
<td>IBM® Informix®</td>
<td>Yes</td>
</tr>
<tr>
<td>IBM® Netezza</td>
<td>No</td>
</tr>
<tr>
<td>Microsoft™ SQL Server</td>
<td>No</td>
</tr>
<tr>
<td>Oracle® MySQL / Mariadb</td>
<td>Yes</td>
</tr>
<tr>
<td>Oracle® Database Server</td>
<td>No</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>Yes</td>
</tr>
<tr>
<td>SAP® ASE</td>
<td>No</td>
</tr>
<tr>
<td>SAP HANA®</td>
<td>No</td>
</tr>
<tr>
<td>SQLite</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Search for GROUP BY in your SQL statements and use explicit column names.

**The LENGTH() function in SQL**

The semantics of the LENGTH() SQL function differs according to the database engine.

The SQL LENGTH() function must be used with care: Each database server has different semantics for this function, regarding length and trailing blanks handling.

**Note:** The language provides a LENGTH built-in function which is part of the runtime system. Do not confuse this with the SQL LENGTH() function, used in SQL statements. The LENGTH() function of the language returns zero when the string expression is NULL.
Table 190: SQL LENGTH() support by database brand

<table>
<thead>
<tr>
<th>Database Server Type</th>
<th>Function name</th>
<th>Counting unit</th>
<th>Significant trailing blanks for CHAR() columns</th>
<th>Return value when NULL</th>
<th>Related topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM® DB2® LUW</td>
<td>LENGTH(expr)</td>
<td>Octets</td>
<td>Yes</td>
<td>NULL</td>
<td>See details</td>
</tr>
<tr>
<td>IBM® Informix®</td>
<td>LENGTH(expr)</td>
<td>Octets</td>
<td>No</td>
<td>NULL</td>
<td>Native SQL feature</td>
</tr>
<tr>
<td>IBM® Netezza</td>
<td>LENGTH(expr)</td>
<td>Characters</td>
<td>No</td>
<td>NULL</td>
<td>See details</td>
</tr>
<tr>
<td>Microsoft™ SQL Server</td>
<td>LEN(expr)</td>
<td>Characters</td>
<td>No</td>
<td>NULL</td>
<td>See details</td>
</tr>
<tr>
<td>Oracle® MySQL / Mariadb</td>
<td>LENGTH(expr)</td>
<td>Characters</td>
<td>No</td>
<td>NULL</td>
<td>See details</td>
</tr>
<tr>
<td>Oracle® Database Server</td>
<td>LENGTH(expr)</td>
<td>Characters</td>
<td>Yes</td>
<td>NULL</td>
<td>See details</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>LENGTH(expr)</td>
<td>Characters</td>
<td>No</td>
<td>NULL</td>
<td>See details</td>
</tr>
<tr>
<td>SAP® ASE</td>
<td>LEN(expr)</td>
<td>Characters</td>
<td>No</td>
<td>NULL</td>
<td>See details</td>
</tr>
<tr>
<td>SAP HANA®</td>
<td>LENGTH(expr)</td>
<td>Characters</td>
<td>No</td>
<td>NULL</td>
<td>See details</td>
</tr>
<tr>
<td>SQLite</td>
<td>LENGTH(expr)</td>
<td>Characters</td>
<td>Yes</td>
<td>NULL</td>
<td>See details</td>
</tr>
</tbody>
</table>

Search for LENGTH() usage in your SQL statements and review the code of the database-specific function.

**Tip:** Create a user-defined SQL function that implements the Informix SQL LENGTH() function. For example, with Oracle PL/SQL:

```sql
CREATE OR REPLACE FUNCTION vlength(
    value IN VARCHAR2
) RETURN INTEGER
AUTHID CURRENT_USER
IS
BEGIN
    RETURN NVL(LENGTH(RTRIM(value)),0);
END;
/
```

**Large Objects (LOBs) data types**

Use **TEXT** and **BYTE** FGL types to store database character and binary large objects.

Genero BDL provides the **TEXT** data type to store character LOBs and the **BYTE** data type to store binary LOBs.
The TEXT and BYTE types can be used to handle large objects data from various database brands. However, some database types are not supported.

### Table 191: Large OBjects support with TEXT/BYTE

<table>
<thead>
<tr>
<th>Database Server Type</th>
<th>LOBs support with TEXT/BYTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM® DB2® LUW</td>
<td>Yes, see details</td>
</tr>
<tr>
<td>IBM® Informix®</td>
<td>Yes, native SQL feature</td>
</tr>
<tr>
<td>IBM® Netezza</td>
<td>N/A, see details</td>
</tr>
<tr>
<td>Microsoft™ SQL Server</td>
<td>Yes, see details</td>
</tr>
<tr>
<td>Oracle® MySQL / Mariadb</td>
<td>Yes, see details</td>
</tr>
<tr>
<td>Oracle® Database Server</td>
<td>Yes, see details</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>Yes, see details</td>
</tr>
<tr>
<td>SAP® ASE</td>
<td>Yes (with limitations), see details</td>
</tr>
<tr>
<td>SAP HANA®</td>
<td>Yes, see details</td>
</tr>
<tr>
<td>SQLite</td>
<td>Yes, see details</td>
</tr>
</tbody>
</table>

**Stored procedures**

Executing stored procedures with different database engine types.

Stored procedures execution needs to be addressed specifically depending on the database type. There are different ways to execute a stored procedure. This section describes how to execute stored procedures on the supported database engines.

**Tip:** In order to write reusable code, it is recommended that you encapsulate each stored procedure execution in a FUNCTION performing database-specific SQL based on a global database type variable. The program function would just take the input parameters and return the output parameters of the stored procedure, hiding database-specific execution steps from the caller.

**Specifying input and output parameters**

Input and output parameters can be specified in SQL statement execution to pass and return values to/from stored procedures, depending on the database type:

```sql
EXECUTE stmt USING param1 IN, param2 INOUT, param3 INOUT
```

**Related concepts**

**EXECUTE (SQL statement)** on page 711
This instruction runs an SQL statement previously prepared.

**Stored procedures returning a result set**

With some database servers it is possible to execute stored procedures that produce a result set, and fetch the rows as normal SELECT statements, by using DECLARE, OPEN, FETCH. Some databases can return multiple result sets and cursor handles declared in a stored procedure as output parameters, but Genero supports only unique and anonymous result sets. See the examples.

**Calling stored procedures with supported databases**

See ODI Adaptation Guide for all details to call stored procedures from your programs:

- Calling stored procedures with IBM® Informix® IDS
- Calling stored procedures with IBM® DB2® LUW
- Calling stored procedures with Microsoft™ SQL Server
- Calling stored procedures with Oracle® MySQL / MariaDB
• Calling stored procedures with Oracle® Database
• Calling stored procedures with PostgreSQL
• Calling stored procedures with SAP® ASE
• Calling stored procedures with SAP HANA®

**Row limiting clause (SELECT)**
How to use the right clause to limit the number of rows produced by a SELECT statement?

IBM® Informix® SQL supports the SKIP and FIRST/LIMIT keywords to limit the number of rows of a result set. For example:

```
SELECT SKIP 10 FIRST 20 customer.* FROM customer ... ORDER BY cust_name
```

**Note:** It is strongly recommended to use an ORDER BY clause when limiting the result set rows.

The above Informix SQL syntax is not portable.

Recent database engines support the row limiting clause syntax defined by the SQL standard:

```
SELECT ... OFFSET n ROWS FETCH FIRST m ROWS ONLY
```

This should be the preferred syntax to be used, if all target database types support this SELECT clause.

The ODI database drivers can convert the Informix SQL SKIP/FIRST row limiting clause to a native SQL equivalent, if the row limiting clause parameters are simple integer literals (the clause is not translated when using SQL parameters / program variables).

**Important:** In addition to the SKIP/FIRST clause of the projection clause, Informix SQL supports also a LIMIT clause after the ORDER BY clause:

```
SELECT customer.* FROM customer ... ORDER BY cust_name LIMIT 10
```

This Informix SQL syntax construction is not converted by the ODI drivers. To benefit from the conversion, review the code to use the Informix SQL SKIP/FIRST clause instead.

<table>
<thead>
<tr>
<th>Database Server Type</th>
<th>Native equivalent</th>
<th>Informix emulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM® DB2® LUW 11.1+</td>
<td>SELECT ... ORDER BY ... [OFFSET n ROWS] FETCH FIRST m ROWS ONLY</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>IBM® Informix® 12+</td>
<td>SELECT [SKIP n] FIRST m ...</td>
<td>Yes, native SQL feature</td>
</tr>
<tr>
<td>IBM® Netezza 7+</td>
<td>SELECT ... ORDER BY ... [OFFSET n] LIMIT m [OFFSET n]</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>Microsoft™ SQL Server 2012+</td>
<td>SELECT ... ORDER BY ... [OFFSET n ROWS] FETCH FIRST m ROWS ONLY</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>Oracle® MySQL 5.5-8.0 / Mariadb 10.x</td>
<td>SELECT ... ORDER BY ... [OFFSET n] LIMIT m [OFFSET n]</td>
<td>Emulated, see details</td>
</tr>
<tr>
<td>Oracle® Database Server 12c+</td>
<td>SELECT ... ORDER BY ... [OFFSET n ROWS] FETCH FIRST m ROWS ONLY</td>
<td>Emulated, see details</td>
</tr>
</tbody>
</table>
SQL support

<table>
<thead>
<tr>
<th>Database Server Type</th>
<th>Native equivalent</th>
<th>Informix emulation</th>
</tr>
</thead>
</table>
| PostgreSQL 9.4+      | SELECT ... ORDER BY ...  
                        | OFFSET n ROWS] FETCH FIRST m ROWS ONLY | Emulated, see details |
| SAP® ASE 16          | SELECT TOP (m) ... | Not supported, see details |
| SAP HANA® 2.0+       | SELECT ... ORDER BY ...  
                        | LIMIT m [OFFSET n] | Emulated, see details |
| SQLite 3.8+          | SELECT ... ORDER BY ...  
                        | LIMIT m [OFFSET n] | Emulated, see details |

SQL performance

Execution time can be greatly improved, when using SQL instructions in the right way.

Performance with dynamic SQL

Comparing static SQL statements and dynamic SQL statements used in a loop.

Although SQL statements can be directly specified in the program source as a part of the language as static SQL, it is sometimes more efficient to use dynamic SQL instead, when you are executing SQL statements repeatedly.

Dynamic SQL allows you to PREPARE the SQL statements once and EXECUTE N times, improving performance.

Note however that implementing prepared statements with dynamic SQL has a cost in terms of database resources and code readability: When a simple static SQL statement is executed, database client and server resources are allocated for the statement and are reused for the next Static SQL statement. With dynamic SQL, you define a statement handle and allocate database resources that last until you FREE the handle. Regarding code readability, static SQL statements can be written directly in the source code (as another language statement), while Dynamic SQL uses several instructions and takes the SQL text as a string expression. Additionally, static SQL statements are parsed at compile time so you can detect syntax errors in the SQL text before executing the programs.

Therefore, dynamic SQL should only be used if the SQL statement is created at runtime (with a where part generated by a CONSTRUCT for example) or if the execution time is too long with static SQL (assuming that it’s only a statement preparation issue).

To perform static SQL statement execution, the database interface must use the basic API functions provided by the database client. These are usually equivalent to the PREPARE and EXECUTE instructions. So when you write a static SQL statement in your program, it is actually converted to a PREPARE + EXECUTE behind the scene.

For example, the following code:

```sql
FOR n=1 TO 100
  INSERT INTO tab VALUES ( n, c )
END FOR
```

is actually equivalent to:

```sql
FOR n=1 TO 100
  PREPARE s FROM "INSERT INTO tab VALUES ( ?, ? )"
  EXECUTE s USING n, c
END FOR
```

To improve the performance of the preceding code, use a PREPARE instruction before the loop and put an EXECUTE instruction inside the loop:

```sql
PREPARE s FROM "INSERT INTO tab VALUES ( ?, ? )"
FOR n=1 TO 100
  EXECUTE s USING n, c
END FOR
```
**Performance with transactions**
Commit database changes by blocks of transaction speeds performance with some database servers.

To mimic the IBM® Informix® auto-commit behavior with an ANSI compliant RDBMS like Oracle® or DB2®, the database driver must perform an implicit commit after each statement execution, if the SQL statement is not inside a transaction block. This generates unnecessary database operations and can slow down big loops. To avoid this implicit commit, you can control the transaction with `BEGIN WORK / COMMIT WORK` around the code containing a lot of SQL statement execution.

This technique is especially recommended with SQLite, because the SQLite database library performs a lot of operations during a commit.

For example, the following loop will generate 2000 basic SQL operations (1000 inserts plus 1000 commits):

```sql
PREPARE s FROM "INSERT INTO tab VALUES ( ?, ? )"
FOR n=1 TO 100
  EXECUTE s USING n, c   -- Generates implicit COMMIT
END FOR
```

You can improve performance if you put a transaction block around the loop:

```sql
PREPARE s FROM "INSERT INTO tab VALUES ( ?, ? )"
BEGIN WORK
FOR n=1 TO 100
  EXECUTE s USING n, c   -- In transaction -> no implicit COMMIT
END FOR
COMMIT WORK
```

With this code, only 1001 basic SQL operations will be executed (1000 inserts plus 1 commit).

However, you must take care when generating large transactions because all modifications are registered in transaction logs. This can result in a lack of database server resources ("transaction too long" errors, for example) when the number of operations is very big. If the SQL operation does not require a unique transaction for database consistency reasons, you can split the operation into several transactions, as in this example:

```sql
PREPARE s FROM "INSERT INTO tab VALUES ( ?, ? )"
BEGIN WORK
FOR n=1 TO 100
  IF n MOD 10 == 0 THEN
    COMMIT WORK
    BEGIN WORK
  END IF
  EXECUTE s USING n, c   -- In transaction -> no implicit COMMIT
END FOR
COMMIT WORK
```

Note that the `LOAD` instruction automatically starts a transaction, if not yet initiated. Therefore there is no need to enclose the `LOAD` statement within a `BEGIN WORK / COMMIT WORK`, except if other SQL statements are part of the transaction and need to be processed as a single atomic database change.

**Related concepts**
- [Database transactions](#) on page 688
  Database transaction concepts and handling.

**Avoiding long transactions**
Long transactions consume resources and decrease concurrent data access.

Old applications based on IBM® Informix® database without transaction logging might perform long running SQL modifications.
With recent database engines, using huge transactions can lead to errors because of transaction log buffer overflow. For example, if a table holds many rows, a "DELETE FROM table" might produce a "snapshot too old" error in Oracle, if the rollback segments are too small.

Therefore, you must avoid long transactions when connected to a database using transactions:

• keep transactions as short as possible.
• access the least amount of data possible while in a transaction.
• split a long transaction into many short transactions. Use a loop to handle each block.
• to delete all rows from a table use the "TRUNCATE TABLE" instruction instead of "DELETE FROM" (Not for all vendors).
• In the end, increase the size of the transaction log to avoid filling it up.

Related concepts
Database transactions on page 688
Database transaction concepts and handling.
Performance with transactions on page 652
Commit database changes by blocks of transaction speeds performance with some database servers.

Declaring prepared statements
Optimize prepared cursor statements by using the FROM clause of DECLARE CURSOR.

Line 2 of this example shows a cursor declared with a prepared statement:

```
PREPARE s FROM "SELECT * FROM table WHERE ", condition
DECLARE c CURSOR FOR s
```

While this has no performance impact with IBM® Informix® database drivers, it can become a bottleneck when using non-IBM Informix database servers:

Statement preparation consumes a lot of memory and processor resources. Declaring a cursor with a prepared statement is a native IBM® Informix® feature, which consumes only one real statement preparation. Non-IBM Informix database servers do not support this feature, so the statement is prepared twice (once for the PREPARE, and once for the DECLARE). When used in a big loop, this code can cause performance problems.

To optimize the code, use the FROM clause in the DECLARE statement:

```
DECLARE c CURSOR FROM "SELECT * FROM table WHERE " || condition
```

By using this solution only one statement preparation will be done by the database server.

Note: This performance problem does not occur with DECLARE statements using static SQL.

Related concepts
DECLARE (result set cursor) on page 717
Associates a database cursor with an SQL statement producing a result set.

Saving SQL resources
SQL cursors and prepared statement consume resources that should be freed when useless.

To write efficient SQL in your programs, you can use prepared SQL. However, when using prepared SQL, a SQL statement handle is allocated on the client and server side, consuming resources. Depending to the database type, this can be a few bytes or a significant amount of memory. When executing several static SQL statements, the same SQL statement handle is reused and thus less memory is needed.

The BDL language allows you to use either static SQL or dynamic SQL, so it's in your hands to choose memory or performance. However, in some cases the same code will be used by different kinds of programs, needing either low resource usage or good performance. In many OLTP applications you can actually distinguish two type of programs:

• Programs where memory usage is not a problem but good performance is needed (typically, batch programs executed as a unique instance during the night).
• Programs where performance is less important but memory usage must be limited (typically, interactive programs executed as multiple instances for each application user).

To reuse the same code for interactive programs and batch programs, use the following pattern:

1. Define a local module variable as an indicator for the prepared statement.
2. Write a function returning the type of program (for example, ‘interactive’ or ‘batch’ mode).
3. Then, in a reusable function using SQL statements, prepare and free the statement based on the type of program, as shown in the next example.

```sql
PRIVATE DEFINE up_prepared BOOLEAN

FUNCTION getUserPermissions( username )
  DEFINE username VARCHAR(20)
  DEFINE cre, upd, del CHAR(1)
  IF NOT up_prepared THEN
    PREPARE up_stmt FROM "SELECT can_create, can_update, can_delete"
    || " FROM user_perms WHERE name = ?"
    LET up_prepared = TRUE
  END IF
  EXECUTE up_stmt USING username INTO cre, upd, del
  IF isInteractive() THEN
    FREE up_stmt
    LET up_prepared = FALSE
  END IF
  RETURN cre, upd, del
END FUNCTION
```

The first time this function is called, up_prepared is FALSE, the statement is prepared and the flag is set to TRUE. The statement is then executed, values are fetched. If the program is interactive, the statement is freed and the variable is set back to FALSE, forcing statement preparation in the next call of this function. If the program is not interactive, the statement handle is kept and the next call will not require to re-prepare the statement.

**Note:** Modern database client interfaces support "deferred prepare", to postpone the SQL statement preparation to the first execution: This avoids a roundtrip with the DB server. In such case, the benefit of using prepared statements makes only sense when the SQL is executed many times.

**Related concepts**

- **Dynamic SQL management** on page 709
  Explains how to execute and manage SQL statements at runtime.

- **Static SQL statements** on page 694
  Describes static SQL statements supported in the language.

**Optimizing scrollable cursors**

A programming pattern to get fresh data from scrollable cursors.

Generally, when using scrollable cursors, the database server or the database client software (i.e. the application) will make a static copy of the result set produced by the SELECT statement. For example, when using an IBM® Informix® database engine, each scrollable cursor will create a temporary table to hold the result set. Thus, if the SELECT statement returns all columns of the table(s) in the FROM clause, the database software will make a copy of all these values. This practice has two disadvantages: A lot of resources are consumed, and the data is static.

A good programming pattern to save resources and always get fresh data from the database server is to declare two cursors based on the primary key usage, if the underlying database table has a primary key (or unique index constraint): The first cursor must be a scrollable cursor that executes the SELECT statement, but returns only the primary keys. The SELECT statement of this first cursor is typically assembled at runtime with the where-part
produced by a CONSTRUCT interactive instruction, to give a subset of the rows stored in the database. The second cursor (actually, a PREPARE/EXECUTE statement handle) performs a single-row SELECT statement listing all columns to be fetched for a given record, based on the primary key value of the current row in the scrollable cursor list. The second statement must use a ? question mark place holder to execute the single-row SELECT with the current primary key as SQL parameter.

If the primary key SELECT statement needs to be ordered, check that the database engine allows that columns used in the ORDER BY clause do not need to appear in the SELECT list. For example, this was the case with IBM® Informix® servers prior to version 9.4. If needed, the SELECT list can be completed with the columns used in ORDER BY, you can then just list the variable that holds the primary key in the INTO clause of FETCH.

Note also that the primary key result set is static. That is, if new rows are inserted in the database or if rows referenced by the scroll cursor are deleted after the scroll cursor was opened, the result set will be outdated. In this case, you can refresh the primary key result set by re-executing the scroll cursor with CLOSE/OPEN commands.

This code example illustrates this programming pattern:

```sql
MAIN
  DEFINE wp VARCHAR(500)
  DATABASE test1
  -- OPEN FORM / DISPLAY FORM with c_id and c_name fields
  ...
  -- CONSTRUCT generates wp string...
  ...
  LET wp = "c_name LIKE 'J%'"
  DECLARE clist SCROLL CURSOR FROM "SELECT c_id FROM customer WHERE " || wp
  PREPARE crec FROM "SELECT * FROM customer WHERE c_id = ?"
  OPEN clist
  COMMAND "Test"
    MENU "Test"
      COMMAND "First" CALL disp_cust("F")
      COMMAND "Next" CALL disp_cust("N")
      COMMAND "Previous" CALL disp_cust("P")
      COMMAND "Last" CALL disp_cust("L")
      COMMAND "Refresh" CLOSE clist OPEN clist
      COMMAND "Quit" EXIT MENU
  END MENU
  FREE crec
  FREE clist
END MAIN

FUNCTION disp_cust(m)
  DEFINE m CHAR(1)
  DEFINE rec RECORD
    c_id INTEGER,
    c_name VARCHAR(50)
  END RECORD
  CASE m
    WHEN "F" FETCH FIRST clist INTO rec.c_id
    WHEN "N" FETCH NEXT clist INTO rec.c_id
    WHEN "P" FETCH PREVIOUS clist INTO rec.c_id
    WHEN "L" FETCH LAST clist INTO rec.c_id
  END CASE
  INITIALIZE rec.* TO NULL
  IF SQLCA.SQLCODE == NOTFOUND THEN
    ERROR "You reached to top or bottom of the result set."
  ELSE
    EXECUTE crec USING rec.c_id INTO rec.*
    IF SQLCA.SQLCODE == NOTFOUND THEN
      ERROR "Row was not found in the database, refresh the result set."
    END IF
  END IF
  DISPLAY BY NAME rec.*
```
Related concepts

Scrollable cursors on page 623
How scrollable cursors can be supported on different databases.

Understanding database result sets on page 715
This is an introduction to database result sets.

Query by example (CONSTRUCT) on page 1565
The CONSTRUCT instruction implements database query criteria input in an application form.

Database connections

Explains how to manage database connections in a program.

Understanding database connections

This is an introduction to database connections.

A database connection is a session of work, opened by the program to communicate with a specific database server, in order to execute SQL statements as a specific user.

Before working with database connections, make sure you have properly installed and configured all software, using the correct database client software/environment, and BDL database driver. It is very important to understand database client settings, regarding user authentication as well as database client character set configuration.

Note that on some platforms like on mobile devices, Genero BDL includes the SQLite lightweight database library, which is the default. Therefore, when executing programs on these platforms, there is no need to install a database client software and configure the database driver for the runtime system.

A database connection is initiated with the DATABASE instruction, or with the CONNECT TO instruction: The CONNECT TO instruction allows to specify database user credentials with the USER/USING clauses.

Multiple database connections can be established in a Genero BDL program.

Figure 24: Schema example of a program using three database connections

The database user can be identified explicitly for each connection. Usually, the user is identified by a login and a password, or by using the authentication mechanism of the operating system (or even from a tier security system).

Note: Database connection instructions can not be prepared and executed as dynamic SQL statements.
There are two kinds of connection modes: **unique-session** and **multi-session** mode. When using the DATABASE and CLOSE DATABASE instructions, the program is in unique-session mode. When using the CONNECT TO, SET CONNECTION and DISCONNECT instructions, the program is in multi-session mode.

**Important:** It is not possible to mix unique-session and multi-session modes.

Once connected to a database server, the program uses the current session to execute SQL statements in that context.

**Related concepts**

- **Database client settings** on page 526
  This section describes the settings defining the locale for the database client.

- **Database driver specification (driver)** on page 662

**Opening a database connection**

A database connection identifies the SQL database server and the database entity the program connects to, in order to execute SQL statements.

To connect to a database server, the database driver needs to be loaded, and the SQL data source must be provided. Additionally, user authentication with user name / password may also be needed. All these parameters define connection information.

There are different ways to give connection information, and it is possible to mix the different methods to specify connection parameters. However, if provided, the database user name and password have to be specified together with the same method.

A database connection is performed in programs with the DATABASE or CONNECT TO instruction:

```
CONNECT TO dbspec [USER username USING password]
```

or

```
DATABASE dbspec
```

Prefer the CONNECT TO instruction, as it allows to specify a user name and password.

For portability reasons, it is not recommended that you use database vendor specific syntax (such as 'dbname@dbserver') in the DATABASE or CONNECT TO instructions: Connections must be identified in programs by a single name, while connection parameters are provided in external files.

Indirect database specification uses entries in the FGLPROFILE configuration file: When a DATABASE or CONNECT TO instruction is executed with the parameter `dbspec`, the runtime system first looks into FGLPROFILE for entries starting with `dbi.database.dbspec`, and uses these connection parameters if found. Otherwise, the runtime system will do direct database specification, by using the `dbspec` string to connect to the server.

**Important:** When using FGLPROFILE entries for database connection parameters, keep in mind that entries must be written in lowercase.

Use a string variable with the DATABASE or CONNECT TO statement, in order to specify the database source at runtime. This solution gives you the best flexibility.

The string variable can be set from your own configuration file, from a program argument or from an environment variable.

**Example**

```
MAIN
    DEFINE db, us, pwd CHAR(50)
    LET db = fgl_getenv("MYDBSOURCE")
    LET us = arg_val(2)
    LET pwd = arg_val(3)
    CONNECT TO db USER us USING pwd
...
```
Related concepts
Direct database specification method on page 667
Genero BDL applies direct database source specification when no FGLPROFILE entry corresponds to the database name used in programs.

Indirect database specification method on page 667
Genero BDL allows to define database connection parameters in FGLPROFILE, that can be referenced by a single identifier in programs.

Unique session mode connection instructions on page 682
Opening and closing a database for a unique session.

Multi-session mode connection instructions on page 683
Opening and closing a database for a unique session.

Database client environment
To connect to a database server, Genero BDL programs use vendor’s database client software.
The database client software is usually included in the database server software, so you do not need to install it when your programs are executed on the same machine as the database server.
However, you must install the database client software in three-tier configurations, when applications and database servers run on different systems.
This section describes basic configuration elements of the database client environment for some well-known database servers.

**Related concepts**
Database client settings on page 526
This section describes the settings defining the locale for the database client.

**IBM® DB2 LUW™**
1. The DB2DIR environment variable must define the DB2® software installation path.
2. The PATH environment variable must define the access path to database client programs.
3. On UNIX®, LD_LIBRARY_PATH (or equivalent) must hold the path to $DB2DIR/lib.
4. The DB2® client library 'DB2DIR/lib/libdb2*' must be available.
5. The remote server node and the remote database must be declared locally with the CATALOG db2 command.
6. Make sure the database client locale is properly defined.
7. You can make a connection test with the IBM® db2 command line tool.

**Related tasks**
Prepare the runtime environment - connecting to the database on page 759

**IBM® Informix® Dynamic Server**
1. The INFORMIXDIR environment variable must define the IBM® Informix® software installation path.
2. The PATH environment variable must define the access path to database client programs.
3. On UNIX®, LD_LIBRARY_PATH (or equivalent) must hold the path to $INFORMIXDIR/lib: $INFORMIXDIR/lib/esql.
4. The IBM® Informix® client libraries 'INFORMIXDIR/lib/*' must be available.
5. The INFORMIXSERVER environment variable can be used to define the name of the database server.
6. The sqlhost file must define the database server identified by INFORMIXSERVER.
7. Make sure the database client locale is properly defined.
8. You can make a connection test with the IBM® Informix® dbaccess command line tool.

**Related concepts**
Installation (Runtime Configuration) on page 742
Oracle® MySQL
1. The MYSQL_HOME environment variable must define the Oracle® MySQL software installation path.
2. The PATH environment variable must define the access path to database client programs.
3. On UNIX™, LD_LIBRARY_PATH (or equivalent) must hold the path to $MYSQL_HOME/lib.
4. Make sure the database client locale is properly defined.
5. You can make a connection test with the mysql command line tool.

Related tasks
Prepare the runtime environment - connecting to the database on page 877

MariaDB
1. The MYSQL_HOME environment variable must define the MariaDB software installation path.
2. The PATH environment variable must define the access path to database client programs.
3. On UNIX™, LD_LIBRARY_PATH (or equivalent) must hold the path to $MYSQL_HOME/lib.
4. Make sure the database client locale is properly defined.
5. You can make a connection test with the mysql command line tool.

Related tasks
Prepare the runtime environment - connecting to the database on page 877

Oracle database
1. The ORACLE_HOME environment variable must define the Oracle software installation path.
2. The ORACLE_SID environment variable can be used to define the name of the local database instance.
3. The PATH environment variable must define the access path to database client programs.
4. On UNIX™, LD_LIBRARY_PATH (or equivalent) must hold the path to $ORACLE_HOME/lib.
5. The Oracle® client library 'ORACLE_HOME/lib/libclntsh*' must be available.
6. The TNSNAMES.ORA file must define the database server identifiers for remote connections (the Oracle® Listener must be started on the database server to allow remote connections).
7. The SQLNET.ORA file must define network settings for remote connections.
8. Make sure the database client locale is properly defined.
9. You can make a connection test with the Oracle® sqlplus command line tool.

Related tasks
Prepare the runtime environment - connecting to the database on page 910

PostgreSQL
1. The PGDIR environment variable must define the PostgreSQL software installation path.
2. The PATH environment variable must define the access path to database client programs.
3. On UNIX™, LD_LIBRARY_PATH (or equivalent) must hold the path to $PGDIR/lib.
4. The PostgreSQL client library 'PGDIR/lib/libpq*' must be available.
5. On the database server, the pg_hba.conf file must define security policies.
6. Make sure the database client locale is properly defined.
7. You can make a connection test with the PostgreSQL psql command line tool.

Related tasks
Prepare the runtime environment - connecting to the database on page 965

Microsoft® SQL Server
1. Make sure that ODBC data source is defined on database client and database server systems, with the correct ODBC driver. Note that Genero FGL provides different types of SQL Server drivers:
Important: Configure your ODBC data source to use the appropriate Microsoft SQL Server driver.

- On Microsoft® Windows® platforms:
  - Use an SNC (dbmsnc*) driver either with the Microsoft ODBC driver for SQL Server, or with the Microsoft SQL Native Client driver (msdn.microsoft.com):
    - For Microsoft ODBC 17 (MSODBCSQL17.DLL), use dbmsnc_17.
    - For Microsoft ODBC 13 (MSODBCSQL13.DLL), use dbmsnc_13.
    - For Microsoft SQL Native Client 11 (SQLNCLI11.DLL), use dbmsnc_11.
  - On Linux® platforms:
    - With the SNC (dbmsnc*) driver, use the Microsoft ODBC for SQL Server on Linux® client (msdn.microsoft.com):
      - For Microsoft ODBC 17 (libmsodbcsql-17.so), use dbmsnc_17.
      - For Microsoft ODBC 13 (libmsodbcsql-13.so), use dbmsnc_13.
      Minimum Microsoft ODBC for SQL Server on Linux® version: 13.0.
    - With the FTM (dbmftm*) driver, use the FreeTDS ODBC client (www.freetds.org, http://www.freetds.org/files/stable/).
      Minimum FreeTDS version: 1.00.104.
    - With the ESM (dbmesm*) driver, use the Easysoft ODBC driver for SQL Server (www.easysoft.com).
      Minimum Easysoft version 1.5; Version 1.9 is strongly recommended.
  2. On Windows®, the PATH environment variable must define the access path to database client programs (ODBC32.DLL). On UNIX platforms, check database client software documentation for environment settings (LD_LIBRARY_PATH, ldconfig).
  3. On Windows®, check the SQL Server Client configuration with the Client Network Utility tool. Verify that the ANSI to OEM conversion corresponds to the execution of applications in a CONSOLE environment.
  4. Make sure the database client locale is properly defined. On UNIX platforms, check that the client character set parameter of the ODBC data source corresponds the locale used by the application (LANG/LC_ALL).
  5. On Windows®, you can make a connection test with the Microsoft® Query Analyzer tool. On UNIX, see client software documentation for available SQL command tools (isql command line tool for example).

Related tasks
Prepare the runtime environment - connecting to the database on page 828

SQLite
1. The SQLite database driver includes the SQLite library, except on systems where that library is commonly available, like Linux® distributions, Mac OS X and mobile devices.
2. Database locale: The SQLite library uses UTF-8. If the current application character set (LANG/LC_ALL) is not UTF-8, like plain ASCII or UTF-8, the SQLite database driver will make appropriate character set conversions.
3. You can make a connection test with the sqlite3 command line tool.

Related tasks
Prepare the runtime environment - connecting to the database on page 1005

SAP ASE
1. The SYBASE environment variable must define the SAP® ASE software installation path.
2. The PATH environment variable must define the access path to database client programs.
3. On UNIX™, LD_LIBRARY_PATH (or equivalent) must hold the path to the client libraries libsybct.so and libsybcs.so. On Windows®, the path to the DLLs must be defined in PATH.
4. Check the SAP® ASE Client configuration, especially server name definition in connection's directory source, see DSQUERY environment variable.
5. Make sure the database client locale is properly defined.
6. You can make a connection test with the SAP ASE `isql` command line tool.

**Related tasks**

- Prepare the runtime environment - connecting to the database on page 1035

**SAP HANA**

1. The PATH environment variable must define the access path to SAP HANA® database client programs.
2. On UNIX™, LD_LIBRARY_PATH (or equivalent) must hold the path to the client libraries `libodbcHDB.so`. On Windows™, the path to the DLL must be defined in PATH.
3. Check the SAP HANA ODBC parameters, especially the `SERVERNODE` and `DATABASENAME` parameters.
4. There is no need to define any database client locale for the SAP HANA client: The Genero ODI driver will do the appropriate codeset conversions based on LANG/LC_ALL settings.
5. You can make a connection test with the SAP HANA `hdsqsl` command line tool:

   ```
   hdsqsl -n fermi:39013 -d test1 -u hdbuser -p fourjs
   ```

**Related tasks**

- Prepare the runtime environment - connecting to the database on page 1070

**Connection parameters**

This section describes the different parameters which need to be specified in order to connect to a database.

The parameters can be provided with different methods (in the connection string or in FGLPROFILE settings). Some of these parameters are optional.

For example, if the database user is authenticated by the operating system, username/password parameters are not needed.

**Database source specification (source)**

In database connection parameters, the `source` parameter identifies the data source name.

If the `source` parameter is defined with an empty value (""), the database interface connects to the default database server, which is usually the local server.

If the `source` entry is not present in FGLPROFILE, direct database specification method takes place.

**Table 193: Meaning of the `source` connection parameter for supported databases**

<table>
<thead>
<tr>
<th>Database Type</th>
<th>Value of &quot;source&quot; entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic ODBC</td>
<td><code>datasource</code></td>
<td>ODBC Data Source</td>
</tr>
<tr>
<td>IBM® Informix®</td>
<td><code>dbname[@dbserveral]</code></td>
<td>IBM® Informix® database specification</td>
</tr>
<tr>
<td>IBM® DB2®</td>
<td><code>dsname</code></td>
<td>DB2® Catalogued Database</td>
</tr>
<tr>
<td></td>
<td><code>dbname@[host]:port[]</code></td>
<td>Database Name @ Host Name: TCP Port or</td>
</tr>
<tr>
<td>Oracle® MySQL / MariaDB</td>
<td>or <code>dbname@[localhost~socket]</code></td>
<td>Database Name @ Local host ~ UNIX™ socket file</td>
</tr>
<tr>
<td>Oracle® Database</td>
<td><code>tnsname</code></td>
<td>Oracle® TNS Service name</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td><code>dbname@[host]:port[]</code></td>
<td>Database Name @ Host Name : TCP Port ? PostgreSQL URI-style query string options</td>
</tr>
<tr>
<td>Microsoft™ SQL Server</td>
<td><code>datasource[?options]</code></td>
<td>ODBC Data Source ? ODBC connection string parameters</td>
</tr>
</tbody>
</table>
### Related concepts

- **Direct database specification method** on page 667

Genero BDL applies direct database source specification when no FGLPROFILE entry corresponds to the database name used in programs.

### Database driver specification (driver)

In database connection parameters, the `driver` parameter identifies the type of database driver to be used.

The driver must correspond to the database client software.

**Important:** Pay attention to the binary architecture of the database client software: Genero runtime system and database client binaries must match. For example, a 32 bit Oracle® client cannot be used with a Genero 64 bit runtime system.

We distinguish two types of database driver names:

- **Generic driver names** ("dbmora", "dbmsnc"), and aliases ("oracle", "sqlserver")
- **Version-stamped driver names** ("dbmora_12", "dbmsnc_17", "dbmmys_8_0")

A driver name "dbmxxx" identifies a generic driver name for the database server identified by the code xxx.

For example, in FGLPROFILE, to define the database driver for the Oracle® OCI client (code "ora"), use the name "dbmora":

```plaintext
dbi.database.stores.driver = "dbmora"
```

**Note:** For convenience, it is also possible to specify a long name (alias) such as "oracle" or "sqlserver", as defined in the database driver table below.

The generic driver names (like dbmora) require the latest database client software available on the platform. This can change from one Genero BDL release to another, when supporting a new database client version. Use the version-stamped driver name (like dbmora_12), to point to a specific database client version, for example when the most recent database client software is not available on the platform.

Check for library dependency on your system, to identify the database client library required by the driver with the generic name. The driver definition table below lists the driver names for each supported database client types and versions. For example, on Linux® platform, use the `ldd` command:

```bash
$ ldd $FGLDIR/dbdrivers/dbmmys.so
... 
libmysqlclient.so.21 => ...
... 
```

Drivers with generic names are compatible with the latest database client version available on the platform. Depending on the platform, the same generic driver name can refer to different database client software versions. For example, on an old platform where only MySQL 5.7 client is available, dbmmys will identify the dbmmys_5_7 driver, depending on libmysqlclient.so.20. On a more recent platform where MySQL 8.0 is available, dbmmys will identify the dbmmys_8_0 driver, depending on libmysqlclient.so.21.
To limit the number of drivers, if the database client software allows it, the drivers are build with the oldest database client version that is compatible with the latest available database client versions. For example, the `dbmdb2_10` driver is linked to `libdb2.so.1`, which is supported by the IBM DB2 V10 and V11 client software.

**Note:** A given driver (combined with the corresponding database client software library) can connect to a database server of an older version, if the database vendor client/server protocol supports the combination. For example, you can use an Oracle® client version 12c to connect to an 11g server. The ODI driver will then adapt SQL translations and emulations to the target database server version.

A default driver can be specified with the `dbi.default.driver` FGLPROFILE entry. This driver will be used for all database connections that do not specify the driver explicitly:

```
dbi.default.driver = "dbmora"
```

If this entry is not defined, and if no driver parameter is specified for the data source, the driver name defaults to `dbmdefault`. This default driver is a copy of the database driver that was chosen during installation.

### Table 194: Database driver names

<table>
<thead>
<tr>
<th>Versionned name / Generic name / Alias</th>
<th>Code</th>
<th>Database client software version</th>
<th>UNIX™ shared objects</th>
<th>Microsoft™ Windows® DLLs</th>
<th>Mac OS X™ dynamic libraries</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbmase_16/dbmase/sybase_ase</td>
<td>ase</td>
<td>SAP® ASE Open Client Library 16.x</td>
<td>libsybctl[64].so,libsybctl[64].dll,N/A</td>
<td>libsybcs[64].so,libsybcs[64].dll</td>
<td></td>
</tr>
<tr>
<td>dbmdb2_10/dbmdb2/db2</td>
<td>db2</td>
<td>IBM® DB2® LUW Client 10.x, 11.x</td>
<td>libdb2.so.1</td>
<td>db2cli.dll</td>
<td>N/A</td>
</tr>
<tr>
<td>dbmesm_1/dbmesm/easysoft_sqlserver</td>
<td>esm</td>
<td>Easysoft ODBC for SQL Server</td>
<td>libessqlsrv.so</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>dbmhdb_2/dbmhdb/sap_hana</td>
<td>hbox</td>
<td>SAP HANA® ODBC</td>
<td>libodbcHDB.so,libodbcHDB.dll,N/A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dbmifx_9/dbmifx/informix</td>
<td>ifx</td>
<td>IBM® Informix® CSDK 3.70 and higher</td>
<td>libifsql.so,</td>
<td>libifasf.dylib,</td>
<td>libifsql.dylib,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>libifgen.so,</td>
<td>libifgen.dylib,</td>
<td>libifgen.dylib,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>libifos.so,</td>
<td>libifos.dylib,</td>
<td>libifos.dylib,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>libifgls.so,</td>
<td>libifgls.dylib,</td>
<td>libifgls.dylib,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>libifglx.so</td>
<td>libifglx.dylib</td>
<td>libifglx.dylib,</td>
</tr>
<tr>
<td>dbmmdb_10_2/dbmmdb/mariadb</td>
<td>mdb</td>
<td>MariaDB Client 10.2.x and higher</td>
<td>libmariadb.so.3,libmariadb.dll</td>
<td>libmariadb.so.3,libmariadb.dll</td>
<td></td>
</tr>
<tr>
<td>dbmmys_5_6</td>
<td>mys</td>
<td>Oracle® MySQL Client 5.6.x</td>
<td>libmysqlcclient.so,libmysqlcclient.so</td>
<td>libmysqlcclient.so,libmysqlcclient.so</td>
<td></td>
</tr>
<tr>
<td>dbmmys_5_7</td>
<td>mys</td>
<td>Oracle® MySQL Client 5.7.x</td>
<td>libmysqlcclient.so,libmysqlcclient.so</td>
<td>libmysqlcclient.so,libmysqlcclient.so</td>
<td></td>
</tr>
<tr>
<td>dbmmys_8_0/dbmmys/mysql</td>
<td>mys</td>
<td>Oracle® MySQL Client 8.0.x and higher</td>
<td>libmysqlcclient.so,libmysqlcclient.so</td>
<td>libmysqlcclient.so,libmysqlcclient.so</td>
<td></td>
</tr>
<tr>
<td>dbmntz_6/dbmntz/netezza</td>
<td>nzt</td>
<td>IBM® Netezza® (6.x)</td>
<td>libnzodbc.so</td>
<td>odbc32.dll,N/A</td>
<td>libnzodbc.so,N/A</td>
</tr>
<tr>
<td>dbmodc_3/dbmodc/odbc</td>
<td>odc</td>
<td>Generic ODBC (ODBC 3.x)</td>
<td>libodbc.so</td>
<td>odbc32.dll,?</td>
<td>libodbc.so</td>
</tr>
<tr>
<td>dbmora_11</td>
<td>ora</td>
<td>OCI Client V11</td>
<td>libclntsh.so.11,libclntsh.so.11</td>
<td>libclntsh.so.11,libclntsh.so.11</td>
<td></td>
</tr>
<tr>
<td>dbmora_12</td>
<td>ora</td>
<td>OCI Client V12</td>
<td>libclntsh.so.12,libclntsh.so.12</td>
<td>libclntsh.so.12,libclntsh.so.12</td>
<td></td>
</tr>
<tr>
<td>dbmora_18/dbmora/oracle</td>
<td>ora</td>
<td>OCI Client V18, V19</td>
<td>libclntsh.so.18,libclntsh.so.18</td>
<td>libclntsh.so.18,libclntsh.so.18</td>
<td></td>
</tr>
</tbody>
</table>
### SQL support

<table>
<thead>
<tr>
<th>Versionned name / Generic name / Alias</th>
<th>Code</th>
<th>Database client software version</th>
<th>UNIX™ shared objects</th>
<th>Microsoft™ Windows® DLLs</th>
<th>Mac OS X™ dynamic libraries</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbmpgs_9 / dbmpgs / postgresql</td>
<td>pgs</td>
<td>PostgreSQL Client 9.4, 9.5, 9.6, 10, 11 and 12</td>
<td>libpq.so.5</td>
<td>libpq.dll</td>
<td>libpq.5.dylib</td>
</tr>
<tr>
<td>dbmsnc_11</td>
<td>snc</td>
<td>SQL Server Native Client 2012 (V11)</td>
<td>N/A</td>
<td>odbc32.dll / SQLNCLI11.DLL</td>
<td>N/A</td>
</tr>
<tr>
<td>dbmsnc_13</td>
<td>snc</td>
<td>Microsoft™ ODBC 13 for SQL Server</td>
<td>libmsodbcsql-1</td>
<td>libmsodbcsql-1</td>
<td>libmsodbcsql-1</td>
</tr>
<tr>
<td>dbmsnc_17 / dbmsnc / sqlserver</td>
<td>snc</td>
<td>Microsoft™ ODBC 17 for SQL Server</td>
<td>libmsodbcsql-1</td>
<td>libmsodbcsql-1</td>
<td>libmsodbcsql-1</td>
</tr>
<tr>
<td>dbmftm_0 / dbmftm / freetds_sqlserver</td>
<td>ftm</td>
<td>FreeTDS ODBC version 1.00+</td>
<td>libtdsodbc.so.0</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>dbmsqt_3 / dbmsqt / sqlite</td>
<td>sqt</td>
<td>SQLite 3.x</td>
<td>libsqlite3.so.0</td>
<td>N/A (statically linked)</td>
<td>libsqlite3.dylib</td>
</tr>
</tbody>
</table>

### Related concepts

- Default database driver on page 664
- `fgl_db_driver_type()` on page 2275

Returns the 3-letter identifier/code of the current database driver.

#### Default database driver

The `dbi.default.driver` **FGPROFILE** entry defines a default database driver to be loaded, if the driver is not specified by the connection parameters.

```
  dbi.default.driver = "driver-name"
```

The driver name must be specified without the .so or .DLL extension.

If this configuration entry is not defined, the driver name defaults to dbmdefault.

#### Related concepts

- Database driver specification (driver) on page 662

#### User name and password (username/password)

In database connection parameters, the `username` and `password` parameters define the default database user, when the program uses the **DATABASE** instruction or the **CONNECT TO** instruction without the **USER/USING** clause.

The `username` and `password` **FGPROFILE** entries are **not encrypted**. These parameters are provided to simplify migration and are not recommended in production. It is better to use **CONNECT TO** with a **USER / USING** clause to avoid any security hole, setup OS user authentication, or use the connection callback method. Example of database servers supporting OS user authentication are: IBM® Informix®, Oracle®, and SQL Server.

**Important**: Do not write clear user passwords in your sources! It is recommended that `username` and `password` parameters are set from variables.

For backward compatibility reasons, when using the IBM® Informix® driver, the `username / password` specification is ignored by the **DATABASE** instruction, only the **CONNECT TO** instruction takes external (or callback) login parameters into account.

#### Related concepts

- Database user authentication on page 679
Different database user authentication methods exist.

**Connection parameters in database specification**

Connection parameters can be provided in the database specification string passed to the DATABASE and CONNECT TO instructions.

**Using connection parameters at runtime**

In the database name specification of CONNECT TO or DATABASE instructions, a + plus sign starts the list of connection specification parameters.

The connection specification parameters override the dbi.database connection parameters defined in FGLPROFILE.

In this example, driver, source and resource parameters are specified in the database specification string of the CONNECT TO instruction:

```plaintext
MAIN
   DEFINE db, un, up STRING
   LET db = "stores+driver='dbmora',source='orcl',resource='myconfig'"
   LET un = ...
   LET up = ...
   CONNECT TO db USER un USING up
   ...
END MAIN
```

**Important:**
- Do not hard code connection parameters in programs to be installed on a production site. Instead, build the connection string at runtime, or consider using the indirect database specification method.
- Do not specify the username and password parameters in connection specification parameters. Instead, provide the SQL user credentials with the USER/USING clause of CONNECT TO.
- Consider backslash interpretation in connection strings, as described below in Connection parameter parsing on page 666.

**Syntax for connection parameters**

Each parameter is defined with a name followed by an equal sign and a value enclosed in single quotes. Connection parameters must be separated by a comma:

```
dbname+parameter='value'[, ...]
```

In this syntax, `parameter` can be one of the following:

**Table 195: Connection parameters in the database specification string**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>resource</td>
<td>Specifies which 'dbi.database' entries have to be read from the FGLPROFILE configuration file.</td>
</tr>
<tr>
<td></td>
<td>When this property is set, the database interface reads <code>dbi.database.name.*</code> entries, where <code>name</code> is the value specified for the resource parameter.</td>
</tr>
<tr>
<td>driver</td>
<td>Defines the database driver library to be loaded (filename without extension).</td>
</tr>
<tr>
<td>source</td>
<td>Specifies the data source of the database.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| **username** | Defines the name of the database user.  
**Important:** Consider using CONNECT TO with USER/USING clause instead! |
| **password** | Defines the password of the database user.  
**Important:** Do not write clear user passwords in your sources! This parameter should be set from a variable value. |

**Passing a plus sign in a connection parameter**

With some databases, the `source` connection parameter can take different forms, that may contain a `+` sign. When specified directly as default `source` in the database connection specification, this `+` sign will be interpreted as the starting character for connection specification parameters, and produce error `-6373`: Invalid database connection string.

To workaround the `+` sign interpretation, put the `source` parameter explicitly in the connection specification.

For example, with SQL Server, you can specify ODBC connection string parameters with the `?options` syntax:

```
CONNECT TO "mydsn?APP=myappid;" USER un USING up
```

In the above example, the connection specification passed to the CONNECT TO instruction defines implicitly the "source" connection parameter. This string contains the ODBC data source name (`mydsn`), and, after the `?` question mark, the ODBC connection string parameter `APP=myappid`; defining the SQL client application identifier.

When using a plus sign in the ODBC connection string parameters, it will produce the error `-6373`. This can happen for example when using authentication credentials, as with the `CRED` parameter in the next example:

```
CONNECT TO "mydsn?APP=myappid;CRED=ZXB+2A" USER un USING up
```

To solve this issue, use an explicit "source" parameter in the connection specification of CONNECT TO:

```
CONNECT TO "mydbc+source='mydsn?APP=myappid;CRED=ZXB+2A'" USER un USING up
```

The plus sign in the "CRED" ODBC parameter value will be considered as part of the "source" parameter value.

**Connection parameter parsing**

The parameters and values after the `+` sign are parsed, and backslashes are interpreted as in a string literal.

For example, `\n` becomes a new line character in the final value used in the database driver.

To pass a backslash to the driver (for example when specifying a Windows® path), it must be doubled in the string value.

Furthermore, a backslash in a string literal of the source code must be quadrupled.

Consider implementing a utility function to double backslashes in a string:

```
FUNCTION escape_backslashes(str)
    DEFINE str STRING
    DEFINE buf base.StringBuffer
    LET buf = base.StringBuffer.create()
    CALL buf.append(str)
    CALL buf.replace("\\","\\\\",0)
    RETURN buf.toString()
END FUNCTION
```
**Related concepts**

*Indirect database specification method* on page 667
Genero BDL allows to define database connection parameters in FGLPROFILE, that can be referenced by a single identifier in programs.

*Database driver specification (driver)* on page 662
*Database source specification (source)* on page 661
*User name and password (username/password)* on page 664

**Direct database specification method**
Genero BDL applies direct database source specification when no FGLPROFILE entry corresponds to the database name used in programs.

*Direct database specification method* takes place when the database name used in a `DATABASE` or `CONNECT TO` instruction is not defined in `FGLPROFILE` with a `dbi.database.dbname.source` entry. In this case, the database specification used in the connection instruction is used as the data source.

This method is well known with IBM® Informix®, for example to specify the database server:

```
MAIN
    DATABASE stores@orion
    ...
END MAIN
```

In the next example, the database server is PostgreSQL. The string used in the connection instruction defines the PostgreSQL database (stock), the host (localhost), and the TCP service (5432) the postmaster is listening to. As PostgreSQL syntax is not allowed in the language, a CHAR variable must be used:

```
MAIN
    DEFINE db CHAR(50)
    LET db = "stock@localhost:5432"
    DATABASE db
    ...
END MAIN
```

This technique makes the compiled program specific to a given database server configuration. Consider using indirect database specification method instead of direct database specification.

**Related concepts**

*Indirect database specification method* on page 667
Genero BDL allows to define database connection parameters in FGLPROFILE, that can be referenced by a single identifier in programs.

**Indirect database specification method**
Genero BDL allows to define database connection parameters in FGLPROFILE, that can be referenced by a single identifier in programs.

*Indirect database specification method* takes place when the database name used in the `DATABASE` or `CONNECT TO` instruction corresponds to a `dbi.database.dbname.source` entry defined in the `FGLPROFILE` configuration file. In this case, the `dbname` database specification is used as a key to read the connection information from the configuration file.

In FGLPROFILE, the entries starting with `dbi.database.dbname` group information defining connection parameters for indirect database specification:

```
dbi.database.dbname.source   = "value"
dbi.database.dbname.driver   = "value"
dbi.database.dbname.username = "value"
dbi.database.dbname.password = "value"
```
-- Warning: Password is not encrypted, do not use in production!

Keep in mind that FGLPROFILE entry names are converted to lower case when loaded by the runtime system. In order to avoid any mistakes, it is recommended to write FGLPROFILE entry names and program database names in lower case.

In the next example, the program specifies a data source with the name stores, and FGLPROFILE defines the source and driver parameters for the stores data source:

Program:

    MAIN
    DATABASE stores
    ...  
    END MAIN

FGLPROFILE:

    dbi.database.stores.source   = "stock@localhost:5432"
    dbi.database.stores.driver   = "dbmpgs"

The indirect database specification technique is a flexible technique to define the database source: The database name in programs is a kind of alias for the real data source, which is defined in an external configuration file (i.e. FGLPROFILE), where entries can be easily changed on production sites without needing program recompilation.

Related concepts
Connection parameters on page 661
This section describes the different parameters which need to be specified in order to connect to a database.

Direct database specification method on page 667
Genero BDL applies direct database source specification when no FGLPROFILE entry corresponds to the database name used in programs.

IBM® Informix® emulation parameters in FGLPROFILE
Emulation of Informix® specific SQL features can be controlled with FGLPROFILE entries.

What are Informix® SQL emulation settings used for?
To simplify the migration process to other database servers such as IBM® Informix®, the database drivers can emulate some IBM® Informix-specific features like SERIAL columns and temporary tables; the drivers can also do some SQL syntax translation.

Avoid using IBM® Informix® emulations; write portable SQL code instead. IBM® Informix® emulations are only provided to help you in the migration process. Disabling IBM® Informix® emulations improves performance, because SQL statements do not have to be parsed to search for IBM® Informix-specific syntax.

Emulations can be controlled with FGLPROFILE parameters. You can disable all possible switches one-by-one, in order to test your programs for SQL compatibility.

    dbi.database.dsnname.ifxemul

This is a global switch to enable or disable IBM® Informix® emulations.

Values can be true or false. Default is true.

    dbi.database.stores.ifxemul = false
**dbi.database.dsnname.ifxemul.datatype.type**

The 'ifxemul.datatype' switches define whether the specified data type must be converted to a native type (for example, when creating a table with the CREATE TABLE statement).

Where `type` can be one of `char`, `varchar`, `datetime`, `decimal`, `money`, `float`, `real`, `integer`, `smallint`, `serial`, `text`, `byte`, `bigint`, `bigserial`, `int8`, `serial8`, `boolean`.

Default is `true` for all types.

```
dbi.database.stores.ifxemul.datatype.serial = false
```

**dbi.database.dsnname.ifxemul.datatype.serial.emulation**

This parameter can be used to control the serial generation technique used by the driver to generate auto-incremented values.

The value can be one of the following:

- `native` uses the database's native sequence generator directly in the table definitions (depends on the db type).
- `native2` uses a secondary native sequence generator directly in the table definitions (depends on the db type).
- `regtable` uses the SERIALREG table with triggers. It is slower than the `native` emulation.
- `trigseq", uses database sequence generator with triggers (not supported by all drivers).

Default is "native".

```
dbi.database.stores.ifxemul.datatype.serial.emulation = "native"
```

Serial emulations depend on the type of database server used. See SQL database guides on page 742 for more details.

**dbi.database.dsnname.ifxemul.datatype.serial.sqlerrd2**

Use this parameter to disable the automatic serial retrieval done by the drivers, to fill the SQLCA.SQLERRD[2] register. When this feature is enabled, depending on the type of serial emulation, drivers need to execute an additional SQL query to fetch the last generated serial. If you have implemented your own function to fetch the last serial and you cannot disable the whole serial emulation, set this parameter to false.

Default is `true`.

```
dbi.database.stores.ifxemul.datatype.serial.sqlerrd2 = false
```


**dbi.database.dsnname.ifxemul.temptables**

This switch can be used to control temporary table emulation.

Defaults is `true`.

```
dbi.database.stores.ifxemul.temptables = false
```

**dbi.database.dsnname.ifxemul.temptables.emulation**

This parameter can be used to specify what technique must be used to emulate temporary tables in the database server.
Possible values are "default", "private" and "global".

```plaintext
dbi.database.stores.ifxemul.temptables.emulation = "global"
```

See SQL database guides on page 742 for more details.

**dbi.database.dsnname.ifxemul.dblquotes**
This switch can be used to define whether double quoted strings must be converted to single quoted strings.
Default is true.

```plaintext
dbi.database.stores.ifxemul.dblquotes = false
```

If this emulation is enabled, all double quoted strings are converted, including database object names.

**dbi.database.dsnname.ifxemul.outers**
This switch can be used to control IBM® Informix® OUTER translation to native SQL outer join syntax.
Default is true.

```plaintext
dbi.database.stores.ifxemul.outers = false
```

**Note:** Consider using standard ISO outer joins in your SQL statements (LEFT OUTER).

**dbi.database.dsnname.ifxemul.today**
This switch can be used to convert the TODAY keyword to a native expression returning the current date.
Default is true.

```plaintext
dbi.database.stores.ifxemul.today = false
```

**dbi.database.dsnname.ifxemul.current**
This switch can be used to convert the CURRENT X TO Y expressions to a native expression returning the current time.
Default is true.

```plaintext
dbi.database.stores.ifxemul.current = false
```

**dbi.database.dsnname.ifxemul.selectunique**
This switch can be used to convert the SELECT UNIQUE to SELECT DISTINCT.
Default is true.

```plaintext
dbi.database.stores.ifxemul.selectunique = false
```

**Note:** Consider replacing all UNIQUE keywords by DISTINCT.

**dbi.database.dsnname.ifxemul.colsubs**
This switch can be used to control column substrings expressions (col[x,y]) to native substring expressions.
Default is true.

```plaintext
dbi.database.stores.ifxemul.colsubs = false
```
**Note:** Consider using substring SQL functions instead of [x,y] expressions in SQL.

**dbi.database.dsnname.ifxemul.matches**
This switch can be used to define whether MATCHES expressions must be converted to LIKE expressions.
Default is true.

```java
dbi.database.stores.ifxemul.matches = false
```

**Note:** Consider using LIKE expressions instead of MATCHES in SQL.

**dbi.database.dsnname.ifxemul.length**
This switch can be used to define whether LENGTH() function names have to be converted to the native equivalent.
Default is true.

```java
dbi.database.stores.ifxemul.length = true
```

**dbi.database.dsnname.ifxemul.rowid**
This switch can be used to define whether ROWID keywords have to be converted to native equivalent (for example, OID in PostgreSQL).
Default is true.

```java
dbi.database.stores.ifxemul.rowid = false
```

**Note:** Consider using primary keys instead of ROWIDs.

**dbi.database.dsnname.ifxemul.listupdate**
This switch can be used to convert the UPDATE statements using non-ANSI syntax.
Default is true.

```java
dbi.database.stores.ifxemul.listupdate = false
```

**dbi.database.dsnname.ifxemul.extend**
This switch can be used to convert simple EXTEND() expressions to native date/time expressions.
Default is true.

```java
dbi.database.stores.ifxemul.extend = true
```

**dbi.database.dsnname.ifxemul.rowlimiting**
This switch can be used to convert Informix SQL row limiting clause SELECT [SKIP n] FIRST m to the native SQL equivalent.

**Note:** The row limiting clause must not use SQL parameters. Only row limiting clauses using integer constants will be converted.
Default is true.

```java
dbi.database.stores.ifxemul.rowlimiting = true
```
**dbi.database.dsnname.ifxemul.nationalchars**

With some database brands, (in DDL statements only) this switch can be used to convert `CHAR(n)`, `VARCHAR(n)`, `LVARCHAR(n)` and `TEXT` data type names to the corresponding national character types of the target database. For example, with Microsoft™ SQL Server, the corresponding national character types are `NCHAR(n)`, `NVARCHAR(n)` and `NVARCHAR(MAX)`.

Default is false.

```
dbi.database.stores.ifxemul.nationalchars = true
```

**Note:** Consider the following:

- The `dbi.database.dsnname.ifxemul.nationalchars` parameter is by default false.
- The current locale must be multibyte (typically UTF-8) to get this type name conversion.
- The `ifxemul.nationalchars` switch is ignored, when the data type emulation parameter corresponding to the type (such as `dbi.database.dsnname.datatype.varchar`) is set to false.

For more details, see [CHAR/VARCHAR types with SQL Server](#) and [TEXT type with SQL Server](#).

**Related concepts**

**SQL programming** on page 590

Covers topics about interacting with a database server using SQL.

---

**Database type specific parameters in FGLPROFILE**

Specific connection parameters can be configured with FGLPROFILE entries.

The FGLPROFILE entries for database-server specific configuration all following the same syntax scheme:

```
dbi.database.dsnname.dbtype.param[.subparam] = "value"
```

Where `dbtype` identifies the database vendor type, such as `ifx`, `ora`, `db2`.

**IBM® DB2® specific FGLPROFILE parameters**

**dbi.database.dsnname.db2.schema**

Name of the database schema to be selected after connection is established.

```
dbi.database.stores.db2.schema = "store2"
```

Set this parameter to a specific schema in order to share the same table with all users.

**Deferred prepare: Use db2cli.ini DeferredPrepare**

Before version 3.20 of Genero BDL, it was possible to control the deferred prepare option with the `dbi.database.dsnname.db2.prepare.deferred` FGLPROFILE parameter. Starting with 3.20, this FGLPROFILE parameter is desupported/ignored, and deferred prepare is the default.

If `PREPARE/DECLARE` instructions must raise an SQL error in case of invalid SQL, DB2 CLI deferred prepare (`SQL_ATTR_DEFERRED_PREPARE`) can be disabled with the following `db2cli.ini` parameter:

```
DeferredPrepare = 0
```
Oracle DB specific FGLPROFILE parameters

**dbi.database.dsname.ora.schema**
Name of the database schema to be selected after connection is established.

```sql
dbi.database.stores.ora.schema = "store2"
```

Set this parameter to a specific schema in order to share the same table with all users.
For more details, see Database users on page 917.

**dbi.database.dsname.ora.prefetch.rows**
Maximum number of rows to prefetch.

```sql
dbi.database.stores.ora.prefetch.rows = 50
```

Use this parameter to increase performance by defining the maximum number of rows to be fetched into the db client buffer. However, the bigger this parameter is, the more memory that is used by each program. This parameter applies to all cursors in the program.
The default is 10 rows.

**dbi.database.dsname.ora.prefetch.memory**
Maximum buffer size for prefetching (in bytes).

```sql
dbi.database.stores.ora.prefetch.memory = 4096
```

This parameter is equivalent to `prefetch.rows`, but here you can specify the memory size instead of the number of rows. Like `prefetch.rows`, this parameter applies to all cursors in the program.
The default is 0, which means that memory size is not included in computing the number of rows to prefetch.

**dbi.database.dsname.ora.sid.command**
SQL command (SELECT) to generate a unique session id (used for temp table names).

```sql
dbi.database.stores.ora.sid.command = "SELECT TO_CHAR(SID)||'_'||TO_CHAR(SERIAL#) FROM V$SESSION WHERE AUDSID=USERENV('SESSIONID')"
```

By default, the driver uses "SELECT USERENV('SESSIONID') FROM DUAL”. This is the standard session identifier in Oracle®, but it can become a very large number and cannot be reset.

This parameter gives you the freedom to provide your own way to generate a session id.
The SELECT statement must return a single row with one single column.
Value can be an integer or an identifier such as "pre999".

**dbi.database.dsname.ora.date.ifxfetch**
Controls the way an Oracle® DATE is fetched into program variables, especially CHAR/VARCHAR targets.

```sql
dbi.database.stores.ora.date.ifxfetch = true
```

By default, since Oracle® DATE type is equivalent to `DATETIME YEAR TO SECOND`, values are fetched into CHAR/VARCHAR with time information and are formatted with the style `YYYY-MM-DD hh:mm:ss`. If you need to get the IBM® Informix® behavior, to fetch DATEs only with the YMD part following the `DBDATE` environment
variable, set this parameter to true. However, this parameter is useless when fetching Oracle© DATEs into DATE or DATETIME variables, which is the recommended way to hold date and time values in programs.

Default is false (with time information, using normalized format).

```
dbi.dbname.ora.temptables.schema.source
```

Defines the source of the schema name used for temporary table emulation.

Possible values are:

- "login": The user name specified in the connection parameters.
- "command": The value returned by a SELECT statement specified in the `ora.temptables.schema.command` parameter.

```
dbi.dbname.stores.ora.temptables.schema.source = "login"
```

By default, when using the default temporary table emulation, the driver uses no schema at all. When using global temporary table emulation, the driver uses the TEMPTABS schema by default.

When specifying a "command" source for the schema, you can provide your own SELECT statement to produce the schema name to be used when creating a table.

```
dbi.dbname.ora.temptables.schema.command
```

SQL command (SELECT) to get the schema name to be used for temporary table emulation.

```
dbi.dbname.stores.ora.temptables.schema.command = "SELECT SYS_CONTEXT('USERENV','SESSION_USER') FROM DUAL"
```

This configuration parameter is only taken into account if the `ora.temptables.schema.source` parameter is defined as "command".

The SELECT statement must return a single row with one single column.

Value must be a character string.

```
dbi.dbname.ora.temptables.tablespace
```

Defines the tablespace to be used for temporary table emulation.

```
dbi.dbname.stores.ora.temptables.tablespace = "mytemptabs"
```

By default:

1. When using the default temporary table emulation, the driver uses the TEMPTABS tablespace. The tablespace specified in the `ora.temptables.tablespace` entry must be a permanent tablespace.
2. When using global temporary table emulation, the driver uses no tablespace by default. The tablespace specified in the `ora.temptables.tablespace` entry must be a temporary tablespace.
3. When using private temporary table emulation, the driver uses no tablespace by default. The tablespace specified in the `ora.temptables.tablespace` entry must be a temporary tablespace.

For more details, see Temporary tables on page 943.

```
dbi.dbname.ora.temptables.private.prefix.source
```

Defines the method to find out the prefix for Oracle private temporary tables.

Possible values are:
- "value": The PTT is specified with the
  `dbi.database.dsname.ora.temptables.private.prefix.value` parameter.
- "command": The PTT is found with the SELECT statement specified in the
  `dbi.database.dsname.ora.temptables.private.prefix.command` parameter.

For more details, see Using the private temporary table emulation on page 949.

**`dbi.database.dsname.ora.temptables.private.prefix.value`**

Defines the prefix for Oracle private temporary tables when
`dbi.database.dsname.ora.temptables.private.prefix.source = "value"`.  
For example:

```
  dbi.database.stores.ora.temptables.private.prefix.source = "value"
  dbi.database.stores.ora.temptables.private.prefix.value = "myprefix"
```

For more details, see Using the private temporary table emulation on page 949.

**`dbi.database.dsname.ora.temptables.private.prefix.source`**

Defines the SELECT statement to find the prefix for Oracle private temporary tables when
`dbi.database.dsname.ora.temptables.private.prefix.source = "command"`.  
For example:

```
  dbi.database.stores.ora.temptables.private.prefix.source = "command"
  dbi.database.stores.ora.temptables.private.prefix.command = "select
                     'myprefix' from dual"
```

For more details, see Using the private temporary table emulation on page 949.

**Oracle® MySQL specific FGLPROFILE parameters**

**`dbi.database.dsname.mys.config`**

Defines an explicit configuration to read MySQL options from.

```
  dbi.database.stores.mys.config = "/opt/myapp/etc/my.cnf"
```

This parameter will be passed to the MySQL API function `mysql_options((MYSQL*), MYSQL_READ_DEFAULT_FILE, filename )`.

It can be used to bypass reading the default MySQL configuration files, to define database client settings in the [client] group, such as the client character set with the default-character-set option.

**Note:**

On Microsoft™ Windows® platforms, the configuration file must be in DOS format.

**MariaDB specific FGLPROFILE parameters**

**`dbi.database.dsname.mdb.config`**

Defines an explicit configuration to read MariaDB options from.

```
  dbi.database.stores.mdb.config = "/opt/myapp/etc/my.cnf"
```

This parameter will be passed to the MariaDB API function `mysql_options((MYSQL*), MYSQL_READ_DEFAULT_FILE, filename )`.
It can be used to bypass reading the default MariaDB configuration files, to define database client settings in the [client] group, such as the client character set with the `default-character-set` option.

**Note:**

On Microsoft™ Windows® platforms, the configuration file must be in DOS format.

**PostgreSQL specific FGLPROFILE parameters**

```bash
dbi.database.dsname.pgs.prefetch.rows
```

Maximum number of rows to be pre-fetched.

```bash
dbi.database.stores.pgs.prefetch.rows = 100
```

Use this parameter to increase performance by defining the maximum number of rows to be fetched into the db client buffer. However, the bigger this parameter is, the more memory is used by each program. This parameter applies to all cursors in the program.

The default is 50 rows, which is a good compromise for regular interactive OTLP applications.

When using server-side cursors (DECLARE CURSOR), this parameter will define the number of rows fetched with the PostgreSQL `FETCH cursor FORWARD nbrows` command. This parameter has no effect on static SELECT statements, nor on SELECT statements executed with PREPARE + EXECUTE INTO.

```bash
dbi.database.dsname.pgs.schema
```

Defines the PostgreSQL schema search path after connection is established.

```bash
dbi.database.stores.pgs.schema = "$user",public,stock"
```

The value specified in this parameter will be used to execute the `SET search_path TO fglprofile-value` SQL instruction.

See also Name resolution of SQL objects on page 987.

**SQL Server (MS ODBC) specific FGLPROFILE parameters**

```bash
dbi.database.dsname.snc.logintime
```

Connection timeout (in seconds).

```bash
dbi.database.stores.snc.logintime = 5
```

Set this parameter to raise an SQL error if the connection can not be established after the given number of seconds. The default is 5 seconds.

```bash
dbi.database.dsname.snc.prefetch.rows
```

Maximum number of rows to be pre-fetched.

```bash
dbi.database.stores.snc.prefetch.rows = 50
```

Use this parameter to increase performance by defining the maximum number of rows to be fetched into the db client buffer. However, the bigger this parameter is, the more memory is used by each program. The default is 10 rows.

```bash
dbi.database.dsname.snc.widechar
```

Control single-char / wide-char mode usage for character string data.
**Note:** Only set the `snc.widechar` parameter to `false`, if you use char/varchar/text columns in the database, and the application locale is a multibyte character set (such as BIG5).

```cpp
dbi.database.stores.snc.widechar = false
```

By default, the SNC driver will select the expected char mode, depending on the current application locale (LANG/LC_ALL):

- In wide char mode, the SNC driver uses SQLWCHAR ODBC API functions, by converting the character data from the current locale to UCS/2, it will add the N prefix before string literals, and will bind SQL parameters with SQL_C_WCHAR and SQL_WCHAR/SQL_WVARCHAR types.
- In single char mode, the SNC driver will pass the character strings without conversion to "ASCII" ODBC API functions, it will leave the string literals (without adding the N prefix) and bind character string parameters with SQL_C_CHAR and SQL_CHAR/SQL_VARCHAR SQL types.

For more details, see also [SQL Server Adaptation Guide: CHARACTER data types](#).

**SQL Server (FreeTDS driver) specific FGLPROFILE parameters**

```cpp
dbi.database.dsname.ftm.logintime
```

Connection timeout (in seconds).

```cpp
dbi.database.stores.ftm.logintime = 5
```

Set this parameter to raise an SQL error if the connection can not be established after the given number of seconds. The default is 5 seconds.

```cpp
dbi.database.dsname.ftm.prefetch.rows
```

Maximum number of rows to be pre-fetched.

```cpp
dbi.database.stores.ftm.prefetch.rows = 50
```

Use this parameter to increase performance by defining the maximum number of rows to be fetched into the db client buffer. However, the bigger this parameter is, the more memory is used by each program. The default is 10 rows.

**SQL Server (Easysoft driver) specific FGLPROFILE parameters**

```cpp
dbi.database.dsname.esm.logintime
```

Connection timeout (in seconds).

```cpp
dbi.database.stores.esm.logintime = 5
```

Set this parameter to raise an SQL error if the connection can not be established after the given number of seconds. The default is 5 seconds.

```cpp
dbi.database.dsname.esm.prefetch.rows
```

Maximum number of rows to be pre-fetched.

```cpp
dbi.database.stores.esm.prefetch.rows = 50
```

Use this parameter to increase performance by defining the maximum number of rows to be fetched into the db client buffer. However, the bigger this parameter is, the more memory is used by each program.
The default is 10 rows.

**SAP ASE specific FGLPROFILE parameters**

`dbi.database.dsname.ase.logintime`

Connection timeout (in seconds).

`dbi.database.stores.ase.logintime = 10`

Set this parameter to raise an SQL error if the connection can not be established after the given number of seconds. The default is 5 seconds.

`dbi.database.dsname.ase.prefetch.rows`

Maximum number of rows to be pre-fetched.

`dbi.database.stores.ase.prefetch.rows = 50`

Use this parameter to increase performance by defining the maximum number of rows to be fetched into the db client buffer. However, the bigger this parameter is, the more memory is used by each program. The default is 10 rows.

**SAP HANA® specific FGLPROFILE parameters**

`dbi.database.dsname.hdb.schema`

Name of the database schema to be selected after connection is established.

`dbi.database.stores.hdb.schema = "store2"`

Set this parameter to a specific schema in order to share the same table with all users.

**SQL connection identifier**

Database client programs can be identified by name with some database server types.

**Purpose of SQL connection identifiers**

When connecting to databases such as PostgreSQL or SQL Server, it is possible to define a name for the database client program.

This allows for the association of a name with a database client connection in the database server, to be used in logging and trace utilities.

For example, if you don't define the application identifier for SQL Server client programs, all connections will get the name "ODBC".

**SQL application identifier with SQL Server**

The SQL Server client application identifier can be defined by using the `APP=\textit{name}` ODBC connection string parameter. The ODBC application identifier will be available in the SQL Server session with `sp_who` and `APP_NAME()` SQL functions. This name will also be visible in the "ApplicationName" column of SQL Profiler trace logs.

To pass ODBC connection string parameters to SQL Server, use the `?\textit{options}` notation in the "source" connection parameter.
With an FGLPROFILE entry:

```
dbi.database.mydb.source = "mydatasource?APP=myappid;"
```

At runtime with connection parameters in database specification:

```
CONNECT TO "mydb+source='mydatasource?APP=myappid;'
```

**Note:** ODBC connection string parameters must be terminated by a ; semi-colon.

### SQL application identifier with PostgreSQL

The PostgreSQL client application identifier can be defined with the `application_name` PostgreSQL URL connection string parameter.

To pass URL connection string parameters to PostgreSQL, use the `?options` notation in the "source" connection parameter:

**With an FGLPROFILE entry:**

```
dbi.database.mydb.source = "mydatabase?application_name=myappid"
```

**At runtime with connection parameters in database specification:**

```
CONNECT TO "mydb+source='mydatabase?application_name=myappid'" USER un USING up
```

**Note:** PostgreSQL URL connection string parameters must be separated by a & ampersand.

### Related concepts

- [Database source specification (source)](page-661)
- [Connection parameters in database specification](page-665)

Connection parameters can be provided in the database specification string passed to the DATABASE and CONNECT TO instructions.

### Database user authentication

Different database user authentication methods exist.

Connecting to a database server is not just specifying a database name: the current user must be identified by the database server. Database users must be declared in the database server and must be authenticated.

The typical user authentication is done by passing a login name and password at connection time. Some database servers support external authentication methods, that do not require login/password information (for example when db users are based on operating system users), as well as delegated user authentication via credential tokens (for example, when using an LDAP distinguished name). See database vendor specific documentation for more details.

Additional user authentication solutions are provided to simplify migration from IBM® Informix® databases, but are not recommended in production for security reasons.

See also SQL adaptation guides for database vendor specific notes regarding user authentication.

### Specifying a user name and password with CONNECT

In order to specify a user name and password, use the `CONNECT` instruction with the `USER/USING` clause:

```
MAIN
DEFINE uname, upswd STRING
CALL login_dialog() RETURNING uname, upswd
CONNECT TO "stock" USER uname USING upswd
...
END MAIN
```
This is the recommended way to connect to a database server.

With some database types, it is possible to use an external user authentication service, such as Kerberos, SSL/TLS, LDAP-based directory services. To connect as an external user, configure database client settings to authenticate the external user and perform the CONNECT TO instruction without specifying a login/password:

```
CONNECT TO "stock"
```

For more details, see for example database user handling in the Oracle SQL Adaptation Guide.

**Specifying a user name and password with DATABASE**

The `DATABASE` instruction does not support the `USER/USING` clause as `CONNECT TO` does. If you don't use an automatic user authentication method of the database server, you must provide a user name and password in some way.

The best way to identify database users is to replace every `DATABASE` instruction by a `CONNECT TO` with `USER/USING` clause. However, it is also possible to provide the user name and password with the user authentication callback function, by defining a global FGLPROFILE entry.

In a development environment, a default login and password can be specified with the `dbi.database.dbname.username` and `dbi.database.dbname.password` FGLPROFILE entries. This solution must not be used in a production environment because the password is not encrypted. For backward compatibility reasons, when using the IBM® Informix® driver, these FGLPROFILE entries are ignored by the `DATABASE` instruction, only the `CONNECT TO` instruction takes external (or callback) login parameters into account.

Login parameters can also be provided in the connection string used in the database name specification in `DATABASE` instruction.

**Related concepts**

- [Indirect database specification method](#) on page 667
- Genero BDL allows to define database connection parameters in FGLPROFILE, that can be referenced by a single identifier in programs.
- [User authentication callback function](#) on page 680

**User authentication callback function**

When using the `DATABASE` connection instruction, the database user name and password can be provided at runtime by a callback function. The callback function must be defined with the `dbi.default.userauth.callback` FGLPROFILE entry:

```

```
```
```
```
```
```
```

This callback method is provided when a lot of programs use the `DATABASE` instruction, and database user credentials are mandatory. If possible, use the `CONNECT TO` instruction with the `USER/USING` clause instead of `DATABASE`.

**Note:** With the IBM® Informix® driver, when using the `DATABASE` instruction, the callback method is invoked, but the user name and password returned by the function are ignored: Only `CONNECT TO` will take the login parameters into account for IBM® Informix®, when no `USER/USING` clause is specified.

The callback function must have the following signature:

```
CALL function-name(dbspec STRING) 
    RETURNS ( STRING, -- username 
      STRING  -- password 
    )
```

If you do not specify the module name, the callback function must be linked to the 42r program. By using the "module-name.function-name" syntax in the FGLPROFILE entry, the runtime system will automatically load the
module. In both cases, the module must be located in a directory where the runtime system can find it, defined by the FGLLDPATH environment variable.

In the callback function body, the value of dbspec can be used to identify the database source, read user name and encrypted password from FGLPROFILE entries with the fgl_getResource() function, then decrypt password with the algorithm of your choice and return user name and decrypted password.

**User authentication callback function for DATABASE:**

```sql
FUNCTION getUserAuth(dbspec STRING) RETURNS (STRING,STRING)  
DEF un, ep STRING  
LET un = fgl_getResource("dbi.database."||dbspec||".username")  
LET ep = fgl_getResource("dbi.database."||dbspec||".password.encrypted")  
RETURN un, decrypt_user_password(dbspec, un, ep)  
END FUNCTION
```

**Order of precedence for database user specification**

Database user login can be specified with different methods, as show in this table. Precedence order if defined from top to bottom:

**Table 196: Database user login methods**

<table>
<thead>
<tr>
<th>Connection Instruction</th>
<th>FGLPROFILE</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONNECT TO &quot;dbname&quot; USER &quot;user&quot; USING &quot;pswd&quot;</td>
<td>N/A (ignored)</td>
<td>The user information in the USER/USING clause of the CONNECT TO instruction or in the connection string of the DATABASE instruction are used to identify the actual user. are used to identify the actual user.</td>
</tr>
<tr>
<td>or</td>
<td></td>
<td>Connection string can also be used with CONNECT TO.</td>
</tr>
<tr>
<td>DEFINE db VARCHAR(200)</td>
<td></td>
<td>No user login and password is provided to the database server. Usually, the Operating System authentication takes place.</td>
</tr>
<tr>
<td>LET db = &quot;dbname&quot; +username='username', password='pswd' DATABASE db</td>
<td></td>
<td>Callback function fx is called to get user name and password when connection instruction is executed.</td>
</tr>
<tr>
<td>DATABASE dbname</td>
<td></td>
<td>The FGLPROFILE default user name and password are used to connect to the database server.</td>
</tr>
<tr>
<td>or</td>
<td></td>
<td>Important: NOT RECOMMENDED IN PRODUCTION!</td>
</tr>
<tr>
<td>CONNECT TO &quot;dbname&quot;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Unique session mode connection instructions**

Opening and closing a database for a unique session.

In unique-session mode, the `DATABASE` instruction initiates a connection the database server and creates the current session. The database connection is terminated with the `CLOSE DATABASE` instruction, or when another `DATABASE` instruction is executed, or when the program ends.

**DATABASE**

Opens a new database connection in unique-session mode.

**Syntax**

```
DATABASE { dbname \[dbserver\] \[variable\] \[string\] } \[EXCLUSIVE\]
```

1. `dbname` identifies the database name.
2. `dbserver` identifies the IBM® Informix® database server (INFORMIXSERVER).
3. `variable` can be any character string defined variable containing the database specification.
4. `string` can be a string literal containing the database specification.

**Usage**

The `DATABASE` instruction opens a connection to the database server, like `CONNECT TO`, but without user and password specification.

```
MAIN
    DATABASE stores
    ...
END MAIN
```

It is possible to use a program variable containing the database specification.

```
MAIN
    DEFINE dbname VARCHAR(100)
    LET dbname = arg_val(1)
    DATABASE dbname
    ...
END MAIN
```

If a current connection exists, it is automatically closed before connecting to the new database.

The connection is closed with the `CLOSE DATABASE` instruction, or when the program ends.

When using the `DATABASE` instruction, the database user is typically authenticated through the operating system user. If needed, the database user name and password can be provided from a callback function, with the `dbi.default.userauth.callback` FGLPROFILE entry.

The `DATABASE` instruction raises an error if the connection can not be established, for example, if you specify a database that the runtime system cannot locate, or cannot open, or for which the user of your program does not have access privileges.

The `EXCLUSIVE` keyword can be used to open an IBM® Informix® database in exclusive mode to prevent access by anyone but the current user. This keyword is IBM® Informix® specific and is not recommended when writing a portable SQL application.

The `CONNECT TO` instructions allow better control over database connections; it is recommended that you use these instructions instead of `DATABASE` and `CLOSE DATABASE`.

**Note:** When used outside a program block, the `DATABASE` instruction defines the database schema for compilation (for `DEFINE LIKE`). Furthermore, when `DATABASE` is used before a `MAIN` block, it defines also an implicit database connection. To specify the database schema for compilation, consider using `SCHEMA` instead.
Related concepts

Opening a database connection on page 657
A database connection identifies the SQL database server and the database entity the program connects to, in order to execute SQL statements.

CLOSE DATABASE
Closes the current database connection created by a DATABASE instruction.

Syntax

```
CLOSE DATABASE
```

Usage

The CLOSE DATABASE instruction closes the current database connection opened by a DATABASE instruction.

The current connection is automatically closed when the program ends.

Example

```
MAIN
  DATABASE stores1
  CLOSE DATABASE
  DATABASE stores2
  CLOSE DATABASE
END MAINs
```

Multi-session mode connection instructions

Opening and closing a database for a unique session.

In multi-session mode, open a database session with the CONNECT TO instruction. Other connections can be created with subsequent CONNECT TO instructions. To switch to a specific session, use the SET CONNECTION instruction; this suspends other opened connections. Disconnect from a specific or from all sessions with the DISCONNECT instruction. The end of the program disconnects all sessions automatically.

CONNECT TO
Opens a new database session in multi-session mode.

Syntax

```
CONNECT TO { dbname | DEFAULT } [ AS session ]
  [ USER login USING auth | TRUSTED ]
  [ WITH CONCURRENT TRANSACTION ]
```

1. `dbname` is the database specification.
2. `session` identifies the database session. By default, it is `dbname`.
3. `login` is the name of the database user.
4. `auth` is the password to authenticate the database user, or an external authentication token.

Usage

The CONNECT TO instruction opens a database connection. If the instruction successfully connects to the database environment, the connection becomes the current database session for the program.

Note: The DEFAULT clause of CONNECT TO is specific to IBM® Informix®.

The session name is case-sensitive.
A program can connect to several database environments at the same time (using different database drivers), and it can establish multiple connections to the same database environment, provided each connection has a unique connection name.

The connection is closed with the **DISCONNECT** instruction, or when the program ends.

When the USER `login` USING `auth` clause is specified, the database user is identified by `login` and `auth`, ignoring all other user settings defined in FGLPROFILE or as connection string parameters.

The `auth` parameter can be a simple password for internal database users, but for some types of database engines, it can be used to specify an external authentication token, such as a distinguished name (DN). For more details, see the SQL adaptation guide for your database type.

The `TRUSTED` keyword can be used after the USER/USING clause, to indicate that the specified user is trusted by an IBM® Informix® trusted context. When a trusted connection is established, a program can use the `SET SESSION AUTHORIZATION` instruction, to switch between database users associated to the connection user, as defined by the related trusted context. See Informix database server documentation for more details.

**Note:** The `TRUSTED` clause of `CONNECT TO` is specific to IBM® Informix®.

The `CONNECT TO` instruction raises an **exception** if the connection can not be established, for example, if you specify a database that the runtime system cannot locate, or cannot open, or for which the user of your program does not have access privileges.

The WITH CONCURRENT TRANSACTION clause allows a program to open several transactions concurrently in different database sessions: The transaction can be started with the `BEGIN WORK` statement in a given connection context, then the program can switch to another connection with `SET CONNECTION`, and when done, switch back to the first connection to issue a `COMMIT WORK` or `ROLLBACK WORK`. This is supported for IBM® Informix® database servers.

**Note:** The WITH CONCURRENT TRANSACTION option is ignored with non-Informix database server types, but it can be used in the CONNECT TO statement, for consistency with Informix.

A `CONNECT TO` statement cannot be executed with dynamic SQL (**PREPARE** + **EXECUTE**).

With IBM® Informix® database servers, when using the `CONNECT TO DEFAULT`, you connect to the default IBM® Informix® database server, identified by the INFORMIXSERVER environment variable, without any database selection.

When using IBM® Informix® on UNIX™, the only restriction on establishing multiple connections to the same database environment is that an program can establish only one connection to each local server that uses the shared-memory connection mechanism. To find out whether a local server uses the shared-memory connection mechanism or the local-loopback connection mechanism, examine the `$INFORMIXDIR/etc/sqlhosts` file.

**Example**

```sql
MAIN
  DEFINE uname, upswd VARCHAR(50)
  CONNECT TO "stores1" -- Session name is "stores1"
  CONNECT TO "stores1" AS "SA" -- Session name is "SA"
  CALL login_dialog() RETURNING uname, upswd
  CONNECT TO "stores2" AS "SB" USER uname USING upswd
END MAIN
```

**Related concepts**

[Connection parameters in database specification](#) on page 665

Connection parameters can be provided in the database specification string passed to the DATABASE and CONNECT TO instructions.

[Opening a database connection](#) on page 657
A database connection identifies the SQL database server and the database entity the program connects to, in order to execute SQL statements.

**SET CONNECTION**

Selects the current session when in multi-session mode.

**Syntax**

```
SET CONNECTION { session | DEFAULT | current DORMANT }
```

1. **session** is a string expression identifying the name of the database session to be set as current.

**Usage**

The **SET CONNECTION** instruction makes a given connection current.

The session name is case-sensitive.

When using the **DEFAULT** keyword, it identifies the default database server connection established with a **CONNECT TO DEFAULT** or a **DATABASE** instruction.

**Note:** The **DEFAULT** clause of **SET CONNECTION** is specific to IBM® Informix®.

To make the current connection dormant, use **CURRENT DORMANT** keyword.

**Note:** The **DORMANT** clause of **SET CONNECTION** is specific to IBM® Informix®.

A **SET CONNECTION** statement cannot be executed with dynamic SQL (**PREPARE** + **EXECUTE**).

**Example**

```
MAIN
    DEFINE c1, c2, c3 INT
    CONNECT TO "stores1"
    CONNECT TO "stores2" AS "SA"
    CONNECT TO "stores3" AS "SB"
    SET CONNECTION "stores1"    -- Select first session
    SELECT COUNT(*) INTO c1 FROM customers
    SET CONNECTION "SA"         -- Select second session
    SELECT COUNT(*) INTO c2 FROM customers
    SET CONNECTION "SB"         -- Select third session
    SELECT COUNT(*) INTO c3 FROM customers
    SET CONNECTION "stores1"    -- Select first session again
END MAIN
```

**Related concepts**

**CONNECT TO** on page 683

Opens a new database session in multi-session mode.

**Dynamic SQL management** on page 709

Explains how to execute and manage SQL statements at runtime.

**SET SESSION AUTHORIZATION**

Select the user under which database operations are performed in the current connection.

**Syntax**

```
SET SESSION AUTHORIZATION TO login USING auth
```

1. **login** is the name of the database user.
2. *auth* is the password to authenticate the database user.

**Usage**

The `SET SESSION AUTHORIZATION` instruction switches to the specified database user in the current database connection. It allows a user to assume the identity of another database user.

**Note:** The `SET SESSION AUTHORIZATION` instruction is specific to IBM® Informix®.

If the current database connection is trusted (it was established with a `CONNECT TO` using the TRUSTED keyword), the `SET SESSION AUTHORIZATION` instruction can be used to switch to a database user associated to the trusted context of the user specified in `CONNECT TO`. The password is optional, if the trusted context defines the user association is defined WITHOUT AUTHENTICATION.

If the current database connection is not established for a trusted context, the `SET SESSION AUTHORIZATION` can be used to switch to another database user, with an optional password.

In both cases, the user specified in `CONNECT TO` must have all required database privileges to switch to the user specified by the `SET SESSION AUTHORIZATION` statement.

**Tip:** Informix database trusted contexts and trusted connections are typically used within a three-tier application model, to increase overall system performances by reusing the current database connection: After a trusted connection is established, the middle-tier application can switch to different database users without reconnecting. Any SQL operation is done on behalf of the current database user. The application can then benefit from database server features such as SQL auditing and user privileges, which are based on database users.

See Informix database documentation for more details about `SET SESSION AUTHORIZATION` and trusted contexts (CREATE TRUSTED CONTEXT).

**Example**

```plaintext
MAIN
  DEFINE db, un, pw STRING
  LET db = "stores"
  LET un = "mike"
  LET pw = "..."
  CONNECT TO db USER un USING pw TRUSTED
  LET un = "scott"
  SET SESSION AUTHORIZATION TO un
  LET un = "robert"
  SET SESSION AUTHORIZATION TO un
END MAIN
```

**Related concepts**

`CONNECT TO` on page 683
Opens a new database session in multi-session mode.

**DISCONNECT**
Terminates database sessions when in multi-session mode.

**Syntax**

```
DISCONNECT  
  | ALL  | DEFAULT  | CURRENT  | session  |
```

1. *session* is a string expression identifying the name of the database session to be terminated.

**Usage**

The `DISCONNECT` instruction closes a given database connection opened with a `CONNECT TO` instruction.

The session name is case-sensitive.
When using the `DEFAULT` keyword, it identifies the default database server connection established with a `CONNECT TO DEFAULT` or a `DATABASE` instruction.

**Note:** The `DEFAULT` clause of `DISCONNECT` is specific to IBM® Informix®.

Use the `ALL` keyword to terminate all opened connections. From that point, you must establish a new connection to execute SQL statements.

Use the `CURRENT` keyword to terminate the current connection only. From that point, in order to execute SQL statements, you must select another connection with `SET CONNECTION`, or establish a new connection with `CONNECT TO`.

A `DISCONNECT` statement cannot be executed with dynamic SQL (i.e. `PREPARE + EXECUTE`).

If a `DISCONNECT` statement is used while a database transaction is active, the transaction is automatically rolled back.

**Example**

```sql
MAIN
    CONNECT TO "stores1"  -- Will be identified by "stores1"
    CONNECT TO "stores1" AS "SA"
    CONNECT TO "stores2" AS "SB" USER "scott" USING "tiger"
    DISCONNECT "stores1"
    DISCONNECT "SB"
    SET CONNECTION "SA"
END MAIN
```

**Related concepts**

- [Database transactions](#) on page 688
  Database transaction concepts and handling.

**Miscellaneous SQL statements**

These are particular SQL statements supported in the static SQL syntax.

**SET EXPLAIN**

Turns on/off SQL report of the optimizer plan.

**Syntax:**

```sql
SET EXPLAIN { ON | OFF }
```

**Usage:**

**Important:** The `SET EXPLAIN` SQL instruction is specific to IBM® Informix®.

**UPDATE STATISTICS**

Updates the statistics for all or for the specified table in the database.

**Syntax:**

```sql
UPDATE STATISTICS [ FOR TABLE table-specification ]
```

**Usage:**

**Important:** The `UPDATE STATISTICS` SQL instruction is specific to IBM® Informix®.
Database transactions

Database transaction concepts and handling.

Understanding database transactions

This is an introduction to database transactions.

A database transaction delimits a set of database operations (SQL statements), that are processed as a whole.

Database operations included inside a transaction are validated or canceled as a unique operation.

Figure 25: Database transaction

The database server is in charge of data concurrency and data consistency. Data concurrency allows for the simultaneous access of the same data by many users, while data consistency gives each user a consistent view of the database.

Without adequate concurrency and consistency control, data can be changed improperly, compromising integrity of your database. If you want to write applications that can work with different kinds of database servers, you must adapt the program logic to the behavior of the database servers, regarding concurrency and consistency management. This requires good knowledge of multiuser database application programming, transactions, locking mechanisms, isolation levels, and wait mode. If you are not familiar with these concepts, carefully read the documentation of each database server that covers this subject.

Usually, database servers set exclusive locks on rows that are modified or deleted inside a transaction. These locks are held until the end of the transaction to control concurrent access to that data. Some database servers implement row versioning (before modifying a row, the server makes a copy of the original row). This technique allows readers to see a consistent copy of the rows that are updated during a transaction not yet committed. When the isolation level is high (REPEATABLE READ) or when using a SELECT FOR UPDATE statement, the database server sets shared locks on fetched rows, to prevent other users from changing the rows fetched by the reader. These locks are held until the end of the transaction. Some database servers allow read locks to be held regardless of the transactions (WITH HOLD cursor option), but this is not a standard.

Programs accessing the database can change transaction parameters such as the isolation level or lock wait mode. To write portable applications, you must use a configuration that produces the same behavior on every database engine.

The recommended programming pattern regarding transactions is the following:

- The database must support transactions; this is usually the case.
- Transactions must be as short as possible (a few seconds).
• The isolation level must be at least **COMMITTED READ**.
• The wait mode for locks must be **WAIT** or **WAIT n** (lock timeout).

To write portable SQL applications, programmers use the **BEGIN WORK**, **COMMIT WORK** and **ROLLBACK WORK** instructions described in this section to delimit transaction blocks and define concurrency parameters with **SET ISOLATION** and **SET LOCK MODE**. These instructions are part of the language syntax. At runtime, the database driver generates the appropriate SQL commands to be used with the target database server. This allows you to use the same source code for different kinds of database servers.

If you initiate a transaction with a **BEGIN WORK** statement, you must issue a **COMMIT WORK** at the end of the transaction. If one of the SQL statement fails in the transaction, you typically issue a **ROLLBACK WORK** to force the database server to cancel any modifications that the transaction made to the database. If you do not issue a **BEGIN WORK** statement to start a transaction, each statement executes within its own transaction. These single-statement transactions do not require either a **BEGIN WORK** statement or a **COMMIT WORK** statement.

Recent database engines support transaction savepoints, which allows you to set markers in the current transaction, in order to rollback to a specific point without canceling the complete transaction. The transaction savepoint instructions **SAVEPOINT**, **ROLLBACK TO SAVEPOINT** and **RELEASE SAVEPOINT** are part of the language syntax and can be directly used in the code.

Some database servers do not support a Data Definition Language (DDL) statements (like **CREATE TABLE**) inside transactions, and some automatically commit the transaction when such a statement is executed. Therefore, it is strongly recommended that you avoid DDL statements inside transactions.

A transaction that processes many rows can exceed the limits that your operating system or the database server configuration imposes on the maximum number of simultaneous locks. Include a limited number of SQL operations in a transaction blocks.

When a program is using several database connections, and if transactions are not terminated before switching to another connection (**SET CONNECTION**), it is mandatory to use the **WITH CONCURRENT TRANSACTION** option in the **CONNECT** instruction.

**Related concepts**
- **Database connections** on page 656
  Explains how to manage database connections in a program.
- **Concurrent data access** on page 599
  Understanding concurrent data access and data consistency.
- **CONNECT TO** on page 683
  Opens a new database session in multi-session mode.

**BEGIN WORK**

Starts a database transaction in the current connection.

**Syntax**

```
BEGIN WORK
```

**Usage**

Use the **BEGIN WORK** instruction to indicate where the database transaction starts in your program. Each row that an **UPDATE**, **DELETE**, or **INSERT** statement affects during a transaction is locked and remains locked throughout the transaction.

**BEGIN WORK** is part of the language syntax, the underlying database driver executes the native SQL statement corresponding to this SQL instruction.

The transaction block is ended by a **COMMIT WORK** or **ROLLBACK WORK** instruction.
Example
The next code example starts a transaction block, inserts a row and updates the row, then commits the transaction. To other users, the INSERT and UPDATE instruction will be seen as an single atomic database modification:

```sql
MAIN
  DATABASE stock
  BEGIN WORK
  INSERT INTO items VALUES ( ... )
  UPDATE items SET ...
  COMMIT WORK
END MAIN
```

Related concepts
Understanding database transactions on page 688
This is an introduction to database transactions.

SAVEPOINT
Defines or resets the position of a rollback point in the current transaction.

Syntax

```
SAVEPOINT spVname [UNIQUE]
```

1. `spname` is the savepoint identifier.

Usage
The SAVEPOINT instruction declares a new rollback label at the current position in the lexical order within the current transaction. After defining a savepoint, you can rollback to the specified point in the transaction by using the ROLLBACK WORK TO SAVEPOINT instruction.

If the same savepoint name was used in a prior SAVEPOINT instruction, the previous savepoint is destroyed and the name is reused to flag the new rollback position. The optional UNIQUE keyword specifies that you do not want to reuse the same savepoint name in a subsequent SAVEPOINT instruction. Reusing the same name after a SAVEPOINT `spname` UNIQUE will raise an SQL error.

Example
In this example, a first savepoint is defined before the INSERT statement, then reset before the UPDATE statement. The ROLLBACK TO SAVEPOINT instruction will cancel the UPDATE statement only:

```sql
MAIN
  DATABASE stock
  BEGIN WORK
  DELETE FROM items
  SAVEPOINT spl
  INSERT INTO items VALUES ( ... )
  SAVEPOINT spl -- releases previous savepoint named spl
  UPDATE items SET ...
  ROLLBACK WORK TO SAVEPOINT spl
  COMMIT WORK
END MAIN
```

Related concepts
RELEASE SAVEPOINT on page 692
Destroys the specified savepoint in the current transaction.

ROLLBACK WORK on page 691
Cancels and terminates a database transaction in the current connection.

**COMMIT WORK**
Validates and terminates a database transaction in the current connection.

**Syntax**

```sql
COMMIT WORK
```

**Usage**

Use the `COMMIT WORK` instruction to commit all modifications made to the database from the beginning of a transaction started with `BEGIN WORK`. The database server takes the required steps to make sure that all modifications that the transaction makes are completed correctly and saved to disk.

`COMMIT WORK` is part of the language syntax, the underlying database driver executes the native SQL statement corresponding to this SQL instruction.

The `COMMIT WORK` statement releases all exclusive locks that have been set during the transaction. With some databases, shared locks are not released if the `FOR UPDATE` cursor is declared with the `WITH HOLD` option. However, the `COMMIT WORK` statement closes all cursors not declared with the `WITH HOLD` option.

**Related concepts**

Understanding database transactions on page 688
This is an introduction to database transactions.

**ROLLBACK WORK**
Cancels and terminates a database transaction in the current connection.

**Syntax**

```sql
ROLLBACK WORK [ TO SAVEPOINT [spname] ]
```

- `spname` is the savepoint identifier.

**Usage**

Use `ROLLBACK WORK` to cancel the current transaction and invalidate all changes since the beginning of the transaction started with `BEGIN WORK`. After the execution of this instruction, the database is restored to the state that it was in before the transaction began. All row and table locks that the canceled transaction holds are released. If you issue this statement when no transaction is pending, an error occurs.

`ROLLBACK WORK` is part of the language syntax, the underlying database driver executes the native SQL statement corresponding to this SQL instruction.

When specifying a `savepoint` with the `TO SAVEPOINT` clause, all SQL statements executed since the specified savepoint will be canceled. The transaction is not canceled, however, and you can continue to execute other SQL statements.

**Example**

This example checks for a potential SQL error after the `DELETE` statement and cancels the complete transaction with a `ROLLBACK` instruction:

```sql
MAIN
   DATABASE stock
   WHENEVER ERROR CONTINUE
   BEGIN WORK
   INSERT INTO orders_hist VALUES ( ... )
```
DELETE FROM orders WHERE ...
IF SQLCA.SQLCODE < 0 THEN
  ROLLBACK WORK
ELSE
  COMMIT WORK
END IF
END MAIN

Related concepts
Understanding database transactions on page 688
This is an introduction to database transactions.

RELEASE SAVEPOINT
Destroys the specified savepoint in the current transaction.

Syntax

```
RELEASE SAVEPOINT spname
```

- `spname` is the savepoint identifier.

Usage
Use the `RELEASE SAVEPOINT` instruction to delete a savepoint defined by the `SAVEPOINT` instruction.

See database documentation for more details about the behavior of this SQL statement.

Note for example that IBM® Informix® IDS will also release any savepoint that has been declared between the specified savepoint and the `RELEASE SAVEPOINT` instruction.

Example
In the next example, the `RELEASE SAVEPOINT` instruction cancels the `UPDATE` and `INSERT` statements and destroys the `sp1` and `sp2` savepoints. Only the `DELETE` statement will take effect at the end of the transaction:

```
MAIN
  DATABASE stock
  BEGIN WORK
  DELETE FROM items
  SAVEPOINT sp1
  INSERT INTO items VALUES ( ... )
  SAVEPOINT sp2
  UPDATE items SET ...
  RELEASE SAVEPOINT sp1
  ROLLBACK WORK TO SAVEPOINT
  COMMIT WORK
END MAIN
```

Related concepts
Understanding database transactions on page 688
This is an introduction to database transactions.

SET ISOLATION
Defines the transaction isolation level for the current connection.

Syntax

```
SET ISOLATION TO
  ↓ DIRTY READ
```
Usage

The `SET ISOLATION` instruction sets the transaction isolation level for the current connection. See database concepts in your database server documentation for more details about isolation levels and concurrency management.

When possible, the underlying database driver sets the corresponding transaction isolation level. If the isolation level cannot be set, the runtime system generates an exception.

When using the `DIRTY READ` isolation level, the database server might return a phantom row, which is an uncommitted row that was inserted or modified within a transaction that has subsequently rolled back. No other isolation level allows access to a phantom row.

On most database servers, the default isolation level is `COMMITTED READ`, which is appropriate to portable database programming.

The `LAST COMMITTED` and `REtain Update LOCKs` options have been added to the language syntax for conformance with IBM® Informix® IDS 11. The `LAST COMMITTED` option can be turned on implicitly with a server configuration parameter, saving unnecessary code changes.

Example

```sql
MAIN
  DATABASE stock
  SET ISOLATION TO COMMITTED READ
  ...
END MAIN
```

Related concepts

- Concurrent data access on page 599
  Understanding concurrent data access and data consistency.

- SQL execution diagnostics on page 591
  If an SQL statement execution fails, error description can be found in the `SQLCA.SQLCODE`, `SQLSTATE`, `STATUS` and `SQLERRMESSAGE` predefined registers.

SET LOCK MODE

Defines the behavior of the program that tries to access a locked row or table.

Syntax

```
SET LOCK MODE TO { NOT WAIT | WAIT [ seconds ] }
```

1. `seconds` defines the number of seconds to wait for lock acquisition.

Usage

The `SET LOCK MODE` instruction defines the timeout for lock acquisition for the current connection.

When possible, the underlying database driver sets the corresponding connection parameter to define the timeout for lock acquisition. But some database servers may not support setting the lock timeout parameter. In this case, the runtime system generates an exception.

When using the `NOT WAIT` clause, the timeout is set to zero. If the resource is locked, the database server ends the operation immediately and raises an exception with the SQL error.

When using the `WAIT` clause with a number of seconds, if the resource is locked, the database server waits for the specified seconds, and then raises an exception with an SQL error.
When using the `WAIT` clause without a number of seconds, the database server waits for lock acquisition for an infinite time.

With most database servers, the default is to wait for locks to be released.

Make sure that the database server and corresponding database driver both support a lock acquisition timeout option, otherwise the program will raise an exception.

**Example**

```main
MAIN
  DATABASE stock
  SET LOCK MODE TO WAIT 20
  ...
END MAIN
```

**Related concepts**

- [Concurrent data access](#) on page 599
  Understanding concurrent data access and data consistency.

- [SQL execution diagnostics](#) on page 591
  If an SQL statement execution fails, error description can be found in the `SQLCA.SQLCODE`, `SQLSTATE`, `STATUS` and `SQLERRMESSAGE` predefined registers.

---

**Static SQL statements**

Describes static SQL statements supported in the language.

**Understanding static SQL statements**

This is an introduction to static SQL statements.

*Static SQL statements* are SQL instructions that are a part of the Genero BDL language syntax. Static SQL statements can be used directly in the source code as a normal procedural instruction. The static SQL statements are parsed and validated at compile time. At runtime, these SQL statements are automatically prepared and executed by the runtime system.

Program variables can be used inside static SQL statements. Variables are detected by the compiler and handled as SQL parameters at runtime.

The following example defines two variables that are directly used in an `INSERT` statement:

```main
MAIN
  DEFINE iref INTEGER, name CHAR(10)
  DATABASE stock
  LET iref = 65345
  LET name = "Kartopia"
  INSERT INTO item (item_ref, item_name) VALUES (iref, name)
  SELECT item_name INTO name
  FROM item WHERE item_ref = iref
END MAIN
```

As it is integrated in the language syntax, static SQL statement usage clarifies the source code, but the SQL text is hard-coded and cannot be modified at runtime as is possible with `PREPARE / EXECUTE` instructions of dynamic SQL.

Limited SQL syntax is part of the language, only common SQL statements such as `INSERT, UPDATE, DELETE, SELECT` are supported.

The compiler supports also `SQL ... END SQL` blocks to write free SQL text in your programs. The SQL syntax in SQL blocks is not limited to the static SQL syntax.
Related concepts

**Database connections** on page 656

Explains how to manage database connections in a program.

**Using program variables in static SQL**

Static SQL syntax supports the usage of program variables as SQL parameters.

Using **program variables** directly in static SQL statements gives a better understanding of the source code and requires less lines as when using SQL parameters in dynamic SQL statements.

```plaintext
MAIN
  DEFINE c_num INTEGER
  DEFINE c_name CHAR(10)
  DATABASE stock
  SELECT cust_name INTO c_name FROM customer WHERE cust_num = c_num
END MAIN
```

If a database table name or column name conflicts with a program variable, you can use the `@` sign as the column prefix. The compiler will treat the identifier following the `@` as an SQL object name:

```plaintext
MAIN
  DEFINE cust_name CHAR(10)
  DEFINE cnt INTEGER
  DATABASE stock
  SELECT COUNT(*) INTO cnt FROM customer WHERE @cust_name = cust_name
END MAIN
```

The `@` sign will not figure in the resulting SQL statement stored in the `.42m` compiled module.

A database object name may also conflict with another symbol of the program code, such as a module name. In this case, you can also use the `@` sign before the conflicting database object name to solve the issue.

In the next example, the imported module name conflicts with a database table name "account":

```plaintext
IMPORT FGL account
MAIN
  DEFINE rec RECORD
    pkey INTEGER,
    name VARCHAR(50)
  END RECORD
  SELECT * INTO rec.* FROM account
    WHERE @account.pkey = rec.pkey
END MAIN
```

Without the `@` sign, the compiler would produce the following error, expecting that `account.pkey` is a public variable of the imported `account` module:

```plaintext
IMPORT FGL account
MAIN
  DEFINE rec RECORD
    pkey INTEGER,
    name VARCHAR(50)
  END RECORD
  SELECT * INTO rec.* FROM account
    WHERE account.pkey = rec.pkey
```

The symbol 'pkey' does not represent a defined variable.

See error number -4369.

END MAIN

**Related concepts**

**Dynamic SQL management** on page 709
Table and column names in static SQL

How are SQL object names and keywords converted in static SQL?

In static SQL statements, table and column names will be converted to lowercase by the fglcomp compiler. The SQL keywords are always converted to uppercase.

For example:

```
UPDATE CUSTOMER set CUST_name = 'undef' WHERE cust_name is null
```

Will be converted to:

```
UPDATE customer SET cust_name = 'undef' WHERE cust_name IS NULL
```

While SQL keywords are not case sensitive for database servers, table names and column names can be case-sensitive.

You can dump the static SQL statement texts with the -S option of fglcomp.

**Related concepts**

SQL basics on page 590
This section contains fundamental information to know about SQL programming with Genero BDL.

SQL texts generated by the compiler

The Genero BDL compiler provides an option to extract static SQL statements from .4gl sources.

The fglcomp compiler parses the static SQL statements and modifies them before writing the resulting SQL text to the .42m module.

You can extract all static SQL statements from the source by using the -S option of fglcomp:

**Example**

```
MAIN
  DEFINE c_name CHAR(10)
  DEFINE cnt INTEGER
  DATABASE stock
  SELECT COUNT(*) INTO cnt FROM customer WHERE customer.cust_name = c_name
END MAIN
```

```
$ fglcomp -S test.4gl
test.4gl^5^SELECT COUNT(*) FROM customer WHERE cust_name = ?
```

**INSERT**

Creates a new row in a database table.

**Syntax 1:**

This is the most standard syntax, works with all types of database engines.

```
INSERT INTO table-specification ↓ ( column ↓,...↓ ) ↓ ↓ VALUES ( ↓ variable ↓ sql-expression ↓ ↓...
↓ ↓ select-statement
↓
```
Syntax 2:
The `fglcomp` compiler will automatically generate a standard INSERT statement with the complete list of members of the record. The generated SQL will depend on the definition of the record.

```sql
INSERT INTO table-specification VALUES (record.* )
```

Syntax 3:
This syntax requires a database schema specification with the `SCHEMA` instruction, and the corresponding database schema file.

```sql
INSERT INTO table-specification VALUES record.*
```

where `table-specification` is:

```
[dbname[@dbserver]:][owner.]table
```

1. `dbname` identifies the database name.
2. `dbserver` identifies the database server (INFORMIXSERVER).
3. `owner` identifies the owner of the table, with optional double quotes.
4. `table` is the name of the database table.
5. `column` is a name of a table column.
6. `variable` is a program variable, a record member or an array member used as a parameter buffer to provide values.
7. `sql-expression` is an expression supported by the database server, this can be a literal or `NULL` for example.
8. `select-statement` is a static `SELECT` statement with or without parameters as variables.
9. `record` is the name of a record (followed by dot star in this syntax).

Usage
The `INSERT` SQL statement can be used to create a row in a specified database table.

The `dbname`, `dbserver` and `owner` prefix of the table name should be avoided for maximum SQL portability.

With `INSERT INTO table VALUES ...`, the statement inserts a row in the table with the values specified in variables, as literals, or with `NULL`. When using a `record`, you can specify all record members with the dot-star notation (`record.*`).

The third syntax can be used to avoid serial column usage in the value list. The record member corresponding to a column defined as `SERIAL`, `SERIAL8` or `BIGSERIAL` in the schema file will be removed by the compiler. This is useful when using databases like Microsoft™ SQL Server, where `IDENTITY` columns must be omitted in `INSERT` statements.

When using a `select-statement`, the statement inserts all rows returned in the result set of the `SELECT` statement. The columns returned by the result set must match the column number and data types of the target table. For SQL portability, it is not recommended that you use this syntax.

Example

```plaintext
MAIN
    DEFINE myrec RECORD
        key INTEGER,
        name CHAR(10),
        cdate DATE,
        comment VARCHAR(50)
    END RECORD
    DATABASE stock
    LET myrec.key     = 123
    LET myrec.name    = "Katos"
    LET myrec.cdate   = TODAY
```
LET myrec.comment = "xxxxxx"
INSERT INTO items VALUES ( 123, 'Practal', NULL, myrec.comment )
INSERT INTO items VALUES ( myrec.* )
INSERT INTO items VALUES myrec.* -- without serial (if one is used)
INSERT INTO items SELECT * FROM histitems WHERE name = myrec.name
END MAIN

Related concepts
Dynamic SQL management on page 709
Explains how to execute and manage SQL statements at runtime.

UPDATE
Modifies rows of a database table.

Syntax 1:
This is the most standard syntax, working with all type of database engines.

```
UPDATE table-specification
SET column = \ variable \ sql-expression \,
    \ sql-condition \n```

Syntax 2:
This syntax is not standard, but will be converted by the compiler to a portable UPDATE syntax.

```
UPDATE table-specification
SET ( column \, \, )
    = ( \ variable \ sql-expression \, \, )
    sql-condition
```

Syntax 3:
This syntax is not portable, and is not converted by the compiler.

```
UPDATE table-specification
SET [table.]* = ( \ variable \ sql-expression \, \, )
    sql-condition
```

Syntax 4:
This syntax requires a database schema specification with SCHEMA instruction, and the corresponding database schema file.

```
UPDATE table-specification
SET [table.]* ( column \, \, )
    = record.*
    sql-condition
```

where table-specification is:

```
[dbname[@dbserver]:][owner.]table
```
And sql-condition is:

WHERE condition CURRENT OF cursor

1. dbname identifies the database name.
2. dbserver identifies the database server (INFORMIXSERVER).
3. owner identifies the owner of the table, with optional double quotes.
4. table is the name of the database table.
5. column is a name of a table column.
6. variable is a program variable, a record member or an array member used as a parameter buffer to provide values.
7. sql-expression is an expression supported by the database server, this can be a literal or NULL for example.
8. record is the name of a record (followed by dot star in this syntax).
9. condition is an SQL expression to select the rows to be updated.
10. cursor is the identifier of a database cursor.

Usage

The UPDATE SQL statement can be used to modify one or more rows in the specified database table.

It is recommended to avoid using the dbname, dbserver and owner prefix of the table name to ensure maximum SQL portability.

The third syntax is not standard and will not work with all database types. It is not recommended.

The fourth syntax can be used if the database schema file has been generated with the correct data types. This is especially important when using SERIAL columns or equivalent auto-incremented columns. The fglcomp compiler will automatically extend the SQL text with the columns identified by the record variable. The columns defined in the database schema file as SERIAL (code 262) will be omitted in the generated column list.

column with a subscript expression (column[a,b]) is not recommended because most database servers do not support this notation.

For more details about the WHERE CURRENT OF clause, see Positioned updates/deletes on page 726.

Example

MAIN
DEFINE myrec RECORD
    key INTEGER,
    name CHAR(10),
    cdate DATE,
    comment VARCHAR(50)
END RECORD
DATABASE stock
LET myrec.key     = 123
LET myrec.name    = "Katos"
LET myrec.cdate   = TODAY
LET myrec.comment = "xxxxxx"
UPDATE items SET
    name    = myrec.name,
    cdate   = myrec.cdate,
    comment = myrec.comment
WHERE key = myrec.key
END MAIN

Related concepts

Positioned updates/deletes on page 726
DELETES
Describes row modification based on a FOR UPDATE cursor.

DELETE
Removes rows from a database table.

Syntax

```
DELETE FROM table-specification
    WHERE \ WHERE \ condition \ WHERE CURRENT OF cursor \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \'options
```
GROUP BY column-list [HAVING condition] [ ORDER BY column [ASC|DESC]] [,...] 

where subset-clause is:

SKIP { integer | variable } [FIRST|MIDDLE|LIMIT] { integer | variable }

where duplicates-option is:

ALL
DISTINCT
UNIQUE

where select-list is:

[[@]table-specification.*
[table-specification.]column
[literal]
[@] column-alias ]
[,...]

where table-list is:

table-name
OUTER table-name
OUTER ( table-name [,..] )
[,...]

where table-name is:

table-specification [ [AS] table-alias]

where table-specification is:

[dbname[@dbserver]:][owner.]table

where column-list is:

column-name [,..]

where column-name is:

[table.]column

1. dbname identifies the database name.
2. dbserver identifies the database server (INFORMIXSERVER).
3. owner identifies the owner of the table, with optional double quotes.
4. table is the name of the database table.
5. table-alias defines a new name to reference the table in the rest of the statement.
6. integer is an integer constant.
7. variable is a program variable.
8. column is a name of a table column.
9. column-alias defines a new name to reference the column in the rest of the statement.
10. condition is an SQL expression to select the rows to be deleted.
Usage

It is recommended to avoid using the `dbname`, `dbserver` and `owner` prefix of the table name to ensure maximum SQL portability.

If the `SELECT` statement returns only one row of data, you can write it directly as a procedural instruction. However, you must use the `INTO` clause to provide the list of variables where column values will be fetched. The `INTO` clause provides the list of fetch buffers. This clause is not part of the SQL language sent to the database server; it is extracted from the statement by the compiler.

```plaintext
MAIN
  DEFINE myrec RECORD
    key INTEGER,
    name CHAR(10),
    cdate DATE,
    comment VARCHAR(50)
  END RECORD
DATABASE stock
LET myrec.key = 123
SELECT name, cdate
  INTO myrec.name, myrec.cdate
FROM items
WHERE key=myrec.key
END MAIN
```

If the `SELECT` statement returns more than one row of data, you must declare a database cursor to process the result set.

```plaintext
MAIN
  DEFINE myrec RECORD
    key INTEGER,
    name CHAR(10),
    cdate DATE,
    comment VARCHAR(50)
  END RECORD
DATABASE stock
LET myrec.key = 123
DECLARE c1 CURSOR FOR
  SELECT name, cdate
  FROM items
  WHERE key=myrec.key
OPEN c1
FETCH c1 INTO myrec.name, myrec.cdate
CLOSE c1
END MAIN
```

The `SELECT` statement can include the `INTO` clause, but it is strongly recommended that you use that clause in the `FETCH` instruction only.

The `SELECT INTO TEMP` statement creates temporary tables. Such a statement does not return a result set.

Related concepts

- [Result set processing](#) on page 715
  Shows how to fetch rows from a database query.

**SQL ... END SQL**

Performs an SQL that is not part of the static SQL syntax.

**Syntax**

```sql
SQL
```
where *sql-statement* is:

```
  sql-keyword
  | identifier
  | INTO $host-variable [, ...]
  | $host-variable
  | {+ sql-directive }
  | --+ sql-directive
  | --# fgl-comment
  [....]
```

1. *sql-keyword* is any keyword of the SQL language.
2. *identifier* is a regular SQL identifier such as a table or column name.
3. *host-variable* is a program variable defined in the current scope.
4. *sql-directive* is a special comment to be kept in the SQL statement.
5. *fgl-comment* defines a comment that will be interpreted as a regular syntax element.

**Usage**

SQL blocks provide a convenient way to execute specific SQL statements that are not supported in the language as static SQL statements.

SQL blocks start with the `SQL` keyword and end with the `END SQL` keywords. The content of the SQL block is parsed by the fglcomp compiler to extract host variables, but the SQL statement syntax is not checked. This is actually the main purpose of SQL blocks, compared to regular static SQL statements; with SQL blocks, you can use any recent SQL statement introduced by the latest version of your database server. Note, however, that you can achieve the same result using dynamic SQL instructions.

Only one SQL statement can be included in an SQL block. Using the `;` semicolon statement separator is forbidden.

Program variables can be used inside the SQL statement. However, unlike static SQL statements, each host variable must be identified with a `$` dollar prefix. The list of fetch targets must be preceded by the `INTO` keyword, as in static SELECT statements. Complete records can be used in SQL blocks by using the dot star notation (`$record.*`), you can also use the THROUGH or THRU keywords, as well as array elements.

SQL blocks can also be used to declare a cursor with the `DECLARE mycursor CURSOR FOR SQL ... END SQL` syntax.

SQL directives can be used inside SQL blocks as special comments with the `{+}` or `--+` syntax. The SQL directives will be kept in the SQL text that will be executed by the database server. You typically write optimizer hints with the SQL directives syntax.

The `--#` specific comment is supported for backward compatibility. The SQL text following this marker will be parsed as regular SQL text, but will be ignored by other compilers. It is not recommended to use this feature.

You can check the resulting SQL statement after parsing by using the `-S` option of fglcomp.

**Example**

```
MAIN
  DEFINE myrec RECORD
    key INTEGER,
    name CHAR(10)
  END RECORD
DATABASE stock
LET myrec.key = 123
SQL
  SELECT (+EXPLAIN) items.* INTO $myrec.*
  FROM items WHERE key=$myrec.key
```
CREATE SEQUENCE

Creates a new sequence object in the database.

Syntax:

```
CREATE SEQUENCE \[ IF NOT EXISTS \] sequence-name \[
  INCREMENT BY integer
  | START WITH integer
  | NOMAXVALUE
  | MAXVALUE integer
  | NOMINVALUE
  | MINVALUE integer
  | CYCLE
  | NOCYCLE
  | CACHE integer
  | NOCACHE
  | ORDER
  | NOORDER
```

Related concepts

ALTER SEQUENCE on page 704
Modifies the definition of an existing sequence in the database.

DROP SEQUENCE on page 705
Drops a sequence object from the database.

ALTER SEQUENCE

Modifies the definition of an existing sequence in the database.

Syntax:

```
ALTER SEQUENCE sequence-name \[
  INCREMENT BY integer
  | RESTART WITH integer
  | NOMAXVALUE
  | MAXVALUE integer
  | NOMINVALUE
  | MINVALUE integer
  | CYCLE
  | NOCYCLE
  | CACHE integer
  | NOCACHE
  | ORDER
  | NOORDER
```

Related concepts

CREATE SEQUENCE on page 704
Creates a new sequence object in the database.

DROP SEQUENCE on page 705
Drops a sequence object from the database.

**DROP SEQUENCE**

Drops a sequence object from the database.

**Syntax:**

```
DROP SEQUENCE [ IF EXISTS ] sequence-name
```

**Related concepts**

- **CREATE SEQUENCE** on page 704
  Creates a new sequence object in the database.
- **ALTER SEQUENCE** on page 704
  Modifies the definition of an existing sequence in the database.

**CREATE TABLE**

Creates a new table object in the database.

**Syntax:**

```
CREATE [ TEMP ] TABLE [ IF NOT EXISTS ] table-specification
{
    column-name data-type
    [ DEFAULT default-value ] [ NOT NULL ]
    [ PRIMARY KEY ] [ contraint-name ]
    [ UNIQUE ] [ contraint-name ]
    [ CHECK ( sql-condition ) ] [ contraint-name ]
    REFERENCES table-name
    [ ( column-name |,...| ) ]
    [ ON DELETE CASCADE ]
    [ contraint-name ]

    [ PRIMARY KEY ( column-name |,...| ) ] [ contraint-name ]
    [ UNIQUE ( column-name |,...| ) ] [ contraint-name ]
    [ CHECK ( sql-condition ) ] [ contraint-name ]
    [ FOREIGN KEY ( column-name |,...| ) ]
    REFERENCES table-name
    [ ( column-name |,...| ) ]
    [ ON DELETE CASCADE ]
    [ contraint-name ]

    [ ]
    [ WITH NO LOG ]
    [ IN tablespace-name ]
    [ EXTENT SIZE integer ]
    [ NEXT SIZE integer ]
    [ LOCK MODE PAGE ] [ ROW ]
}
```

**Related concepts**

- **ALTER TABLE** on page 706
  Modifies the definition of an existing table in the database.
- **DROP TABLE** on page 707
Drops a table object from the database.

**ALTER TABLE**
Modifies the definition of an existing table in the database.

**Syntax:**

```
ALTER TABLE table-specification
(
  DROP ( column-name [, ...] )
  ADD ( column-name data-type
    DEFAULT default-value | NOT NULL |
    PRIMARY KEY | constraint-name |
    UNIQUE | constraint-name |
    CHECK ( sql-condition ) | constraint-name |
    REFERENCES table-name
      ( column-name [, ...] ) |
      ON DELETE CASCADE |
      constraint-name |
    | BEFORE column-name
  [, ...]
)
ADD ( column-name data-type
  DEFAULT default-value | NOT NULL |
  PRIMARY KEY | constraint-name |
  UNIQUE | constraint-name |
  CHECK ( sql-condition ) | constraint-name |
  REFERENCES table-name
    ( column-name [, ...] ) |
    ON DELETE CASCADE |
    constraint-name |
  [, ...]
)
DROP CONSTRAINT constraint-name
ADD CONSTRAINT
  PRIMARY KEY ( column-name [, ...] ) | constraint-name |
  UNIQUE ( column-name [, ...] ) | constraint-name |
  CHECK ( sql-condition ) | constraint-name |
  FOREIGN KEY ( column-name [, ...] )
  REFERENCES table-name
    ( column-name [, ...] ) |
    ON DELETE CASCADE |
    constraint-name |
  [, ...]
LOCK MODE ( PAGE | ROW )
MODIFY NEXT SIZE integer
)
```

**Related concepts**

- CREATE TABLE on page 705
  Creates a new table object in the database.
- DROP TABLE on page 707
Drops a table object from the database.

**DROP TABLE**
Drops a table object from the database.

**Syntax:**
```
DROP TABLE IF EXISTS table-specification
```

**Related concepts**
- **ALTER TABLE** on page 706
  Modifies the definition of an existing table in the database.
- **CREATE TABLE** on page 705
  Creates a new table object in the database.

**CREATE INDEX**
Creates a new index object in the database.

**Syntax:**
```
CREATE [ UNIQUE | CLUSTER | UNIQUE CLUSTER | INDEX ]
[ IF NOT EXISTS ] index-name
ON table-specification
( column-name [ ASCENDING | DESCENDING ] [, ... ] )
```

**Related concepts**
- **ALTER INDEX** on page 707
  Modifies the definition of an existing index in the database.
- **DROP INDEX** on page 707
  Drops an index object from the database.

**ALTER INDEX**
Modifies the definition of an existing index in the database.

**Syntax:**
```
ALTER INDEX index-name TO [ NOT ] CLUSTER
```

**Related concepts**
- **CREATE INDEX** on page 707
  Creates a new index object in the database.
- **DROP INDEX** on page 707
  Drops an index object from the database.

**DROP INDEX**
Drops an index object from the database.

**Syntax:**
```
DROP INDEX [ IF EXISTS ] index-name
```
**Related concepts**

**ALTER INDEX** on page 707
Modifies the definition of an existing index in the database.

**CREATE INDEX** on page 707
Creates a new index object in the database.

---

**CREATE VIEW**

Creates a new view object in the database.

**Syntax:**

```
CREATE VIEW IF NOT EXISTS view-name
  (column-alias-name [, ...])
  AS sub-query
  WITH CHECK OPTION
```

where `sub-query` is a limited syntax of the `SELECT` statement.

**Related concepts**

**DROP VIEW** on page 708
Drops a view object from the database.

---

**DROP VIEW**

Drops a view object from the database.

**Syntax:**

```
DROP VIEW IF EXISTS view-name
```

**Related concepts**

**CREATE VIEW** on page 708
Creates a new view object in the database.

---

**CREATE SYNONYM**

Creates a new synonym object in the database.

**Syntax:**

```
CREATE SYNONYM IF NOT EXISTS synonym-name
  FOR table-specification
```

**Related concepts**

**DROP SYNONYM** on page 708
Drops a synonym object from the database.

---

**DROP SYNONYM**

Drops a synonym object from the database.

**Syntax:**

```
DROP SYNONYM IF EXISTS synonym-name
```

**Related concepts**

**CREATE SYNONYM** on page 708
Creates a new synonym object in the database.

**RECOVER TABLE**

Restores a database table.

**Syntax:**

```
RECOVER TABLE table-name
```  

**RENAME**

Renames an object in the database.

**Syntax:**

```
RENAME { TABLE | COLUMN | INDEX | SEQUENCE } old-name TO new-name
```  

**Dynamic SQL management**

Explains how to execute and manage SQL statements at runtime.

**Understanding dynamic SQL**

This is an introduction to dynamic SQL programming.

Basic SQL instructions are part of the language syntax as Static SQL statements, but only a limited number of SQL instructions are supported this way.

*Dynamic SQL* management allows you to execute any kind of SQL statement, hard coded or created at runtime, with or without SQL parameters, returning or not returning a result set.

In order to execute an SQL statement dynamically, you must first **PREPARE** the SQL statement to initialize a *statement handle*, then **EXECUTE** the prepared statement one or more times:

![Dynamic SQL management diagram](image)

*Figure 26: Dynamic SQL management diagram*

When you no longer need the prepared statement, you can **FREE** the statement handle to release allocated resources:
When using insert cursors or SQL statements that produce a result set (like SELECT), you must declare a cursor with a prepared statement handle.

Prepared SQL statements can contain SQL parameters by using `?` placeholders in the SQL text. In this case, the EXECUTE or OPEN instruction supplies input values in the USING clause.

**Note:** To increase performance efficiency of SQL executed in a loop, use PREPARE outside the loop, together with EXECUTE inside the loop, to eliminate overhead caused by redundant parsing and optimizing.

The `EXECUTE IMMEDIATE` instruction prepares and executes an SQL statement in a single instruction. SQL parameters and result sets cannot be used with `EXECUTE IMMEDIATE`.

The `base.SQLHandle` built-in class is a 3GL API to execute SQL statements dynamically and perform SQL introspection (to get result set column types).

**Related concepts**

- Database transactions on page 688
- Database transaction concepts and handling.

**PREPARE (SQL statement)**

Prepares an SQL statement for execution.

**Syntax**

```
PREPARE sid FROM sqltext
```

1. `sid` is an identifier to handle the prepared SQL statement.
2. `sqltext` is a string expression containing the SQL statement to be prepared.

**Usage**

The `PREPARE` instruction allocates resources for an SQL statement handle, in the context of the current database connection. The SQL text is sent to the database server for parsing, validation and to generate the execution plan.

Prepared SQL statements can be executed with the `EXECUTE` instruction, or, when the SQL statement generates a result set, the prepared statement can be used to declare cursors with the `DECLARE` instruction.
A statement identifier (sid) can represent only one SQL statement at a time. You can execute a new PREPARE instruction with an existing statement identifier if you wish to assign the text of a different SQL statement to the statement identifier. The scope of reference of the sid statement identifier is local to the module where it is declared. That is, the identifier of a statement that was prepared in one module cannot be referenced from another module.

The SQL statement can have parameter placeholders, identified by the question mark (?) character. You cannot directly reference a variable in the text of a prepared SQL statement. You cannot use question mark (?) placeholders for SQL identifiers such as a table name or a column name; you must specify these identifiers in the statement text when you prepare it.

Resources allocated by PREPARE can be released later by the FREE instruction.

The number of prepared statements in a single program is limited by the database server and the available memory. Make sure that you free the resources when you no longer need the prepared statement.

Some database servers support multiple SQL statement preparation in a unique PREPARE instruction, but most database servers deny multiple statements. It is recommended that you only prepare one SQL statement at a time.

Example

FUNCTION deleteOrder(n)
  DEFINE n INTEGER
  PREPARE s1 FROM "DELETE FROM order WHERE key=?"
  EXECUTE s1 USING n
  FREE s1
END FUNCTION

See EXECUTE for more code examples.

Related concepts

SQL execution diagnostics on page 591

If an SQL statement execution fails, error description can be found in the SQLCA.SQLCODE, SQLSTATE, STATUS and SQLERRMESSAGE predefined registers.

EXECUTE (SQL statement)

This instruction runs an SQL statement previously prepared.

Syntax

EXECUTE sid
  [USING pvar [IN|OUT|INOUT] [, ...]]
  [INTO fvar [, ...]]

1. sid is an identifier to handle the prepared SQL statement.
2. pvar is a variable containing an input value for an SQL parameter.
3. fvar is a variable used as fetch buffer.

Usage

The EXECUTE instruction performs the execution of an SQL statement initiated by a PREPARE instruction.

Once prepared, an SQL statement can be executed several times with different SQL parameters.

If the SQL statement has (?) parameter placeholders, you must specify the USING clause to provide a list of variables as parameter buffers. Parameter values are assigned by position.

If the SQL statement returns a result set with one row, you can specify the INTO clause to provide a list of variables to receive the result set column values. Fetched values are assigned by position. If the SQL statement returns a result set with more than one row, the instruction raises an exception.

The IN, OUT or INOUT options can only be used for simple variables, you cannot specify those options for a complete record with the record.* notation.
The IN, OUT or INOUT options can be used to call stored procedures having input / output parameters. Use the IN, OUT or INOUT options to indicate if a parameter is respectively for input, output or both.

You cannot execute a prepared SQL statement based on database tables if the table structure has changed (ALTER TABLE) since the PREPARE instruction; you must re-prepare the SQL statement.

**Example**

```sql
MAIN
  DEFINE var1 CHAR(20)
  DEFINE var2 INTEGER
  DATABASE stores
  PREPARE s1 FROM "UPDATE tab SET col=? WHERE key=?"
  LET var1 = "aaaa"
  LET var2 = 345
  EXECUTE s1 USING var1, var2
  
  PREPARE s2 FROM "SELECT col FROM tab WHERE key=?"
  LET var2 = 564
  EXECUTE s2 USING var2 INTO var1
  
  PREPARE s3 FROM "CALL myproc(?,?)"
  LET var1 = 'abc'
  EXECUTE s3 USING var1 IN, var2 OUT
END MAIN
```

**Related concepts**

- SQL execution diagnostics on page 591
  - If an SQL statement execution fails, error description can be found in the SQLCA.SQLCODE, SQLSTATE, STATUS and SQLERRMESSAGE predefined registers.

- Stored procedures on page 649
  - Executing stored procedures with different database engine types.

**FREE (SQL statement)**

Releases the resources allocated to a prepared statement.

**Syntax**

```sql
FREE sid
```

1. *sid* is the identifier of the prepared SQL statement.

**Usage**

The FREE instruction takes the name of a statement as parameter and releases the resources allocated by the PREPARE instruction.

After using FREE, the statement identifier cannot be referenced by a cursor declaration (DECLARE), or by the EXECUTE statement, until you prepare the statement again.

Free the statement if it is not needed anymore, this saves resources on the database client and database server side.

**Example**

```sql
FUNCTION update_customer_name( key, name )
  DEFINE key INTEGER
  DEFINE name CHAR(10)
  PREPARE s1 FROM "UPDATE customer SET name=? WHERE customer_num=?"
```
EXECUTE IMMEDIATE
Performs a simple SQL execution without SQL parameters or result set.

Syntax
EXECUTE IMMEDIATE sqltext

1. sqltext is a string expression containing the SQL statement to be executed.

Usage
The EXECUTE IMMEDIATE instruction passes an SQL statement to the database server for execution in the current database connection.

The SQL statement used by EXECUTE IMMEDIATE must be a single statement without SQL parameters and must not produce a result set.

This instruction is equivalent to PREPARE, EXECUTE and FREE done in one step.

Note: Unlike the EXECUTE instruction, EXECUTE IMMEDIATE has no USING clause (to specify SQL parameters), nor does it support an INTO clause (to fetch values into variables). When specifying a USING keyword after the string containing the SQL statement, the compiler will consider this as the USING formatting operator.

Example
MAIN
  DATABASE stores
  EXECUTE IMMEDIATE "UPDATE tab SET col='aaa' WHERE key=345"
END MAIN

Related concepts
SQL execution diagnostics on page 591
If an SQL statement execution fails, error description can be found in the SQLCA.SQLCODE, SQLSTATE, STATUS and SQLERRMESSAGE predefined registers.

The base.SQLHandle built-in class
Handle SQL queries with a 3GL API.

Genero BDL provides a 3GL API to execute SQL queries and introspect result set column information with the base.SQLHandle built-in class.

The class implements typical SQL statement execution methods existing in well-known APIs, such as:

• prepare("sql-text")
• setParameter()
• execute()
• open(), openScrollCursor()
• fetch(), fetchFirst(), fetchLast(), ...

The class also implements introspection methods for the result set columns:

• getResultCount()
• getResultType(index)
• getResultName(index)
• getResultValue(index)
This class is provided to allow generic code implementation for specific needs. Consider using traditional static and dynamic SQL instruction for regular code implementing your business rules; the 3GL code based on the SqlHandle class is not as readable as static or dynamic SQL.

**fglhint_* SQL comments**

Using special SQL comment hints to control statement execution.

**Syntax**

```
/*
  { fglhint_insert
  | fglhint_update
  | fglhint_delete
  | fglhint_select
  | fglhint_other
  }
  [ fglhint_no_ifxemul ]
*/
```

The `fglhint_*` indicators help the database driver to identify the SQL statement and control Informix® emulation:

1. `fglhint_insert`: Regular INSERT (preparable, without result set).
2. `fglhint_update`: Regular UPDATE (preparable, without result set).
3. `fglhint_delete`: Regular DELETE (preparable, without result set).
4. `fglhint_select`: Regular SELECT (preparable, with result set).
5. `fglhint_other`: Any other non-preparable SQL statement (maybe with result set).
6. `fglhint_no_ifxemul`: Disable Informix® emulation (like the FGLPROFILE entry, but for this statement only).

**Usage**

When preparing an SQL statement, the ODI database drivers must identify the type of SQL statement, in order to set database client API options to perform the statement properly.

For example, when executing an `SELECT` statement that returns a result set, ODBC options for a server-side cursor must be set with Microsoft® SQL Server clients. If a server-side cursor is not used only one active cursor is used in a Genero program.

Or, for example, when executing an `SELECT` statement that creates temporary tables with SQL Server, it must be done with a direct ODBC execution (using ODBC SQLExecDirect instead of SQLPrepare/SQLExecute), otherwise the temporary table is created in the scope of the sp_prepare() stored procedure, and is dropped immediately after returning from the sp_prepare() call.

Specified inside /* */ C-style comments, the `fglhint_*` keywords can be used as SQL statement recognition directives:

- When the SQL statement looks like a regular INSERT statement, but uses an `OUTPUT INSERTED.*` clause, add the `/* fglhint_select */` SQL comment, to indicate that it must be treated as a regular, preparable SELECT statement returning a result set.
- When the SQL statement looks like a regular SELECT statement, but has an `INTO newtable` clause to create a new table, add the `/* fglhint_insert */` SQL comment, to indicate that it must be treated as a regular, preparable INSERT statement.
- When the SQL statement looks like a regular SELECT statement, but has an `INTO #newtemptable` clause to create a new temporary table, add the `/* fglhint_other */` SQL comment, to indicate that it must be treated as a non-preparable INSERT statement (to be executed directly).

**Note:** /* */ C-style comments are only allowed in dynamic SQL statements, the BDL language does not allow such comments in static SQL statements, which is based on the Informix-SQL syntax.
For example, this statement performs an INSERT with an OUTPUT clause, that will produce a result set. Use the fglhint_select hint, to indicate that the statement must be executed as a regular SELECT:

```
DECLARE c1 CURSOR
    FROM "/* fglhint_select */ INSERT INTO table1 OUTPUT INSERTED.* SELECT *
    FROM customers"
```

This statement performs a SELECT INTO newtable, that will be executed as an INSERT (without a result set), because we use the fglhint_insert hint:

```
EXECUTE IMMEDIATE "/* fglhint_insert */ SELECT * INTO newtable FROM
    customers WHERE cust_valid='Y'"
```

In order to perform a SELECT INTO #temptable, the statement must be executed directly without using the sp_prepare/sp_execute procedures of SQL Server (otherwise the temp table will only exist in the context of the stored procedures). Therefore, we use the fglhint_other hint to force a direct SQL execution:

```
EXECUTE IMMEDIATE "/* fglhint_other */ SELECT * INTO #tmptab1 SELECT * FROM
    customers"
DECLARE c2 CURSOR FROM "SELECT * FROM #tmptab1 ..."
```

Furthermore, Informix® SQL emulation can be disabled by using the fglhint_no_ifxemul hint:

```
PREPARE s1 FROM "/* fglhint_no_ifxemul */ SELECT * FROM [dbo].[mytable]"
```

Note: When using the fglhint_no_ifxemul hint, Informix® emulation will only be disabled for this SQL statement. Other SQL statements executed by your program can use the Informix® emulations.

SQL statement type identification and Informix® emulation hints can be combined:

```
PREPARE s1 FROM "/* fglhint_no_ifxemul fglhint_select */ INSERT INTO [dbo].
    [mytable] OUTPUT ..."
```

Related concepts

IBM Informix emulation parameters in FGLPROFILE on page 668
Emulation of Informix® specific SQL features can be controlled with FGLPROFILE entries.

Static SQL statements on page 694
Describes static SQL statements supported in the language.

---

**Result set processing**

Shows how to fetch rows from a database query.

**Understanding database result sets**

This is an introduction to database result sets.

A database result set is a group of rows produced by an SQL statement such as SELECT. The result set is maintained by the database server. In a program, you handle a result set with a database cursor.

First you must declare the database cursor with the DECLARE instruction. This instruction sends the SQL statement to the database server for parsing, validation and to generate the execution plan.
**Figure 28: Database result set**

The result set is produced after execution of the SQL statement, when the database cursor is associated with the result set by the `OPEN` instruction. At this point, no data rows are transmitted to the program. You must use the `FETCH` instruction to retrieve data rows from the database server.

**Figure 29: FETCH instruction**

When finished with the result set processing, you must `CLOSE` the cursor to release the resources allocated for the result set on the database server. The cursor can be reopened if needed. If the SQL statement is no longer needed, you can free the resources allocated to statement execution with the `FREE` instruction.
The scope of reference of a database cursor is local to a module, so a cursor that was declared in one source file cannot be referenced in a statement in another file.

The language supports sequential cursors and scrollable cursors. Sequential cursors, which are unidirectional, are used to retrieve rows for a REPORT, for example. Scrollable cursors allow you to move backwards or to an absolute or relative position in the result set. Specify whether a cursor is scrollable with the SCROLL option of the DECLARE instruction.

For better code readability, use a FOREACH / END FOREACH loop, to perform the equivalent of an OPEN + FETCH (in WHILE loop) + CLOSE.

**Related concepts**
Reports on page 2026

### DECLARE (result set cursor)
Associates a database cursor with an SQL statement producing a result set.

**Syntax 1: Cursor declared with a static SQL statement.**

```sql
DECLARE cid [SCROLL] CURSOR [WITH HOLD] FOR select-statement
```

1. `cid` is the identifier of the database cursor.
2. `select-statement` is a SELECT statement defined in static SQL.

**Syntax 2: Cursor declared with a prepared statement.**

```sql
DECLARE cid [SCROLL] CURSOR [WITH HOLD] FOR sid
```

1. `cid` is the identifier of the database cursor.
2. `sid` is the identifier of a prepared SQL statement.

**Syntax 3: Cursor declared with a string expression.**

```sql
DECLARE cid [SCROLL] CURSOR [WITH HOLD] FROM expr
```

1. `cid` is the identifier of the database cursor.
2. *expr* is any expression that evaluates to a string.

**Syntax 4: Cursor declared with an SQL Block.**

```
DECLARE cid [SCROLL] CURSOR [WITH HOLD] FOR SQL sql-statement END SQL
```

1. *cid* is the identifier of the database cursor.
2. *sql-statement* is a statement defined in an SQL block.

**Usage**

The `DECLARE` instruction allocates resources for an SQL statement handle, in the context of the current connection. The SQL text is sent to the database server for parsing, validation and to generate the execution plan.

After declaring the cursor, you can use the `OPEN` instruction to execute the SQL statement and produce the result set. Rows can be fetched with the `FETCH` instruction or in a `FOREACH` loop.

`DECLARE` must precede any other statement that refers to the cursor during program execution.

The scope of reference of the *cid* cursor identifier is local to the module where it is declared.

Resources allocated by the `DECLARE` *cursor-name* can be released later by the `FREE` *cursor-name* instruction.

The static `select-statement` used in the `DECLARE` can contain program variables, or ? (question mark) parameter placeholders, that can be bound to program variables with the `USING` clause of the `OPEN` instruction.

**Important**: When using program variables in a static SQL statement of a `DECLARE CURSOR` instruction, make sure that these variables are available when executing the `OPEN` or `FOREACH` instruction. Otherwise, the behavior is unexpected and can make the runtime system crash. Do not mix ? SQL parameter placeholders with program variables: This would lead to a runtime error at `OPEN` or `FOREACH` time, because the number of SQL parameters will not match the number of variable provided in the `USING` clause. The `fglcomp -S` option extracts the static SQL statements showing ? placeholders instead of programs variables. This can help to check the actual number of SQL parameters in a static SQL statement.

The maximum number of declared cursors in a single program is limited by the database server and the available memory. Make sure that you free the resources when you no longer need the declared cursor.

When declaring a cursor with a static `select-statement`, the statement can include an `INTO` clause. However, to be consistent with prepared statements you better omit the `INTO` clause in the SQL text and use the `INTO` clause of the `FETCH` statement to retrieve the values from the result set.

Add the `FOR UPDATE` clause to the `SELECT` statement to declare an update cursor. You can then use the update cursor to modify (update or delete) the current row with `WHERE CURRENT OF`.

Use the `WITH HOLD` option with care, because this feature is specific to IBM® Informix® servers. Other database servers do not behave as Informix® does with this type of cursor. For example, if the `SELECT` is not declared `FOR UPDATE`, most database servers keep cursors open after the end of a transaction, but IBM® DB2® automatically closes all cursors when the transaction is rolled back.

**Forward only cursors**

If you use only the `DECLARE CURSOR` keywords, you create a *sequential cursor*, which can fetch only the next row in sequence from the result set. The sequential cursor can read through the result set only once each time it is opened.

If you are using a sequential cursor for a select cursor, on each execution of the `FETCH` statement, the database server returns the contents of the current row and locates the next row in the result set.

Cursors can be declare with a static `SELECT` statement:

```
MAIN
  DATABASE stores
  DECLARE c1 CURSOR FOR SELECT * FROM customer
```
Cursors can also be declared with a SELECT statement defined in a character string:

```sql
MAIN
  DEFINE key INTEGER
  DEFINE cust RECORD
      num INTEGER,
      name CHAR(50)
  END RECORD
DATABASE stores
DECLARE c1 CURSOR FOR
    SELECT customer_num, cust_name FROM customer WHERE customer_num > ?
  LET key=101
  FOREACH c1 USING key INTO cust.*
    DISPLAY cust.*
  END FOREACH
END MAIN
```

**Scrollable cursors**

Use the `DECLARE SCROLL CURSOR` keywords to create a scrollable cursor, which can fetch rows of the result set in any sequence. Until the cursor is closed, the database server retains the result set of the cursor in a static data set (for example, in a temporary table like Informix®). You can fetch the first, last, or any intermediate rows of the result set as well as fetch rows repeatedly without having to close and reopen the cursor. On a multiuser system, the rows in the tables from which the result set rows were derived might change after the cursor is opened and a copy of the row is made in the static data set. If you use a scroll cursor within a transaction, you can prevent copied rows from changing, either by setting the isolation level to `REPEATABLE READ` or by locking the entire table in share mode during the transaction. Scrollable cursors cannot be declared `FOR UPDATE`.

With most database servers, scrollable cursors take quite a few resources to hold a static copy of the result set. Therefore you should consider optimizing scrollable cursor usage by fetching only the primary keys of rows, and execute a secondary `SELECT` statement to fetch other fields for each row that must be displayed.

The `DECLARE [SCROLL] CURSOR` syntax allows you to declare a cursor directly with a string expression, so that you do not have to use the `PREPARE` instruction. This simplifies the source code and speeds up the execution time for non-Informix® databases, because the SQL statement is not parsed twice.

```sql
MAIN
  DEFINE key INTEGER
  DEFINE cust RECORD
      num INTEGER,
      name CHAR(50)
  END RECORD
DATABASE stores
DECLARE c1 SCROLL CURSOR FOR
    SELECT customer_num, cust_name FROM customer WHERE customer_num > ?
  LET key=101
  FOREACH c1 USING key INTO cust.*
    DISPLAY cust.*
  END FOREACH
END MAIN
```

**Hold cursors**

Use the `WITH HOLD` option to create a hold cursor. A hold cursor allows uninterrupted access to a set of rows across multiple transactions. Ordinarily, all cursors close at the end of a transaction. A hold cursor does not close; it remains open after a transaction ends. A hold cursor can be either a sequential cursor or a scrollable cursor.
Note: WITH HOLD cursors are fully supported by Informix® database engines, but these must be used with care to write portable SQL code: Most database engines keep cursors open across transactions (with hold cursor is the default), and the ODI drivers emulate the Informix behavior by closing all cursors not declared WITH HOLD at the end of a transaction. But each database brand can have a specific behavior and implicitly close the SQL cursor, when rolling the transaction back, or when combining hold cursor option with SELECT ... FOR UPDATE. See the SQL programming topic about WITH HOLD cursors.

```sql
MAIN
  DEFINE key INTEGER
  DEFINE cust RECORD
    num INTEGER,
    name CHAR(50)
  END RECORD
  DATABASE stores
  DECLARE c1 CURSOR WITH HOLD FOR
    SELECT customer_num, cust_name FROM customer WHERE customer_num > ?
  LET key=101
  FOREACH c1 USING key INTO cust.*
    BEGIN WORK
      UPDATE cust2 SET name=cust.cust_name WHERE num=cust.num
      COMMIT WORK
    END FOREACH
END MAIN
```

Related concepts
CLOSE (result set cursor) on page 723
Closes a database cursor and frees resources allocated on the database server for the result set.

FREE (result set cursor) on page 723
Releases SQL cursor resources allocated by the DECLARE instruction.

OPEN (result set cursor)
Executes the SQL statement with result set associated with the specified database cursor

Syntax

```
OPEN cid
  ↓ USING pvar IN\OUT\INOUT {,....} ↓
  ↓ WITH REOPTIMIZATION ↓
```

1. `cid` is the identifier of the database cursor.
2. `pvar` is a variable containing an input value for an SQL parameter.

Usage:
The OPEN instruction executes the SQL statement of a cursor created by the DECLARE instruction. The result set is produced on the server side and rows can be fetched.

The USING clause is required to provide the SQL parameters as program variables, if the cursor was declared with a prepared statement that includes (?) question mark placeholders.

A subsequent OPEN statement closes the cursor and then reopens it. When the database server reopens the cursor, it creates a new result set, based on the current values of the variables in the USING clause. If the variables have changed since the previous OPEN statement, reopening the cursor can generate an entirely different result set.

The IN, OUT or INOUT options can be used to call stored procedures having input / output parameters and generating a result set. Use the IN, OUT or INOUT options to indicate if a parameter is respectively for input, output or both.

Sometimes, query execution plans need to be re-optimized when SQL parameter values change. Use the WITH REOPTIMIZATION clause to indicate that the query execution plan has to be re-optimized on the database server.
(this operation is normally done during the DECLARE instruction). If this option is not supported by the database server, it is ignored.

In an IBM® Informix® database that is ANSI-compliant, you receive an error code if you try to open a cursor that is already open. **Informix® only!**

A cursor is closed with the CLOSE instruction, or when the parent connection is terminated (typically, when the program ends). By using the CLOSE instruction explicitly, you release resources allocated for the result set in the db client library and on the database server.

The database server evaluates the values that are named in the USING clause of the OPEN statement only when it opens the cursor. While the cursor is open, subsequent changes to program variables in the OPEN clause do not change the result set of the cursor; you must re-open the cursor to re-execute the statement.

If you release cursor resources with a FREE instruction, you cannot use the cursor unless you declare the cursor again.

The IN, OUT or INOUT options can only be used for simple variables, you cannot specify those options for a complete record with the record.* notation.

**Example**

```main
MAIN
  DEFINE k INTEGER
  DEFINE n VARCHAR(50)
  DATABASE stores
  DECLARE c1 CURSOR FROM "SELECT cust_name FROM customer WHERE cust_id > ?"
  LET k = 102
  OPEN c1 USING k
  FETCH c1 INTO n
  LET k = 103
  OPEN c1 USING k
  FETCH c1 INTO n
END MAIN
```

**FETCH (result set cursor)**

Moves a cursor to a new row in the corresponding result set and retrieves the row values into fetch buffers.

**Syntax**

```sql
FETCH [direction] cid
   INTO fvar [, ...]
```

where **direction** is one of:

```sql
  NEXT
  | { PREVIOUS | PRIOR }
  | CURRENT
  | FIRST
  | LAST
  | ABSOLUTE position
  | RELATIVE offset
```

1. **cid** is the identifier of the database cursor.
2. **fvar** is a variable used as fetch buffer.
3. **direction** options different from NEXT can only be used with scrollable cursors.
4. **position** is an positive integer expression.
5. **offset** is a positive or negative integer expression.
Usage

The **FETCH** instruction retrieves a row from a result set of an opened cursor.

Before fetching rows, the cursor SQL statement must be executed with the **OPEN** instruction.

Fetching rows can have specific behavior when the cursor was declared **FOR UPDATE** to perform a positioned update or delete.

The **INTO** clause can be used to provide the fetch buffers that receive the result set column values.

A sequential cursor can fetch only the next row in sequence from the result set.

The **NEXT** clause (the default) retrieves the next row in the result set. If the row pointer was on the last row before executing the instruction, the SQL code is set to 100 (**NOTFOUND**), and the row pointer remains on the last row. (if you issue a **FETCH PREVIOUS** at this time, you get the next-to-last row).

The **PREVIOUS** clause retrieves the previous row in the result set. If the row pointer was on the first row before executing the instruction, the SQL code is set to **NOTFOUND/100**), and the row pointer remains on the first row. (if you issue a **FETCH NEXT** at this time, you get the second row).

The **CURRENT** clause retrieves the current row in the result set.

The **FIRST** clause retrieves the first row in the result set.

The **LAST** clause retrieves the last row in the result set.

The **ABSOLUTE** clause retrieves the row at position in the result set. If the position is not correct, the SQL code is set to 100 (**NOTFOUND**). Absolute row positions are numbered from 1.

The **RELATIVE** clause moves offset rows in the result set and returns the row at the current position. The offset can be a negative value. If the offset is not correct, the SQL code is set to 100 (**NOTFOUND**). If offset is zero, the current row is fetched.

Example

```main
DEFINE cust_rec RECORD
    cnum INTEGER,
    cname CHAR(20)
END RECORD
DATABASE stores
DECLARE c1 SCROLL CURSOR FOR SELECT customer_num, cust_name FROM customer
OPEN c1
FETCH c1 INTO cust_rec.*
FETCH LAST c1 INTO cust_rec.*
FETCH PREVIOUS c1 INTO cust_rec.*
FETCH FIRST c1 INTO cust_rec.*
FETCH LAST c1 -- INTO clause is optional
FETCH FIRST c1 -- INTO clause is optional
END MAIN
```

Related concepts

**FOREACH (result set cursor)** on page 724
Processes a series of data rows returned from a database cursor.

**SQL execution diagnostics** on page 591
If an SQL statement execution fails, error description can be found in the SQLCA.SQLCODE, SQLSTATE, STATUS and SQLERRMESSAGE predefined registers.

**CLOSE (result set cursor)**
Closes a database cursor and frees resources allocated on the database server for the result set.

**Syntax**

```
CLOSE cid
```

1. *cid* is the identifier of the database cursor.

**Usage**
The CLOSE instruction releases the resources allocated for the result set on the database server.

After using the CLOSE instruction, you must reopen the cursor with OPEN before retrieving values with FETCH.

It is recommended that you close the cursor when the result set is no longer used, this saves resources on the database client and database server side.

**Example**

```
MAIN
   DATABASE stores
   DECLARE c1 CURSOR FOR SELECT * FROM customer
   OPEN c1
   CLOSE c1
   OPEN c1
   CLOSE c1
END MAIN
```

**Related concepts**
FREE (result set cursor) on page 723
Releases SQL cursor resources allocated by the DECLARE instruction.

**FREE (result set cursor)**
Releases SQL cursor resources allocated by the DECLARE instruction.

**Syntax**

```
FREE cid
```

1. *cid* is the identifier of the database cursor.

**Usage**
The FREE instruction releases the resources allocated for a cursor created by a DECLARE instruction.

If not done, the cursor is automatically closed when doing a FREE.

When cursor resources are released with FREE, the cursor must be declared again before usage.

Free the cursor when the result set is no longer used by the program; this saves resources on the database client and database server side.

**Example**

```
MAIN
   DEFINE i, j INTEGER
```
```
DATABASE stores
FOR i=1 TO 10
    DECLARE c1 CURSOR FOR SELECT * FROM customer
    FOR j=1 TO 10
        OPEN c1
        FETCH c1
        CLOSE c1
    END FOR
    FREE c1
END FOR
END MAIN
```

**Related concepts**

CLOSE (result set cursor) on page 723
Closes a database cursor and frees resources allocated on the database server for the result set.

**FOREACH (result set cursor)**
Processes a series of data rows returned from a database cursor.

**Syntax**

```
FOREACH cid
    [ USING pvar [IN|OUT|INOUT] [,....] ]
    [ WITH REOPTIMIZATION ]
    [ INTO fvar [,....] ]
    [ statement ]
    [ CONTINUE FOREACH ]
    [ EXIT FOREACH ]
[....]
END FOREACH
```

1. `cid` is the identifier of the database cursor.
2. `pvar` is a variable containing an input value for an SQL parameter.
3. `fvar` is a variable used as fetch buffer.

**Usage**

Use the FOREACH instruction to retrieve and process database rows that were selected by a query. This instruction is equivalent to using the OPEN, FETCH and CLOSE cursor instructions:

1. Open the specified cursor
2. Fetch the rows selected
3. Close the cursor (after the last row has been fetched)

You must declare the cursor (by using the DECLARE instruction) before the FOREACH instruction can retrieve the rows. A compile-time error occurs unless the cursor was declared prior to this point in the source module. You can reference a sequential cursor, a scroll cursor, a hold cursor, or an update cursor, but FOREACH only processes rows in sequential order.

**Important:** When fetching rows into a dynamic array element with FOREACH `cursor INTO dyn-arr[index] . *`, the array element will be implicitly allocated in order to fetch the row into the array variable. This occurs also when reaching the end of the result set. Therefore, you always need to delete the last element of the dynamic array filled by a FOREACH loop.

The FOREACH statement performs successive fetches until all rows specified by the SELECT statement are retrieved. Then the cursor is automatically closed. It is also closed if a WHENEVER NOT FOUND exception handler within the FOREACH loop detects a NOTFOUND condition.
After a FOREACH loop, STATUS and SQLCA.SQLCODE will not be set to NOTFOUND/100 if no rows are returned by the query: If no error occurred, these registers will hold the value zero.

The USING clause is required to provide the SQL parameter buffers, if the cursor was declared with a prepared statement that includes (?) question mark placeholders.

The IN, OUT or INOUT options can be used to call stored procedures having input / output parameters and generating a result set. Use the IN, OUT, or INOUT options to indicate if a parameter is respectively for input, output, or both.

The INTO clause can be used to provide the fetch buffers that receive the row values.

Use the WITH REOPTIMIZATION clause to indicate that the query execution plan has to be re-optimized.

The CONTINUE FOREACH instruction interrupts processing of the current row and starts processing the next row. The runtime system fetches the next row and resumes processing at the first statement in the block.

The EXIT FOREACH instruction interrupts processing and ignores the remaining rows of the result set.

The IN, OUT, or INOUT options can only be used for simple variables; you cannot specify those options for a complete record with the record.* notation.

**Example**

Fetching rows into a dynamic array:

```plaintext
MAIN
  DEFINE clist DYNAMIC ARRAY OF RECORD
    cnum INTEGER,
    cname CHAR(50)
  END RECORD
DEFINE i INTEGER
DATABASE stores
DECLARE c1 CURSOR FOR SELECT customer_num, cust_name FROM customer
LET i=1
FOREACH c1 INTO clist[i].*
  DISPLAY clist[i].*
  LET i=i+1
END FOREACH
CALL clist.deleteElement(i)
DISPLAY "Number of rows found: ", clist.getLength()
END MAIN
```

Fetching rows into a static array:

```plaintext
MAIN
  DEFINE clist ARRAY[200] OF RECORD
    cnum INTEGER,
    cname CHAR(50)
  END RECORD
DEFINE i INTEGER
DATABASE stores
DECLARE c1 CURSOR FOR SELECT customer_num, cust_name FROM customer
LET i=0 -- Assign zero!
FOREACH c1 INTO clist[i+1].*
  LET i=i+1
  DISPLAY clist[i].*
END FOREACH
DISPLAY "Number of rows found: ", i
END MAIN
```

**Related concepts**

SQL execution diagnostics on page 591
If an SQL statement execution fails, error description can be found in the SQLCA.SQLCODE, SQLSTATE, STATUS and SQLERRMESSSAGE predefined registers.

## Positioned updates/deletes

Describes row modification based on a FOR UPDATE cursor.

### Understanding positioned update or delete

This is an introduction to SQL positioned UPDATE/DELETE.

When declaring a database cursor with a SELECT statement using a unique table and ending with the FOR UPDATE keywords, you can modify the current row pointed by the FOR UPDATE cursor with UPDATE ... WHERE CURRENT OF, or the current row with DELETE ... WHERE CURRENT OF statements. Such an operation is called positioned update or positioned delete.

Do not confuse positioned update with the use of SELECT FOR UPDATE statements that are not associated with a database cursor. Executing SELECT FOR UPDATE statements is supported by the language, but you cannot perform positioned updates since there is no cursor identifier associated with the result set.

**Note:** Some database servers do not support hold cursors (WITH HOLD) declared with a SELECT statement including the FOR UPDATE keywords. The SQL standards require for update cursors to be automatically closed at the end of a transaction. Therefore, it is strongly recommended that you use positioned updates in a transaction block.

To perform a positioned update or delete, perform a DECLARE instruction with a SELECT FOR UPDATE statement.

---

**Figure 31: SELECT FOR UPDATE statement**

Then, start a transaction, OPEN the cursor and FETCH a row.
Then, **UPDATE** or **DELETE** the current row with the **WHERE CURRENT OF** clause, before ending the transaction.

**Related concepts**
- **Result set processing** on page 715
  Shows how to fetch rows from a database query.
- **Database transactions** on page 688
  Database transaction concepts and handling.
- **Example 1: Positioned UPDATE statement** on page 729
DECLARE (SELECT ... FOR UPDATE)

Associate a database cursor with a SELECT statement to perform positioned updates and deletes

Syntax

```
DECLARE cid [SCROLL] CURSOR [WITH HOLD] 
    FOR select-statement | sid |
```

1. `cid` is the identifier of the database cursor.
2. `select-statement` is a SELECT statement defined in static SQL, with the FOR UPDATE keywords.
3. `sid` is the identifier of a prepared SELECT statement including the FOR UPDATE keywords.

Usage

DECLARE ... FOR UPDATE will define a cursor that can be used to do positioned updates and deletes with the WHERE CURRENT OF clause, and/or to set an exclusive lock on fetched rows in a transaction.

DECLARE must precede any other statement that refers to the cursor during program execution.

To perform positioned updates, the select-statement must include the FOR UPDATE keywords.

The scope of reference of the `cid` cursor identifier is local to the module where it is declared. Therefore, you must execute the DECLARE, UPDATE or DELETE instructions in the same module.

The static select-statement used in the DECLARE can contain ? (question mark) parameter placeholders, that can be bound to program variables with the USING clause of the OPEN instruction.

**Important:** Use the WITH HOLD option carefully, because this feature is specific to IBM® Informix® servers. Other database servers do not behave as Informix® does with such cursors. For example, if the SELECT is not declared FOR UPDATE, most database servers keep cursors open after the end of a transaction, but IBM® DB2® automatically closes all cursors when the transaction is rolled back.

Related concepts

Result set processing on page 715
Shows how to fetch rows from a database query.

Database transactions on page 688
Database transaction concepts and handling.

UPDATE ... WHERE CURRENT OF

Updates the current row in a result set of a database cursor declared for update.

Syntax

```
UPDATE table-specification
SET
    column = [ variable | sql-expression | ]
WHERE CURRENT OF cid
```

1. `table-specification` identifies the target table (see UPDATE for more details).
2. `column` is a name of a table column.
3. `variable` is a program variable, a record member or an array member used as a parameter buffer to provide values.
4. `sql-expression` is an expression supported by the database server, this can be a literal or NULL for example.
5. `cid` is the identifier of the database cursor declared for update.
Usage

Use `UPDATE ... WHERE CURRENT OF` to modify the values of the row currently pointed by the associated `FOR UPDATE` cursor.

The scope of reference of the `cid` cursor identifier is local to the module where it is declared. Therefore, you must execute the `DECLARE, UPDATE` or `DELETE` instructions in the same module.

There must be a current row in the result set. Make sure that the SQL status returned by the last `FETCH` is equal to zero.

If the `DECLARE` statement that created the cursor specified one or more columns in the `FOR UPDATE` clause, you are restricted to updating only those columns in a subsequent `UPDATE ... WHERE CURRENT OF` statement.

Related concepts

- Database transactions on page 688
- Database transaction concepts and handling.
- `DELETE ... WHERE CURRENT OF` on page 729
- Deletes the current row in a result set of a database cursor declared for update.

Example 1: Positioned UPDATE statement on page 729

DELETE ... WHERE CURRENT OF

Deletes the current row in a result set of a database cursor declared for update.

Syntax

```
DELETE FROM table-specification
    WHERE CURRENT OF cid
```

1. `table-specification` identifies the target table
2. `cid` is the identifier of the database cursor declared for update.

Usage

Use `DELETE ... WHERE CURRENT OF` to remove the row currently pointed by the associated `FOR UPDATE` cursor.

The scope of reference of the `cid` cursor identifier is local to the module where it is declared. Therefore, you must execute the `DECLARE, UPDATE` or `DELETE` instructions in the same module.

There must be a current row in the result set. Make sure that the SQL status returned by the last `FETCH` is equal to zero.

After the deletion, no current row exists; you cannot use the cursor to delete or update a row until you reposition the cursor with a `FETCH` statement.

Related concepts

- Database transactions on page 688
- Database transaction concepts and handling.
- `UPDATE ... WHERE CURRENT OF` on page 728
- Updates the current row in a result set of a database cursor declared for update.

Examples

WHERE CURRENT OF usage examples.

Example 1: Positioned UPDATE statement

```
MAIN
    DEFINE pname CHAR(30)
```
SQL insert cursors

Explains how to insert a log of rows into a table efficiently.

Understanding SQL insert cursors

This is an introduction to SQL insert cursors.

An insert cursor is a database cursor declared with a restricted form of the INSERT statement, designed to perform buffered row insertion in database tables.

The insert cursor simply inserts rows of data; it cannot be used to fetch data. When an insert cursor is opened, a buffer is created in memory to hold a block of rows. The buffer receives rows of data as the program executes PUT statements. The rows are written to disk only when the buffer is full. You can use the CLOSE, FLUSH, or COMMIT WORK statement to flush the buffer when it is less than full. You must close an insert cursor to insert any buffered rows into the database before the program ends. You can lose data if you do not close the cursor properly.

When the database server supports buffered inserts, an insert cursor increases processing efficiency (compared with embedding the INSERT statement directly). This process reduces communication between the program and the database server and also increases the speed of the insertions.

Before using the insert cursor, you must declare it with the DECLARE instruction using an INSERT statement.

```
DATABASE stock
DECLARE uc CURSOR FOR
  SELECT name FROM item WHERE key=123 FOR UPDATE
BEGIN WORK
  OPEN uc
  FETCH uc INTO pname
  IF sqlca.sqlcode=0 THEN
    LET pname = "Dummy"
    UPDATE item SET name=pname WHERE CURRENT OF uc
  END IF
  CLOSE uc
  COMMIT WORK
END MAIN
```
Once declared, you can open the insert cursor with the **OPEN** instruction. This instruction prepares the insert buffer. When the insert cursor is opened, you can add rows to the insert buffer with the **PUT** statement.

**Figure 35: OPEN and PUT statements**

Rows are automatically added to the database table when the insert buffer is full. To force row insertion in the table, you can use the **FLUSH** instruction.

**Figure 36: FLUSH statement**

Finally, when all rows are added, you can **CLOSE** the cursor and if you no longer need it, you can deallocate resources with the **FREE** instruction.
By default, insert cursors must be opened inside a transaction block, with BEGIN WORK and COMMIT WORK, and they are automatically closed at the end of the transaction. If needed, you can declare insert cursors with the WITH HOLD clause, to allow uninterrupted row insertion across multiple transactions.

**Related concepts**

Database transactions on page 688
Database transaction concepts and handling.

**DECLARE (insert cursor)**

The DECLARE with an INSERT instruction defines an insert cursor.

**Syntax**

```
DECLARE cid CURSOR [WITH HOLD] FOR insert-statement | sid
```

1. `cid` is the identifier of the insert cursor.
2. `insert-statement` is an INSERT statement defined in static SQL.
3. `sid` is the identifier of a prepared INSERT statement.

**Usage**

Use the DECLARE instruction with an INSERT instruction to define a new insert cursor in the current database session.

The INSERT statement is parsed, validated and the execution plan is created.

DECLARE must precede any other statement that refers to the cursor during program execution.

The scope of reference of the `cid` cursor identifier is local to the module where it is declared.

After declaring the insert cursor, it must be opened with the OPEN instruction.

The static `insert-statement` statement can include a list of variables in the VALUES clause. These variables are automatically read by the PUT statement; you do not have to provide the list of variables in that statement. As an alternative, use the ? (question mark) SQL parameter placeholder in the VALUE clause to bind program variables provided in the FROM clause of the PUT instruction.
When declaring a cursor with a prepared sid statement, the statement can include ? (question mark) placeholders for SQL parameters. In this case you must provide a list of variables in the FROM clause of the PUT statement.

Use the WITH HOLD option to declare cursors that have uninterrupted inserts across multiple transactions.

Resources allocated by the DECLARE can be released later by the FREE instruction.

The number of declared cursors in a single program is limited by the database server and the available memory. Make sure that you free the resources when you no longer need the declared insert cursor.

The identifier of a cursor that was declared in one module cannot be referenced from another module.

**Related concepts**

SQL execution diagnostics on page 591
If an SQL statement execution fails, error description can be found in the SQLCA.SQLCODE, SQLSTATE, STATUS and SQLERRMESSAGE predefined registers.

Database transactions on page 688
Database transaction concepts and handling.

### OPEN (insert cursor)

Initializes an insert cursor.

**Syntax**

```
OPEN cid
```

1. `cid` is the identifier of the insert cursor.

**Usage**

The OPEN statement initializes the insert cursor if the specified cursor was created with a DECLARE ... INSERT statement.

Once the insert cursor is opened, you can add rows with the PUT statement.

When used with an insert cursor, the OPEN instruction cannot include a USING clause.

A subsequent OPEN statement closes the cursor and then reopens it.

If the insert cursor was not declared WITH HOLD option, the OPEN instruction generates an SQL error if there is no current transaction started.

If you release cursor resources with a FREE instruction, you cannot use the cursor unless you declare the cursor again.

**Related concepts**

Database transactions on page 688
Database transaction concepts and handling.

### PUT (insert cursor)

Adds a new row to the insert cursor buffer.

**Syntax**

```
PUT cid FROM pvar [,...]
```

1. `cid` is the identifier of the insert cursor.
2. `pvar` is a variable containing an input value for the new row.
Usage
The **PUT** instruction adds a row to the insert cursor buffer.

If the insert cursor was not declared **WITH HOLD** option, the **PUT** instruction generates an SQL error if there is no current transaction started.

If the insert buffer has no room for the new row when the statement executes, the buffered rows are written to the database in a block, and the buffer is emptied. As a result, some **PUT** statement executions cause rows to be written to the database, and some do not.

**Related concepts**
- **OPEN (insert cursor)** on page 733
  - Initializes an insert cursor.
- **FLUSH (insert cursor)** on page 734
  - Flushes the buffer of an insert cursor.
- **Variables** on page 390
  - Explains how to define program variables.
- **Database transactions** on page 688
  - Database transaction concepts and handling.

**FLUSH (insert cursor)**
Flushes the buffer of an insert cursor.

**Syntax**

```
FLUSH cid
```

1. *cid* is the identifier of the insert cursor.

**Usage**

When flushing an insert cursor, all buffered rows are inserted into the target database table and the insert buffer is cleared.

The insert buffer may be automatically flushed by the runtime system if there no room when a new row is added with the **PUT** instruction.

**Related concepts**
- **CLOSE (insert cursor)** on page 734
  -Flushes and closes an insert cursor.

**CLOSE (insert cursor)**
Flushes and closes an insert cursor.

**Syntax**

```
CLOSE cid
```

1. *cid* is the identifier of the insert cursor.

**Usage**

Closing the insert cursor flushes automatically the rows remaining in the insert buffer, and releases the resources allocated for the insert buffer on the database server.

After using the **CLOSE** instruction, you must reopen the cursor with **OPEN** before adding new rows with **PUT/FLUSH**.
FREE (insert cursor)

Releases resources allocated for an insert cursor.

Syntax

```sql
FREE cid
```

1. `cid` is the identifier of the insert cursor.

Usage

After executing the `FREE` statement, all resources allocated to the insert cursor are released.

It is recommended that the cursor be explicitly closed with the `CLOSE` instruction, before it is freed.

If you release cursor resources with this instruction, you cannot use the cursor unless you declare the cursor again.

Related concepts

- DECLARE (insert cursor) on page 732

The `DECLARE` with an `INSERT` instruction defines an insert cursor.

Examples

Insert cursor usage examples.

Example 1: Insert Cursor declared with a Static INSERT

```plaintext
MAIN
  DEFINE i INTEGER
  DEFINE rec RECORD
    key INTEGER,
    name CHAR(30)
  END RECORD
  DATABASE stock
  DECLARE ic CURSOR FOR
    INSERT INTO item VALUES (rec.*)
  BEGIN WORK
    OPEN ic
    FOR i=1 TO 100
      LET rec.key = i
      LET rec.name = "Item #" || i
      PUT ic
      IF i MOD 50 = 0 THEN
        FLUSH ic
      END IF
    END FOR
  END WORK
  CLOSE ic
  COMMIT WORK
  FREE ic
END MAIN
```

Example 2: Insert Cursor declared with an SQL text

```plaintext
MAIN
  DEFINE i INTEGER
  DEFINE rec RECORD
    key INTEGER,
    name CHAR(30)
  END RECORD
  DATABASE stock
  DECLARE ic CURSOR FROM "INSERT INTO item VALUES (?,?)"
```
BEGIN WORK
  OPEN ic
  FOR i=1 TO 100
    LET rec.key = i
    LET rec.name = "Item #" || i
    PUT ic FROM rec.*
    IF i MOD 50 = 0 THEN
      FLUSH ic
    END IF
  END FOR
  CLOSE ic
  COMMIT WORK
  FREE ic
END MAIN

Example 3: Insert Cursor declared with 'hold' option

MAIN
  DEFINE name CHAR(30)
  DATABASE stock
  DECLARE ic CURSOR WITH HOLD FOR
    INSERT INTO item VALUES (1,name)
  OPEN ic
  LET name = "Item 1"
  PUT ic
  BEGIN WORK
    UPDATE refs SET name="xyz" WHERE key=123
  COMMIT WORK
  PUT ic
  PUT ic
  FLUSH ic
  CLOSE ic
  FREE ic
END MAIN

**SQL LOAD and UNLOAD**

Describes the instructions to export/import information from/to a database.

**LOAD**

Inserts data from a file into an existing database table.

**Syntax**

```
LOAD FROM filename [ DELIMITER delimiter ]
  [ INSERT INTO table-specification [ ( column [, ...] ) ] ]
  insert-string
```

where `table-specification` is:

```
[dbname[@dbserver]:][owner.]table
```

1. `filename` is a string expression containing the name of the file the data is read from.
2. `delimiter` is the character used as the value delimiter. Default is | pipe or DBDELIMITER environment variable when set.
3. The **INSERT** clause is a pseudo **INSERT** statement (without the **VALUES** clause), where you can specify the list of columns in parentheses.

4. **dbname** identifies the database name.

5. **dbserver** identifies the database server (INFORMIXSERVER).

6. **owner** identifies the owner of the table, with optional double quotes.

7. **table** is the name of the database table.

8. **column** is a name of a table column.

9. **insert-string** is a string expression containing the pseudo-**INSERT** statement.

**Usage**

The **LOAD** instruction reads serialized data from an input file and inserts new rows in a database table specified in the **INSERT** clause. A file created by the **UNLOAD** statement can be used as input for the **LOAD** statement if its values are compatible with the schema of **table**.

A data file used by **LOAD** or **UNLOAD** instructions looks like this (when the delimiter is the pipe), and the record has an **INTEGER**, **VARCHAR** and **DATE** fields:

| 102 | Mike | 12/24/2020 |
| 192 | Tom  | 04/29/2019 |

**Note:** Files encoded in UTF-8 can start with the UTF-8 Byte Order Mark (BOM), a sequence of 0xEF 0xBB 0xBF bytes, also known as UNICODE U+FEFF. When reading files, Genero BDL will ignore the UTF-8 BOM, if it is present at the beginning of the file. This applies to instructions such as **LOAD**, as well as I/O APIs such as `base.Channel.read()` and `readLine()`.

The **LOAD** statement must include a pseudo-**INSERT** statement (either directly or as text in a variable) to specify where to store the data. **LOAD** appends the new rows to the specified table, synonym, or view, but does not overwrite existing data. It cannot add a row that has the same key as an existing row.

Avoid use of the the **dbname**, **dbserver** and **owner** prefix of the table name for maximum SQL portability.

The number and the order of columns in the **INSERT** statement must match the values of the input file.

The **LOAD** instruction cannot be prepared with a **PREPARE** statement. However, **LOAD** can take a string literal as parameter, that allows to build the **INSERT** statement at runtime.

The variable or string following the **LOAD FROM** keywords must specify the name of a file of ASCII characters (or characters that are valid for the current locale) that holds the data values that are to be inserted.

Each set of data values in **filename** that represents a new row is called an input record. Each input record must contain the same number of delimited data values. If the **INSERT** clause has no list of columns, the sequence of values in each input record must match the columns of **table** in number and order. Each value must have the literal format of the column data type, or of a compatible data type.

If **LOAD** is executed within a **transaction block** (`BEGIN WORK / COMMIT WORK`), the rows inserted by the **LOAD** instruction are part of the transaction. With some database servers the insert rows remain locked until the **COMMIT WORK** or **ROLLBACK WORK** statement terminates the transaction. Consider locking the whole table to

If the database does not support transactions, a failing **LOAD** statement cannot remove any rows that were loaded before the failure occurred. You must manually remove the already loaded records from either the load file or from the receiving table, repair the erroneous records, and rerun **LOAD**.

If the database supports transactions, you can do the following actions:

- Run **LOAD** as a singleton transaction, so that any error causes the entire **LOAD** statement to be automatically rolled back.
- Run **LOAD** within an explicit `BEGIN WORK / COMMIT WORK` transaction block, so that a data error merely stops the **LOAD** statement in place with the transaction still open.

**Note:** When the **LOAD** instruction is not surrounded by **BEGIN WORK** and **COMMIT WORK** or **ROLLBACK WORK** instructions, terminating the transaction when **LOAD** is finished will automatically close cursors not defined WITH
HOLD option. To keep cursors open, either use the WITH HOLD option in DECLARE CURSOR, or surround the DECLARE, OPEN and LOAD instruction with an explicit BEGIN WORK/COMMIT WORK.

A single character delimiter instructs LOAD to read data in the default format. When using "CSV" as delimiter specification, the LOAD instruction will read the data in CSV format. If the DELIMITER clause is not specified, the delimiter is defined by the DBDELMITER environment variable. If the DBDELMITER environment variable is not set, the default is a | pipe. The field delimiter can be a blank character. It cannot be backslash or any hexadecimal digit (0–9, A–F, a–f). If the delimiter specified in the LOAD command is NULL, the runtime system will use the default delimiter or DBDELMITER if the variable is defined.

At this time, data type description of the input file fields is implicit; in order to create the SQL parameter buffers to hold the field values for inserts, the LOAD instruction uses the current database connection to get the column data types of the target table. Those data types depend on the type of database server. For example, IBM® Informix® DATE columns do not store the same data as the Oracle® DATE data type. Therefore, be careful when using the LOAD/UNLOAD instructions; if the application connects to different kinds of database servers, it can result data conversion errors.

Pay attention to numeric (DECIMAL, MONEY) and date/time values (DATE, DATETIME): These must match the current format settings (DBFORMAT, DBDATE). As a general programming pattern, use simple INSERT statements to load default and configuration data into your database, in order to be independent from the numeric and date format settings.

Default LOAD format

The next table describes the recommended representation for data values in the input file used by the LOAD instruction. Values must be serialized with a character string following the SQL data type of the receiving column of the table.

Table 197: Data representation for the default LOAD format

<table>
<thead>
<tr>
<th>Data type</th>
<th>Input Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR, VARCHAR, TEXT</td>
<td>Values can have more characters than the declared maximum length of the column, but any extra characters are ignored. A backslash () is required before any literal backslash or any literal delimiter character, and before any NEWLINE character anywhere in character value. Blank values can be represented as one or more blank characters between delimiters, but leading blanks must not precede other CHAR, VARCHAR, or TEXT values.</td>
</tr>
<tr>
<td>DATE</td>
<td>In the default locale, values must be in month/day/year format unless another format is specified by DBDATE environment variable. The day and month must be a 2-digit number, and the year must be a 4-digit number.</td>
</tr>
<tr>
<td>DATETIME</td>
<td>DATETIME values must be in the format: year-month-day hour:minute:second.fraction or a contiguous subset, without the DATETIME keyword or qualifiers. Time units outside the declared column precision can be omitted. The year must be a four-digit number; all other time units (except fraction) require two digits.</td>
</tr>
<tr>
<td>INTERVAL</td>
<td>INTERVAL values must be formatted: year-month or day hour:minute:second.fraction or a contiguous subset thereof, without the INTERVAL keyword or qualifiers. Time units outside the declared column precision can be omitted. All time units (except year and fraction) require two digits.</td>
</tr>
</tbody>
</table>
### SQL support

<table>
<thead>
<tr>
<th>Data type</th>
<th>Input Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>DECIMAL, MONEY</td>
<td>Values must use the decimal separator defined by DBFORMAT/DBMONEY. For MONEY, values can include currency symbols, but these are not required.</td>
</tr>
<tr>
<td>BYTE</td>
<td>Values must be ASCII-hexadecimals; no leading or trailing blanks.</td>
</tr>
<tr>
<td>SERIAL, BIGSERIAL, SERIAL8</td>
<td>Values can be represented as 0 to tell the database server to supply a new serial value. You can specify a literal integer greater than zero, but if the column has a unique index, an error results if this number duplicates an existing value.</td>
</tr>
</tbody>
</table>

The NEWLINE character must terminate each input record in `filename`. Specify only values that the language can convert to the data type of the database column. For database columns of character data types, inserted values are truncated from the right if they exceed the declared length of the column.

NULL values of any data type must be represented by consecutive delimiters in the input file; you cannot include anything between the delimiter symbols.

The `LOAD` statement expects incoming data in the format specified by environment variables like `DBFORMAT`, `DBMONEY`, `DBDATE`, `GL_DATE`, and `GL_DATETIME`. The precedence of these format specifications is consistent with forms and reports. If there is an inconsistency, an error is reported and the `LOAD` is canceled.

The backslash symbol (\) serves as an escape character in the input file to indicate that the next character in a data value is a literal. The `LOAD` statement scans for backslash escaped elements to read special characters in the following contexts:

- The backslash character appears anywhere in the value.
- The delimiter character appears anywhere in the value.
- The NEWLINE character appears anywhere in a value.

**CSV LOAD format**

The CSV (comma separated values) format is similar to the default format when using a simple comma delimiter, with the following differences:

- Input values might be surrounded with " double quotes.
- If an input value contains a comma or a NEWLINE, it is not escaped be the value must be quoted in the file.
- Double-quote characters in input values are doubled and will be converted to a unique " character; the value must be quoted.
- Backslash characters are not escaped in the input file and are read as; the value must be quoted.
- Leading and trailing blanks are kept (no truncation).
- No ending delimiter is expected at the end of the input record.

**Example**

```
MAIN
   DATABASE stores
   BEGIN WORK
   DELETE FROM items
   LOAD FROM "items01.unl" INSERT INTO items
   LOAD FROM "items02.unl" INSERT INTO items
   COMMIT WORK
END MAIN
```

**Related concepts**

[`UNLOAD` on page 740](#)
Copies data from the database tables into a file.

**UNLOAD**

Copies data from the database tables into a file.

**Syntax**

```
UNLOAD TO filename [DELIMITER delimiter]
  ↓
  ↓
select-statement
```  

1. `filename` is a string expression containing the name of the file the data is written to.
2. `delimiter` is the character used as the value delimiter. Default is | pipe or DBDELIMITER environment variable when set.
3. `select-statement` is static SELECT statement.
4. `select-string` is string expression containing the SELECT statement.

**Usage**

The UNLOAD instruction serializes into a file the SQL data produced by a SELECT statement.

A data file used by LOAD or UNLOAD instructions looks like this (when the delimiter is the pipe), and the record has an INTEGER, VARCHAR and DATE fields:

```
102|Mike|12/24/2020|
192|Tom|04/29/2019|
```

The UNLOAD command cannot be used in a PREPARE statement. However, the UNLOAD command accepts a string literal in place of a static SELECT statement:

```
UNLOAD TO file-name
  select-string
```

The `filename` after the TO keyword identifies an output file in which to store the rows retrieved from the database by the SELECT statement. In the default (U.S. English) locale, this file contains only ASCII characters. (In other locales, output from UNLOAD can contain characters from the codeset of the locale.)

The UNLOAD statement must include a SELECT statement (directly, or in a variable) to specify what rows to copy into `filename`. UNLOAD does not delete the copied data.

A single character delimiter instruct UNLOAD to write data in the default format. When using "CSV" as delimiter specification, the UNLOAD instruction will write the data in CSV format. If the DELIMITER clause is not specified, the delimiter is defined by the DBDELIMITER environment variable. If the DBDELIMITER environment variable is not set, the default is a | pipe. The field delimiter can be a blank character. It cannot be backslash or any hexadecimal digit (0–9, A–F, a–f). If the delimiter specified in the UNLOAD command is NULL, the runtime system will use the default delimiter or DBDELIMITER if the variable is defined.

When using a `select-string`, do not attempt to substitute question marks (?) in place of host variables to make the SELECT statement dynamic, because this usage has binding problems.

At this time, data type description of the output file fields is implicit; in order to create the fetch buffers to hold the column values, the UNLOAD instruction uses the current database connection to get the column data types of the generated result set. Those data types depend on the type of database server. For example, IBM® Informix® INTEGER columns are integers of 4 bytes, while the Oracle INTEGER data type is actually a NUMBER(10, 0) type. Therefore, be aware when using this instruction that if your application connects to different kinds of database servers, you may get data conversion errors.
Default UNLOAD format

A set of values in the output representing a row from the database is called an output record. A NEWLINE character (ASCII 10) terminates each output record.

The UNLOAD statement represents each value in the output file as a character string based on the current locale, depending on the data type of the database column:

Table 198: Default UNLOAD format

<table>
<thead>
<tr>
<th>Data type</th>
<th>Output Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR, VARCHAR, TEXT</td>
<td>Trailing blanks are dropped from CHAR and TEXT (but not from VARCHAR) values. A backslash () is inserted before any literal backslash or delimiter character and before a NEWLINE character in a character value.</td>
</tr>
<tr>
<td>DECIMAL, FLOAT, INTEGER, MONEY, SMALLFLOAT, SMALLINT</td>
<td>Values are written as literals with no leading blanks. MONEY values are represented with no leading currency symbol. Zero values are represented as 0 for INTEGER or SMALLINT columns, and as 0.00 for FLOAT, SMALLFLOAT, DECIMAL, and MONEY columns.</td>
</tr>
<tr>
<td>DATE</td>
<td>Values are written in the format month/day/year unless some other format is specified by the DBDATE environment variable.</td>
</tr>
<tr>
<td>DATETIME</td>
<td>DATETIME values are formatted year-month-day hour:minute:second.fraction or a contiguous subset, without DATETIME keyword or qualifiers. Time units outside the declared precision of the database column are omitted.</td>
</tr>
<tr>
<td>INTERVAL</td>
<td>INTERVAL values are formatted year-month or day hour:minute:second.fraction or a contiguous subset, without INTERVAL keyword or qualifiers. Time units outside the declared precision of the database column are omitted.</td>
</tr>
<tr>
<td>BYTE</td>
<td>BYTE Values are written in ASCII hexadecimal form, without any added blank or NEWLINE characters. The logical record length of an output file that contains BYTE values can be very long, and thus might be very difficult to print or to edit.</td>
</tr>
</tbody>
</table>

NULL values of any data type are represented by consecutive delimiters in the output file, without any characters between the delimiter symbols.

The backslash symbol (\) serves as an escape character in the output file to indicate that the next character in a data value is a literal. The UNLOAD statement automatically inserts a preceding backslash to prevent literal characters from being interpreted as special characters in the following contexts:

- The backslash character appears anywhere in the value.
- The delimiter character appears anywhere in the value.
- The NEWLINE character appears anywhere in a value.

CSV UNLOAD format

The CSV (comma separated values) format is similar to the standard format when using a simple comma delimiter, with the following differences:

- A comma character generates a quoted output value, and the comma is written as is (not escaped).
- A " double-quote character generate quoted output value and the quote in the value is doubled.
- NEWLINE characters generate a quoted output value, and the NEWLINE is written as is (not escaped).
- Backslash \ characters are written as is in the output value (i.e. not escaped).
- Leading and trailing blanks are not truncated in the output value.
- No ending delimiter is written at the end of the output record.

Example

```
MAIN
  DEFINE var INTEGER
  DATABASE stores
  LET var = 123
  UNLOAD TO "items.unl"
    SELECT * FROM items WHERE item_num > var
END MAIN
```

Related concepts
LOAD on page 736
Inserts data from a file into an existing database table.

SQL database guides

This section includes the SQL guides for various supported database servers.

The SQL guides provide you with information about installation and configuration requirements, as well as details on what is and is not supported when using database-specific SQL.

Note: The SQL adaptation guides focus specific features of the target database. Read first the General SQL programming topics, which cover good practices in database programming with Genero BDL.

IBM® Informix® Dynamic Server

For supported versions, see Supported IBM Informix server and CSDK versions on page 742

Purpose of the Informix® SQL guide

This section contains information to configure your Genero runtime system to work with an Informix® database engine, and describes the IBM® Informix® SQL features that are not supported (or partially supported) by Genero BDL.

Understand that Genero BDL was designed to work with IBM® Informix®: Most of the IBM® Informix® SQL features are supported. However, new features implemented in recent server versions need modifications in the Genero BDL compilers and runtime system to be supported.

Installation (Runtime Configuration)
ODI adaptation guide Installation topics.
Supported IBM® Informix® server and CSDK versions

Supported versions
Genero BDL is certified with all IBM® Informix® servers from version 5.x to the last available version, including the Standard Engine, On-Line and IDS server families, as long as the IBM® Informix® Client SDK is compatible with the server.

Genero BDL is certified with IBM® Informix® CSDK version 3.70 or higher.

Install IBM® Informix® and create a database - database configuration/design tasks

1. Install the IBM® Informix® database software (IDS for example) on your database server.
2. Install the IBM® Informix® Software Development Kit (SDK) on your application server.
With some IBM® Informix® distributions (IDS 11), this package is included in the server bundle. It is recommended that you check the IBM® web site for SDK upgrades or patches. Genero BDL is certified with IBM® Informix® SDK version 3.70 or higher.

3. Setup the IDS server (onconfig file, etc)
   a) Starting with IDS version 11, the TEMPTAB_NOLOG is set to 1 by default.

   Consider setting the TEMPTAB_NOLOG parameter to 0, if you want to log temporary table changes. This can affect the behavior of programs expecting that a ROLLBACK WORK cancels changes done on a temporary table.

   b) Starting with IDS version 11, by default the precision of SQL statement timing is the second (USEOSTIME is 0). For example, CURRENT HOUR TO FRACTION(3) returns a fraction part of zero.

   If sub-second precision is required, set the USEOSTIME configuration parameter to 1.

4. Define a database user dedicated to your application: the application administrator.

   This user will manage the database schema of the application (all tables will be owned by this user). With IBM® Informix®, database users reference Operating System users, and must be part of the IBM® Informix® group. See IBM® Informix® documentation.

5. Connect to the server as the "informix" user (for example with the dbaccess tool) and give all requested database administrator privileges to the application administrator.

   ```
   GRANT CONNECT TO appadmin;
   GRANT RESOURCE TO appadmin;
   GRANT DBA TO appadmin;
   ```

6. Login as the application administrator (appadmin), and setup the Informix client environment.

7. Define the client and database locale.

   The Informix® environment variables defining locale settings are:
   - CLIENT_LOCALE: defines the locale of the client application.
   - DB_LOCALE: defines the locale of the database (matters at DB creation!)
   - SERVER_LOCALE: defines the locale to be used by the server to interact with the OS file system.

   **Important:** Always define the DB_LOCALE environment variable, before executing the CREATE DATABASE statement: If DB_LOCALE is not set, it will default to the DB_LOCALE value set when starting the Informix engine. If no DB_LOCALE was set when starting the engine, it defaults to en_us.8859-1 (i.e. en_us.819).

8. Create a database entity, for example with the following SQL statement:

   ```
   CREATE DATABASE dbname WITH BUFFERED LOG;
   ```

   **Tip:** To check the locale of an Informix database, connect to the sysmaster database, and run the following query:

   ```
   $ dbaccess sysmaster -
   Database selected.
   > SELECT * FROM sysdbslocale WHERE dbs_dbsname = 'mydb';
   dbs_dbsname  mydb
   dbs_collate  en_US.819
   ```

   The charset code for ISO-8859-1 is 819, for UTF8 it is 57372.

   **Tip:** To check the value of the SQL_LOGICAL_CHAR size multiplier of an existing database, do the following query:

   ```
   $ dbaccess mydb -
   Database selected.
   > SELECT MOD(flags,4)+1 FROM systables WHERE tabname = ' VERSION';
   ```
9. Create the application tables.

**Prepare the runtime environment - connecting to the database**

1. In order to connect to IBM® Informix®, you must have a database driver "dbmifx" in $FGLDIR/dbdrivers.

2. Make sure the IBM® Informix® client environment variables are properly set.
   Check for example INFORMIXDIR (the path to the installation directory), INFORMIXSERVER (the name of the server defined in the sqlhosts list), etc. For more details, see the IBM® Informix® documentation.

3. In order to connect to an IBM® Informix® server, you must define a line in the $INFORMIXDIR/etc/sqlhosts file, referencing the server name specified in the INFORMIXSERVER environment variable.
   On Windows® platforms, the sqlhost entries are defined in the registry database. See IBM® Informix® documentation.

4. Verify the environment variable defining the search path for IBM® Informix® SDK database client shared libraries.

   **Table 199: Shared library environment setting for IBM® Informix® SDK version**

<table>
<thead>
<tr>
<th>IBM® Informix® SDK version</th>
<th>Shared library environment setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>All versions</td>
<td><code>UNIX</code>: Add $INFORMIXDIR/lib, $INFORMIXDIR/lib/esql,$INFORMIXDIR/lib/tools and $INFORMIXDIR/lib/cli to LD_LIBRARY_PATH (or its equivalent).</td>
</tr>
<tr>
<td></td>
<td><code>Windows®</code>: Add %INFORMIXDIR%\bin to PATH.</td>
</tr>
</tbody>
</table>

5. Check the Informix database client locale settings.
   The Informix database client locale must match the locale used by the runtime system (LC_ALL, LANG):
   
   The Informix® environment variables defining locale settings are:
   - CLIENT_LOCALE: defines the locale of the client application.
   - DB_LOCALE: defines the locale of the database (matters at DB creation!)
   - SERVER_LOCALE: defines the locale to be used by the server to interact with the OS file system.

6. To verify if the IBM® Informix® client environment is correct, you can start the SQL command interpreter:

   ```
   $ dbaccess - -
   > CONNECT TO "dbname" USER "appadmin";
   ENTER PASSWORD: password
   ```

7. Set up the FGLPROFILE entries for database connections.
   **Important**: Make sure that you are using the ODI driver corresponding to the database client and server version.

**Fully supported IBM® Informix® SQL features**

Fully supported IBM® Informix® SQL features.

**What are the supported IBM® Informix® SQL features?**

Genero BDL was first designed for IBM® Informix® database servers. The answer to this question is: Every SQL feature that is not listed in the other sections of this chapter.

The following list gives an idea of the IBM® Informix® SQL elements you can use with Genero BDL:

- Database connection control instructions (DATABASE, CONNECT). See [Connections](#), with DB user authentication.
- Transaction control instructions and concurrency settings (BEGIN WORK, SET ISOLATION). See [Transactions](#).
- Basic, portable data types (INT, BIGINT, DECIMAL, CHAR, VARCHAR, DATE, DATETIME, TEXT, BYTE, etc). See [data types](#).
• SERIAL, BIGSERIAL with last generated serial in SQLCA.SQLERRD[2] after INSERT.
• Common Data Definition Language statements (CREATE TABLE, DROP TABLE, etc). See Static SQL.
• Common Data Manipulation Language statements (SELECT, INSERT, UPDATE, DELETE, etc). See Static SQL.
• Cursors declared with SELECT ... FOR UPDATE, with or without the WITH HOLD option.
• Result set handling with cursors (DECLARE / OPEN / FETCH / CLOSE / FREE). See Result Sets.
• Positioned UPDATES and DELETEs (UPDATE/DELETE WHERE CURRENT OF). See Positioned Updates.
• Cursors to insert rows (DECLARE / OPEN / PUT / FLUSH). See Insert Cursors.
• Stored procedure calls. See SQL Programming.
• SQL statement interruption. See Using SQL interruption on page 601.
• SQL execution status and error messages (SQLCA, SQLSTATE). See Connections.
• Global Language Support with single and multibyte character sets for CHAR/ VARCHAR data storage. See Localization.
• LOAD and UNLOAD utility statements. See I/O SQL instructions.
• Database schema extraction to define program variables LIKE database columns. See Database Schema.

Using UTF-8 in IBM® Informix® databases

UTF-8 in Genero programs and Informix databases

Genero BDL programs can run in a UTF-8 locale, and use an IBM® Informix® database created with a UTF-8 locale.

The locale of an Informix database is defined at creation by the DB_LOCALE environment variable.

At runtime, the Informix client locale is defined by the CLIENT_LOCALE environment variable and must match the Genero application locale (LANG/LC_ALL), while DB_LOCALE must match the database locale. If needed, Informix will make charset conversions between CLIENT_LOCALE and DB_LOCALE. For more details about Informix client setup, see Prepare the runtime environment - connecting to the database on page 744.

When creating a database for UTF-8 storage:
• Define the DB_LOCALE environment variable to a UTF-8 locale like `en_us.utf8`
• Check the SQL_LOGICAL_CHAR parameter in the onconfig Informix server file.

Defining the Informix database locale

When you create a database with a CREATE DATABASE SQL statement, the current value of the DB_LOCALE environment variable defines the locale for the created database:

```
$ DB_LOCALE=en_US.utf8
$ export DB_LOCALE
$ dbaccess - -
> CREATE DATABASE mydb WITH BUFFERED LOG;
```

**Important:** Always define the DB_LOCALE environment variable, before executing the CREATE DATABASE statement: If DB_LOCALE is not set, it will default to the DB_LOCALE value set when starting the Informix engine. If no DB_LOCALE was set when starting the engine, it defaults to `en_us.8859-1` (i.e. `en_us.819`).

**Tip:** To check the locale of an Informix database, connect to the sysmaster database, and run the following query:

```
$ dbaccess sysmaster -
Database selected.
> SELECT * FROM sysdbslocale WHERE dbs_dbsname = 'mydb';
dbs_dbsname  mydb
dbs_collate  en_US.819
```

The charset code for ISO-8859-1 is 819, for UTF8 it is 57372.
The SQL_LOGICAL_CHAR onconfig parameter

Informix (14.10) does not support CHAR length semantics, it uses only BYTE length semantics: By default, in a CHAR(10) SQL type, the size represents a number of bytes. Since UTF-8 locale is a multi-byte, variable-length encoding, it is not possible to store 10 of all possible UTF-8 characters in 10 bytes.

However, the Informix onconfig SQL_LOGICAL_CHAR parameter can be defined as a size multiplier for CHAR/VARCHAR SQL types, when creating tables. For example, when using a CHAR(10) SQL type and SQL_LOGICAL_CHAR=4, the real SQL type will be CHAR(40), where 40 is the size in bytes.

Tip: To check the value of the SQL_LOGICAL_CHAR size multiplier of an existing database, do the following query:

```
$ dbaccess mydb -
Database selected.
> SELECT MOD(flags,4)+1 FROM systables WHERE tabname = ' VERSION';
4
```

Persistence of database settings

The DB_LOCALE and SQL_LOGICAL_CHAR values used at database creation are persistent properties for the lifetime of that database.

If you stop the Informix engine, change the SQL_LOGICAL_CHAR, then restart the engine, or set a different DB_LOCALE, the locale settings of existing databases remain unchanged: New created tables are based on the DB_LOCALE and SQL_LOGICAL_CHAR settings when the database was created.

Database schema and CHAR/VARCHAR sizes

Program variables can be declared with DEFINE LIKE, to use SQL types from .sch schema files.

Schema files are produced with the fgldbsch tool, reading Informix (syscolumns) system tables. The sizes of CHAR/VARCHAR columns is always expressed in bytes in syscolumns.collength.

For databases created with an SQL_LOGICAL_CHAR size multiplier, the fgldbsch tool divides the collength by the SQL_LOGICAL_CHAR value used by the database.

For example:

1. At database creation, onconfig server file has SQL_LOGICAL_CHAR=4
2. CREATE TABLE tab1 ( col1 CHAR(10) )
3. In the Informix system table: syscolumns.collength = 40 (40 bytes)
4. fgldbsch -db mydb produces the .sch file (for tab1.col1: 40/4=10)
5. In .sch schema file, tab1.col1 is defined as CHAR(10)
6. Source code declares: DEFINE rec RECOR LIKE tab1.*
7. The compiler reads schema file and defines rec.col1 as CHAR(10)

At runtime, the actual storage capabilities of the CHAR(10) variable is then defined by the FGL_LENGTH_SEMANTICS environment variable:

- When FGL_LENGTH_SEMANTICS=CHAR, a CHAR(10) variable can store 10 UTF-8 characters, no matter the number of bytes used by these characters.
- When FGL_LENGTH_SEMANTICS=BYTE, a CHAR(10) variable can store 10 ASCII-7 characters, 5 (5x2b=10b) UTF-8 accute characters, or 3 (3x3b=9b) UTF-8 Chinese characters.

FGL_LENGTH_SEMANTICS: CHAR or BYTE?

With UTF-8 in Genero programs, it is recommended to use char length semantics by setting the FGL_LENGTH_SEMANTICS=CHAR environment variable.
However, depending on the level of compatibility with Informix SQL length semantics, and the complexity of character string operations in the program's code, you need to choose CHAR or BYTE length semantics for Genero programs.

If Informix would support CHAR length semantics, there would be no question: We should just use FGL_LENGTH_SEMANTICS=CHAR as well.

But Informix fundamentally uses BYTE length semantics (even when SQL_LOGICAL_CHAR is defined). For example, in SQL statements:

- \( \text{LENGTH} (\text{expr}) \) returns a number of bytes
- \( \text{custname}[2,5] \) means the substrings from byte position 2 to 5

If the application code does not have complex string operations, you may consider using FGL_LENGTH_SEMANTICS=BYTE, to get the same length semantics as in SQL statements.

But when the application code uses a lot of substring positions such as \( \text{custcode}[2,5] \), it is mandatory to use FGL_LENGTH_SEMANTICS=CHAR, to keep the source code untouched.

Note: The fgldbsch tool takes database's SQL_LOGICAL_CHAR setting into account: A column created as CHAR(10) will become a CHAR(10) variable from DEFINE LIKE (even if syscolumns.collength is defined as 40 bytes). Thus, the schema file is prepared for FGL_LENGTH_SEMANTICS=CHAR usage.

The future of Informix UTF-8 support

Ideally, Informix should support CHAR length semantics like most other databases, to get the same behavior with SQL and FGL expressions.

This feature request is known to Informix developers and should be available in a future version (> 14.10).

Related concepts

- **CHAR(size)** on page 293
  The CHAR data type is a fixed-length character string data type.
- **VARCHAR(size)** on page 309
  The VARCHAR data type is a variable-length character string data type, with a maximum size.
- **TEXT** on page 307
  The TEXT data type stores large text data.

Stored procedure calls

IBM® Informix® stored procedures are written in the SPL, C or Java programming languages, also known as User Defined Routines.

To create a stored procedure in an IBM® Informix® database, use the CREATE FUNCTION statement. See reference documentation for more details.

Stored functions returning values

To return values from an IBM® Informix® SPL routine, execute the routine and fetch the output values, as you would for a regular SELECT statement producing a result set.

Note: Informix® distinguishes between stored functions from stored procedures. Only stored functions (with a RETURNING clause) can return values. Stored procedures do not return values.

To execute an Informix® stored function from a BDL program, use the EXECUTE FUNCTION SQL instruction:

```
PREPARE stmt FROM "execute function procl(?)";
```

In order to retrieve returning values into program variables, use an INTO clause in the EXECUTE instruction.

This example shows how to call a stored function:

```
MAIN
```
**DATABASE** test1

**EXECUTE IMMEDIATE** "create function proc1( p1 integer )"
" returning decimal(6,2), varchar(200);"
" define p2 decimal(6,2);"
" define p3 varchar(200);"
" let p2 = p1 + 0.23;"
" let p3 = 'Value = ' || p1;"
" return p2, p3;"
" end function;"

**PREPARE** stmt FROM "execute function proc1(?)"

**LET** n = 111

**EXECUTE** stmt USING n INTO d, c

**DISPLAY** d

**DISPLAY** c

**END MAIN**

---

**Stored functions defined with output parameters**

Starting with IDS 10.00, IBM® Informix® introduced the concept of output parameters for stored functions.

To retrieve the output parameters, you must execute the routine in a SELECT statement defining Statement Local Variables. These variables will be listed in the select clause to be fetched as regular column values. See Informix® documentation for more details.

In order to retrieve returning values into program variables, use an INTO clause in the EXECUTE instruction.

This example shows how to call a stored function with output parameters:

**MAIN**

**DEFINE** pi, pr INTEGER

**DATABASE** test1

**EXECUTE IMMEDIATE** "create function proc2(i INT, OUT r INT)"
" returning int;"
" let r=i+10;"
" return 1;"
" end function;"

**PREPARE** s FROM "select r from systables where tabid=1 and proc2(?,#int)==1"

**LET** pi = 33

**EXECUTE** s USING pi INTO pr

**DISPLAY** "Output value: ", pr

**EXECUTE** IMMEDIATE "drop function proc2"

**END MAIN**

---

**Stored functions returning a result set**

To retrieve the rows of a result set produced by an IBM® Informix® stored function, you must create a cursor, as you would for a regular SELECT statement.

This example shows how to execute a stored function producing a result set:

**MAIN**

**DEFINE** m, p_pk INT, p_name VARCHAR(10)

**DATABASE** test1

**CREATE TABLE** t1 ( pk INT, name VARCHAR(10) )

**INSERT INTO** t1 VALUES (1, 'aaaa')

**INSERT INTO** t1 VALUES (2, 'bbbbbb')

**INSERT INTO** t1 VALUES (3, 'cccc')

**EXECUTE IMMEDIATE** "create function proc3(v_max INT)"
" returning int, lvarchar;"
" define r_pk integer;"
Partially supported IBM® Informix® SQL features

The **BIGSERIAL / SERIAL8** data types

IBM® Informix® supports the BIGSERIAL and SERIAL8 data types for auto-generated 64 bit integer sequences.

The **BIGINT** data type can be used to store data from BIGSERIAL SERIAL8 values.

Note that SQLCA.SQLERRD[2] is defined as an INTEGER and therefore cannot be used to get the last generated serial. To retrieve the last generated BIGSERIAL or SERIAL8, you must use the `dbinfo()` SQL function as in the following code example:

```sql
MAIN
DEFINE new_val BIGINT
INSERT INTO mytable VALUES ( 0, 'aaaa' )
SELECT dbinfo('bigserial') INTO new_val
  FROM systables WHERE tabid=1
DISPLAY new_val
END MAIN
```

Related concepts

The [SQLCA diagnostic record](#) on page 594

The SQLCA variable is a predefined record containing SQL statement execution information.

[SERIAL and BIGSERIAL data types](#) on page 931

The **NCHAR / NVARCHAR** data types

IBM® Informix® supports the standard NCHAR and NVARCHAR data types. These types are equivalent to CHAR and VARCHAR (the same character set is used), except that the collation order is locale specific with NCHAR/NVARCHAR types.

Genero BDL syntax does allow to define program variables by using NCHAR / NVARCHAR keywords. However, the character strings of Informix NCHAR/NVARCHAR database columns can be managed by program variables defined with the CHAR/VARCHAR types.

**Note:** Since the character set is identical for NCHAR/NVARCHAR and CHAR/VARCHAR columns in an Informix database, no specific consideration needs to be given for the "N" character types.

When extracting a database schema with `fgldbsch`, NCHAR/NVARCHAR types will be identified in the .sch file by the native Informix® type codes 15 and 16. When compiling .4gl or .per sources referencing NCHAR/NVARCHAR columns in the schema file, the compilers will automatically use the CHAR/VARCHAR Genero BDL types for the type codes 15 and 16.

Related concepts

[Primitive Data types](#) on page 289
Selecting the correct data type assists you in the input, storage, and display of your data.

**The LVARCHAR data type**

IBM® Informix® supports the LVARCHAR type as a "large" VARCHAR type. The LVARCHAR type was introduced to bypass the 255 bytes size limitation of the standard VARCHAR type. Starting with IDS version 9.4, the LVARCHAR size limit is 32739 bytes. In older versions the limit was 2048 bytes.

Genero BDL does not support the LVARCHAR type natively, but it has the VARCHAR type which can hold up to 65535 bytes. IBM® Informix® LVARCHAR values can be inserted or fetched by using the BDL VARCHAR type.

Static SQL statements such as CREATE TABLE can include the LVARCHAR column type.

When extracting a schema with fgdbsch, LVARCHAR(N) columns will by default be converted to VARCHAR2(N) in the schema file. VARCHAR2 is a Genero BDL-only pseudo type identified with the type code 201 that allows for VARCHAR variables with a size that can be greater than 255 bytes to be defined.

**Related concepts**

Related concepts
- **Primitive Data types** on page 289
Selecting the correct data type assists you in the input, storage, and display of your data.

**DISTINCT data types**

IBM® Informix® supports DISTINCT data types as User Defined Types based on a source data type, but with different casts and functions than those on the source data type.

Genero BDL partially supports the IBM® Informix® DISTINCT data types:

The fgdbsch schema extractor can extract columns defined with a distinct type and write the distinct type code in the .sch schema file. For more details, see the list of distinct types in the Column Definition File (.sch) on page 470

However, there are some restrictions you must be aware of:

- It is not possible to define BDL variables explicitly with the name of a distinct type. Variables must be defined indirectly with the schema by using the DEFINE LIKE statement.
- The static SQL syntax does not support OPAQUE-related syntax elements:
  - The DDL statements CREATE DISTINCT TYPE, DROP TYPE, CREATE CAST, and DROP CAST are not allowed,
  - In CREATE TABLE / ALTER TABLE DDL statements, the data type must be a built-in type.
  - The :: cast operator is not supported.

**Stored Procedures**

With IBM® Informix® database servers, you can write stored procedures with the SPL (Stored Procedure Language) or with an external language in C or JAVA.

If you plan to support different types of database servers, you must be aware that each DB vendor has defined its own stored procedure language. In such cases, you may consider writing most of your business logic in BDL, and implementing only some stored procedures in the database, mainly to get better performance or to use database features that only exist with stored procedures.

Genero BDL partially supports SP creation, but has full support of SP invocation:

- The Genero BDL static SQL syntax does not include CREATE FUNCTION and CREATE PROCEDURE with a body block. However, you can create stored procedures with a body block by using dynamic SQL (EXECUTE IMMEDIATE), or with CREATE PROCEDURE and the FROM filename clause, which is supported by Genero BDL static SQL.
- The EXECUTE FUNCTION or EXECUTE PROCEDURE instruction is not allowed in the static SQL syntax. To invoke a stored procedure with Informix®, you must use the PREPARE instruction, followed by EXECUTE or OPEN. The PREPARE instruction must initiate the EXECUTE FUNCTION/PROCEDURE instruction.

For more details about stored procedure invocation, see SQL Programming.
Related concepts

Stored procedure calls on page 747
Describes static SQL statements supported in the language.

Dynamic SQL management on page 709
Explains how to execute and manage SQL statements at runtime.

PREPARE (SQL statement) on page 710
Prepares an SQL statement for execution.

Database Triggers

Triggers can be created for IBM® Informix® database tables with the CREATE TRIGGER instruction.

If you plan to support different types of database servers, you must be aware that each DB vendor has defined its own trigger creation syntax and stored procedure language. In such cases, you may consider writing most of your business logic in BDL, and implementing only some triggers in the database, mainly to get better performance or use database features that only exist with stored procedures.

Genero BDL partially supports trigger creation:

• The Genero BDL static SQL syntax does not include the CREATE TRIGGER and DROP TRIGGER instructions. However, you can create database triggers by using dynamic SQL (EXECUTE IMMEDIATE).

Related concepts

Static SQL statements on page 694
Describes static SQL statements supported in the language.

EXECUTE IMMEDIATE on page 713
Performs a simple SQL execution without SQL parameters or result set.

Optimizer directives

IBM® Informix® SQL allows you to specify query optimization directives to force the query optimizer to use a different path than the implicit plan. With IBM® Informix®, optimizer directives are specified with the following SQL comment markers followed by a plus sign:

```sql
/++ optimizer-directives */
( optimizer-directives )
--- optimizer-directives
```

Genero BDL partially supports optimizer directives:

• The static SQL syntax does not allow the C-style optimizer syntax.
• The curly-brace and dash-dash optimizer directive syntaxes cannot be used in static SQL statements, because these correspond to the 4GL language comments.
• However, you can execute queries with optimization directives with Dynamic SQL.

Tip: Optimization directives are not portable. If you plan to use different types of database servers, it is recommended that you avoid the usage of query plan hints.

Related concepts

Static SQL statements on page 694
Describes static SQL statements supported in the language.

XML publishing support

IBM® Informix® IDS 11.10 introduced a set of XML built-in functions when the idsxmlvlp virtual processor is turned on. Built-in XML functions are of two types: those returning LVARCHAR values, and those returning CLOB values. For example, genxml() returns an LVARCHAR(32739), while genxmlclob() returns a CLOB. XML data is typically stored in LVARCHAR or CLOB columns.
Genero BDL partially supports XML functions:

- Because Genero BDL does not support BLOB/CLOB types, functions returning CLOB values cannot be used. You can however use the XML functions returning LVARCHAR values, and fetch the result into a VARCHAR variable of the appropriate size.
- Some of the XML functions such as genxml() take ROW() values as parameters. Because literal unnamed ROW() expressions are like regular function calls, you can use XML functions in static SQL statements.

Example:

```sql
FUNCTION get_cust_data(id)
    DEFINE id INT, v VARCHAR(5000)
    SELECT genxml(ROW(cust_name, cust_address), "custdata") INTO v
    FROM customers WHERE cust_id = id
    RETURN v
END FUNCTION
```

Related concepts

Static SQL statements on page 694
Describes static SQL statements supported in the language.

DataBlade® modules

IBM® Informix® IDS provides several database extensions implemented with the DataBlade® Application Programming Interface, such as MQ Messaging, Large OObjects management, Text Search DataBlades, Spatial DataBlade® Module, etc.

Genero BDL partially supports DataBlade® modules:

- DataBlade® extensions are based on User Defined Functions and User Defined Types. It is not possible to define program variables with specific User Defined Types. For example, you cannot define a program variable with the ST_Point type implemented by the Spatial DataBlade® module.
- The static SQL grammar does not support DataBlade® specific syntax. For example, it is not possible to create a Basic Text Search index with the USING bts clause of the CREATE INDEX statement.

However, as long as the syntax of the DataBlade® functions follows basic SQL expressions, it can be used in static SQL statements. For example, this query uses the bts_contains() function of the Basic Text Search extension:

```sql
SELECT id FROM products WHERE bts_contains( brands, 'standard' )
```

You can also use Dynamic SQL to perform queries with a syntax that is not allowed in the static SQL grammar.

Related concepts

Static SQL statements on page 694
Describes static SQL statements supported in the language.

Specific CREATE INDEX clauses

In addition to the standard index-key specification using a column list, the CREATE INDEX statement supported by IBM® Informix® SQL allows specific clauses, for example to define storage options.

Genero BDL partially supports the CREATE INDEX statement; the following are not supported in static SQL grammar:

- The IF NOT EXISTS clause.
- Functional index specification is not allowed in the index-key list.
- Storage options such as IN dbspace, EXTEND SIZE, NEXT SIZE.
- The index mode clauses such as FILTERING WITH/WITHOUT ERROR.
- The USING clause.
- The HASH ON clause.
- The FILLFACTOR clause.
You can use **Dynamic SQL** to execute CREATE INDEX statements with clauses that are not allowed in the static SQL grammar.

**Related concepts**

*Static SQL statements* on page 694

Describes static SQL statements supported in the language.

**Other SQL instructions**

Genero BDL static SQL syntax implements common Data Manipulation Statements such as SELECT, INSERT, UPDATE and DELETE. Data Definition Language statements such as CREATE TABLE, CREATE INDEX, CREATE SEQUENCE and their corresponding ALTER and DROP statements are also part of the static SQL grammar. These are supported with a syntax limited to the standard SQL clauses. For example, Genero BDL might not support the most recent CREATE TABLE storage options supported by IBM® Informix® SQL.

Since the first days of the Informix® 4GL language, the SQL language has been extended, and it has become so large that it’s impossible to embed all the existing new statements without introducing grammar conflicts with the I4GL language. In addition, each DB vendor has improved the standard SQL language with proprietary SQL statements that are not portable; it would not be a good idea to use these specific instructions if you plan to make your application run with different types of database engines.

However, the Genero BDL static SQL is constantly being improved with standard SQL syntax that works with most types of database servers. For example, Genero BDL supports the ANSI outer join syntax, constraints definition in DDL statements, sequence instructions, BIGINT and BOOLEAN data types, and there is more to come.

If a statement is unsupported in static SQL, that does not mean that you cannot execute it. If you want to execute an SQL instruction that is not part of the static SQL grammar, you can use **Dynamic SQL** as follows:

- Use **PREPARE + EXECUTE** for statements that do not generated a result set
- Use **(PREPARE/) DECLARE + OPEN** for statements returning a result set
- Use **EXECUTE IMMEDIATE** if no SQL parameters are required and no result set is generated

Dynamic SQL instructions take a string as the input, so there is no limitation regarding the SQL text you can execute. However, only one statement can be executed at a time. Preferably, write your SQL statements directly in static SQL when possible, because it makes the code more readable and the syntax is checked at compiled time.

For more details about statements supported in the static SQL syntax, see *Static SQL*.

Below is a list of the IBM® Informix® SQL statements that are not allowed in the static SQL syntax (last updated from IDS 11.50 SQL instructions). The IBM® Informix® SQL Syntax manual includes ESQL/C specific statements such as ALLOCATE DESCRIPTOR, which are not part of the basic SQL statements supported by the engines. ESQL/C specific statements are not listed here:

- ALTER ACCESS_METHOD
- ALTER FRAGMENT
- ALTER FUNCTION
- ALTER PROCEDURE
- ALTER ROUTINE
- ALTER SECURITY LABEL COMPONENT
- CREATE ACCESS_METHOD
- CREATE AGGREGATE
- CREATE CAST
- CREATE DISTINCT TYPE
- CREATE EXTERNAL TABLE Statement
- CREATE FUNCTION (with body)
- CREATE OPAQUE TYPE
- CREATE OPCLASS
- CREATE PROCEDURE (with body)
- CREATE ROLE
- CREATE ROUTINE FROM
- CREATE ROW TYPE
- CREATE SCHEMA
- CREATE SECURITY LABEL
Unsupported IBM® Informix® SQL features

Unsupported IBM® Informix® SQL features.

CLOB and BLOB data types

In addition to the TEXT and BYTE data types (known as Simple Large Objects), IBM® Informix® servers support the CLOB and BLOB types to store large objects. CLOB/BLOB are known as Smart Large Objects. The main difference is that Smart Large Objects support random access to the data - seek, read and write through the LOB as if it was an OS file.

Genero BDL does not support the CLOB and BLOB types:
• It is not possible to define BDL variables with the CLOB or BLOB types, so you cannot manipulate CLOB/BLOB objects within programs.
• Defining a TEXT / BYTE variable to hold CLOB / BLOB column data is not supported; you will get error -609 (Illegal attempt to use a Text/Byte host variable).
• The static SQL syntax for DDL statements like CREATE TABLE does not allow the CLOB / BLOB keywords for column types.
• The fgldbsch schema extractor will report an invalid data type if you try to get the schema for a table with a CLOB or BLOB column.

You can, however:
• Create a table with CLOB/BLOB columns by using Dynamic SQL.
• Use the Smart Large Object functions FILETOBLOB(), FILETOCLOB(), LOCOPY(), LOTOFIILE() in static SQL statements.

Related concepts
Static SQL statements on page 694
Describes static SQL statements supported in the language.

The LIST data type
With IBM® Informix®, the LIST type is a collection type that can store ordered elements of a specific base type. Unlike the MULTISET type, the elements of a LIST have ordinal positions. Elements can be duplicated.
Genero BDL does not support the IBM® Informix® LIST data type.
• It is not possible to define BDL variables with the LIST type.
• The static SQL syntax does not support collection-related syntax elements:
  • DDL statements like CREATE TABLE cannot use the LIST keyword for column types,
  • The collection-derived notation TABLE() is not allowed,
  • The INSERT AT position instruction is not supported,
  • The LIST { } literal syntax is not allowed.
  • The value IN identifier syntax is not allowed.
• The fgldbsch schema extractor will report an invalid data type if you try to get the schema for a table with a LIST column.

Related concepts
Static SQL statements on page 694
Describes static SQL statements supported in the language.

The MULTISET data type
The MULTISET IBM® Informix® data type is a collection type that can store non-ordered elements of a specific base type. Unlike the LIST type, the elements of a MULTISET have no ordinal positions. Elements can be duplicated.
Genero BDL does not support the IBM® Informix® MULTISET data type:
• It is not possible to define BDL variables with the MULTISET type.
• The static SQL syntax does not support collection-related syntax elements:
  • DDL statements like CREATE TABLE cannot use the MULTISET keyword for column types,
  • The collection-derived notation TABLE() is not allowed,
  • The MULTISET { } literal syntax is not allowed.
  • The value IN identifier syntax is not allowed.
• The fgldbsch schema extractor will report an invalid data type if you try to get the schema for a table with a MULTISET column.

Related concepts
Static SQL statements on page 694
Describes static SQL statements supported in the language.

The SET data type

The SET IBM® Informix® data type is a collection type that stores non-ordered unique elements of a specific base type. Unlike the LIST type, the elements of a LIST have no ordinal positions. Elements cannot be duplicated.

Genero BDL does not support the IBM® Informix® SET data type:

• It is not possible to define BDL variables with the SET type.
• The static SQL syntax does not support collection-related syntax elements:
  • DDL statements like CREATE TABLE cannot use the SET keyword for column types,
  • The collection-derived notation TABLE() is not allowed,
  • The SET { } literal syntax is not allowed.
  • The value IN identifier syntax is not allowed.
• The fgldbsch schema extractor will report an invalid data type if you try to get the schema for a table with a SET column.

Related concepts
Static SQL statements on page 694
Describes static SQL statements supported in the language.

The ROW data types

IBM® Informix® supports the named and unnamed ROW data types. A ROW type is a complex type that combines several table columns. You create a ROW type with the CREATE ROW TYPE instruction, and then you can reuse the type definition for a table column.

Genero BDL does not support the IBM® Informix® ROW data types:

• It is not possible to define BDL variables with a named ROW type. The equivalent would be a RECORD variable, but data is not mapped directly from a structured ROW column, you must list individual fields of the ROW column.
• The static SQL syntax does not support ROW-related syntax elements:
  • The DDL statements CREATE ROW TYPE, DROP ROW TYPE, CREATE CAST and DROP CAST are not allowed,
  • In CREATE TABLE / ALTER TABLE DDL statements, the data type must be a built-in type.
  • The :: cast operator is not supported when specifying a ROW() literal. However, the CAST() expressions are allowed.
• The fgldbsch schema extractor will report an invalid data type if you try to get the schema for a table with a column defined with a ROW type.

However:

• Static SQL allows multilevel single-dot notation, so you can, for example, identify a ROW field as employee.address.city.
• Dynamic SQL can be used to insert or update rows with ROW type columns.
• Individual ROW column fields can be fetched to BDL program variables, as long as the basic types match.

Related concepts
Static SQL statements on page 694
Describes static SQL statements supported in the language.

Dynamic SQL management on page 709
Explains how to execute and manage SQL statements at runtime.

**OPAQUE data types**

Opaque User Defined Types can be implemented in IBM® Informix® with the CREATE OPAQUE TYPE statement. The storage structure of an OPAQUE type is unknown to the database server, data can only be accessed through user-defined routines.

Genero BDL does not support the IBM® Informix® OPAQUE data types:

- It is not possible to define BDL variables with an opaque type.
- The static SQL syntax does not support OPAQUE-related syntax elements:
  - The DDL statements CREATE OPAQUE TYPE, DROP TYPE, CREATE CAST and DROP CAST are not allowed,
  - In CREATE TABLE / ALTER TABLE DDL statements, the data type must be a built-in type.
  - The :: cast operator is not supported. However, the CAST() expressions are allowed.
- The fgldbsch schema extractor will report an invalid data type if you try to get the schema for a table with a column defined with a OPAQUE type.

**Related concepts**

[Static SQL statements](#) on page 694

Describes static SQL statements supported in the language.

**The :: cast operator**

IBM® Informix® SQL implements the :: cast operator and the CAST() expressions to do an explicit cast of a value:

```
CREATE TABLE tab ( v INTEGER )
INSERT INTO tab VALUES ( 123456::INTEGER )
SELECT 'abcdef'::CHAR(20)||'.' FROM tab
SELECT CAST('abcdef' AS CHAR(20))||'.' FROM tab
```

Genero BDL does not support the :: cast operator in the static SQL grammar. However, the CAST() expressions are allowed. If you need to use the :: cast operator, you must use Dynamic SQL to perform such queries.

**Related concepts**

[Static SQL statements](#) on page 694

Describes static SQL statements supported in the language.

**Table inheritance**

IBM® Informix® SQL allows you to define a table hierarchy through named row types. Table inheritance allows a table to inherit the properties of the supertable in the meaning of constraints, storage options, triggers. You must first create the types with CREATE ROW TYPE, then you can create the tables with the UNDER keyword to define the hierarchy relationship.

```
CREATE ROW TYPE person_t ( name VARCHAR(50) NOT NULL, 
                        address VARCHAR(200), birthdate DATE )
CREATE ROW TYPE employee_t ( salary INTEGER, manager VARCHAR(50) )
CREATE TABLE person OF TYPE person_t
CREATE TABLE employee OF TYPE employee_t UNDER person
```

A table hierarchy allows you to do SQL queries whose row scope is the supertable and its subtables. For example, after inserting one row in the person table and another one in the employee table, if you UPDATE the name column without a WHERE clause, it will update all rows from both tables. To limit the set of rows affected by the statement to rows of the supertable, you must use the ONLY keyword:

```
UPDATE ONLY(person) SET birthdate = NULL
SELECT * FROM ONLY(person)
```
Genero BDL static SQL grammar does not include the syntax elements related to table hierarchy management. You can however use Dynamic SQL to perform such queries.

**Related concepts**

Static SQL statements on page 694
Describes static SQL statements supported in the language.

**IBM® DB2® Linux-Unix-Windows**

**Supported versions**

Genero BDL supports the following IBM® DB2® LUW versions:

- IBM DB2 LUW 10.5
- IBM DB2 LUW 11.x

**Installation (Runtime Configuration)**

IBM® DB2® related installation topics.

*Install IBM® DB2® and create a database - database configuration/design tasks*

If you are tasked with installing and configuring the database, here is a list of steps to be taken:

1. Install the IBM® DB2® LUW on your database server.
2. Create a DB2® database entity: `dbname`

   To create the database entity in DB2, use the `db2` command interpreter in a DB2 operating system user session (`db2inst`).

   **Note:** Consider creating your database with the correct database locale (codeset and territory)

   For example:

   ```
   $ db2
   ...
   db2 => CREATE DATABASE dbname
   AUTOMATIC STORAGE YES
   USING CODESET UTF-8 TERRITORY EN_US
   DB20000I The CREATE DATABASE command completed successfully.
   ```

3. Connect to the new created database with the DB2 administrator user.

   Open a database connection as in the following example:

   ```
   db2 => connect to dbname
   Database Connection Information
   Database server = DB2/LINUX 10.1.0
   SQL authorization ID = DB2INST
   Local database alias = dbname
   ```

4. Declare a database user dedicated to your application: the application administrator. This user will manage the database schema of the application (all tables will be owned by it).

   Create the user as follows:

   ```
   db2 => GRANT CONNECT ON DATABASE TO USER appadmin
   DB20000I The SQL command completed successfully.
   ```

5. Give all requested database administrator privileges to the application administrator.

   Grant the privileges to create tables to the new created user as follows:

   ```
   db2 => GRANT CREATETAB ON DATABASE TO USER appadmin
   ```
DB20000I  The SQL command completed successfully.

Or, provide full database administrator privileges to the new created user:

```
db2 => GRANT DBADM ON DATABASE TO USER appadmin
DB20000I  The SQL command completed successfully.
```

6. If you plan to use temporary table emulation, you must setup the database for DB2® global temporary tables (create a user temporary tablespace and grant privileges to all users).

See Temporary tables on page 785.

7. Connect as the application administrator:

   Open a new database connection:

```
db2 => connect to dbname user appadmin using password
```

Database Connection Information

<table>
<thead>
<tr>
<th>Database server</th>
<th>DB2/LINUX 10.1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL authorization ID</td>
<td>DB2INST</td>
</tr>
<tr>
<td>Local database alias</td>
<td>dbname</td>
</tr>
</tbody>
</table>

8. Create the application tables with CREATE TABLE statements.

   Convert Informix® data types to DB2® data types. See issue Data Type Conversion Table for more details.

9. If you plan to use SERIAL column emulation, you must prepare the database.

   See SERIAL and BIGSERIAL data types on page 774.

Prepare the runtime environment - connecting to the database

1. In order to connect to IBM® DB2®, you must have the database driver "dbmdb2" in $FGLDIR/dbdrivers.

2. If you want to connect to a remote DB2® server, the "IBM® DB2® Client Application Enabler" must be installed and configured on the computer running the BDL applications.

   You must declare the data source set up as follows:

   a) Login as root.

      1. Create a user dedicated to the DB2® client instance environment, for example, "db2cli1".
      2. Create a client instance environment with the db2icrt tool as in following example:

```
    # db2dir /instance/db2icrt -a server -s client instance-user
```

   b) Login as the instance user (the environment is set automatically, verify DB2DIR).

      1. Catalog the remote server node:

```
    # db2 "catalog tcpip node db2node remote hostname server tcp-service"
```

      2. Catalog the remote database:

```
    # db2 "catalog database datasource at node db2node authentication server"
```

      3. Test the connection to the remote database:

```
    # db2 "connect to datasource user dbuser using password"
```

   (where dbuser is a database user declared on the remote database server)

See IBM® DB2® documentation for more details.
3. **Important:** If you have a non-English environment, you may need to set the PATCH2=15 configuration parameter in the db2cli.ini file, to ensure that DECIMAL values will be properly inserted or fetched:

```
[datasource]
PATCH2=15
```

For more details, see the DB2® README.TXT file in the SQLLIB directory.

4. Make sure that the DB2® client environment variables are properly set.

Check variables such as DB2DIR (the path to the installation directory), DB2INSTANCE (the name of the DB2® instance), INSTHOME (the path to the home directory of the instance owner). On UNIX™, you will find environment settings in the file $INSTHOME/sqllib/db2profile. See IBM® DB2® documentation for more details.

5. Check the database client locale settings (DB2CODEPAGE, etc).

The database client locale must match the locale used by the runtime system (LC_ALL, LANG).

6. Verify the environment variable defining the search path for DB2 CLI database client shared libraries (libdb2.so on UNIX™, DB2CLI.DLL on Windows®).

### Table 200: Shared library environment setting for DB2® version

<table>
<thead>
<tr>
<th>DB2® version</th>
<th>Shared library environment setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2® 10.x and higher</td>
<td>UNIX™: Add $DB2DIR/lib32 (for 32 bit) or $DB2DIR/lib64 (for 64 bit) to LD_LIBRARY_PATH (or its equivalent). Windows®: Add %DB2DIR%\bin to PATH.</td>
</tr>
</tbody>
</table>

7. To verify if the DB2® client environment is correct, you can, for example, start the db2 command interpreter and connect to the server:

```
$ db2
db2 => CONNECT TO dbname USER username USING password
```

8. Setup the FGLPROFILE entries for database connections.

   a) Define the IBM® DB2® database driver:

   ```
   dbi.database.dbname.driver = "dbmdb2"
   ```

   b) The "source" parameter defines the name of the IBM® DB2® database name.

   ```
   dbi.database.dbname.source = "test1"
   ```

   c) Define the database schema selection if needed:

   Use the following entry to define the database schema to be used by the application. The database interface will automatically perform a SET SCHEMA name instruction to switch to a specific schema:

   ```
   dbi.database.dbname.db2.schema = 'name'
   ```

   Here dbname identifies the database name used in the BDL program (DATABASE dbname) and name is the schema name to be used in the SET SCHEMA instruction. If this entry is not defined, no "SET SCHEMA" instruction is executed and the current schema defaults to the user's name.

9. Setup db2cli.ini to disable deferred prepare option, if PREPARE/DECLARE instructions must raise errors when the SQL statement is invalid:

   ```
   DeferredPrepare = 0
   ```
By default, DB2 CLI uses deferred prepare, which gives best performances but does not raise an SQL error with PREPARE/DECLARE in case of invalid SQL.

**Database concepts**

**IBM® DB2® related database concept topics.**

**Database concepts**

As with Informix®, an IBM® DB2® database server can handle more than one database entity. Informix® servers have an ID (INFORMIXSERVER) and databases are identified by name. IBM® DB2® instances are identified by the DB2INSTANCE environment variable and databases have to be cataloged as data sources (see IBM® DB2® documentation for more details).

**Tip:** If you have several Informix database entities, migrating from the Informix database to another database it is a good opportunity to centralize all tables in a single database. To avoid conflicts with table names, use a prefix when needed.

**Data storage concepts**

When converting from Informix® to IBM® DB2® the aim is to try to preserve as much of the data storage information as possible in the process. The most important storage decisions made for Informix® database objects (like initial sizes and physical placement) can be applied to the IBM® DB2® database.

Storage concepts are quite similar in Informix® and in IBM® DB2®, but the names are different.

These tables compare Informix® storage concepts to IBM® DB2® storage concepts:

**Table 201: Physical units of storage (Informix® vs. DB2®)**

<table>
<thead>
<tr>
<th>Informix®</th>
<th>IBM® DB2®</th>
</tr>
</thead>
<tbody>
<tr>
<td>The largest unit of physical disk space is a &quot;chunk&quot;, which can be</td>
<td>One or more &quot;containers&quot; are created for each &quot;tablespace&quot; to physically</td>
</tr>
<tr>
<td>allocated either as a cooked file (I/O is controlled by the OS) or as</td>
<td>store the data of all logical structures. Like Informix® &quot;chunks&quot;, &quot;</td>
</tr>
<tr>
<td>raw device (=UNIX® partition, I/O is controlled by the database engine).</td>
<td>containers&quot; can be an OS file or a raw device.</td>
</tr>
<tr>
<td>A &quot;dbspace&quot; uses at least one &quot;chunk&quot; for storage.</td>
<td>You can add &quot;containers&quot; to a &quot;tablespace&quot; in order to increase the size</td>
</tr>
<tr>
<td>You must add &quot;chunks&quot; to &quot;dbspaces&quot; in order to increase the size of the</td>
<td>of the logical unit of storage or you can define EXTEND options.</td>
</tr>
<tr>
<td>logical unit of storage.</td>
<td></td>
</tr>
<tr>
<td>A &quot;page&quot; is the smallest physical unit of disk storage that the</td>
<td>At the finest level of granularity, IBM® DB2® stores data in &quot;data</td>
</tr>
<tr>
<td>engine uses to read from and write to databases.</td>
<td>blocks&quot; with size corresponding to a multiple of the operating system's</td>
</tr>
<tr>
<td>A &quot;chunk&quot; contains a certain number of &quot;pages&quot;.</td>
<td>block size.</td>
</tr>
<tr>
<td>The size of a &quot;page&quot; must be equal to the operating system's block size.</td>
<td>You set the &quot;data block&quot; size when creating the database.</td>
</tr>
<tr>
<td>An &quot;extent&quot; consists of a collection of contiguous &quot;pages&quot; that the</td>
<td>An &quot;extent&quot; is a specific number of contiguous &quot;data blocks&quot;, obtained in</td>
</tr>
<tr>
<td>engine uses to allocate both initial and subsequent storage space for</td>
<td>a single allocation.</td>
</tr>
<tr>
<td>database tables.</td>
<td>When creating a table, you can specify the first extent size and the size</td>
</tr>
<tr>
<td>When creating a table, you can specify the first extent size and the</td>
<td>of future extents with the STORAGE() option.</td>
</tr>
<tr>
<td>size of future extents with the EXTENT SIZE and NEXT EXTENT options.</td>
<td>For a single table, &quot; extents&quot; can be located in different &quot;data files&quot;</td>
</tr>
<tr>
<td>For a single table, &quot; extents&quot; can be located in different &quot;chunks&quot; of</td>
<td>of the same &quot;tablespace&quot;.</td>
</tr>
</tbody>
</table>
Table 202: Logical units of storage (Informix® vs. DB2®)

<table>
<thead>
<tr>
<th>Informix®</th>
<th>IBM® DB2®</th>
</tr>
</thead>
<tbody>
<tr>
<td>A &quot;table&quot; is a logical unit of storage that contains rows of data values.</td>
<td>Same concept as Informix®.</td>
</tr>
<tr>
<td>A &quot;database&quot; is a logical unit of storage that contains table and index data. Each database also contains a system catalog that tracks information about database elements like tables, indexes, stored procedures, integrity constraints and user privileges.</td>
<td>Same concept as Informix®, An IBM® DB2® instance can manage several databases.</td>
</tr>
<tr>
<td>Database tables are created in a specific &quot;dbspace&quot;, which defines a logical place to store data. If no dbspace is given when creating the table, Informix® defaults to the current database dbspace.</td>
<td>Database tables are created in a specific &quot;tablespace&quot;, which defines a logical place to store data. The main difference with Informix® &quot;dbspaces&quot; is that IBM® DB2® tablespaces belong to a &quot;database&quot;, while Informix® &quot;dbspaces&quot; are external to a database.</td>
</tr>
</tbody>
</table>

Table 203: Other storage concepts (Informix® vs. DB2®)

<table>
<thead>
<tr>
<th>Informix®</th>
<th>IBM® DB2®</th>
</tr>
</thead>
<tbody>
<tr>
<td>When initializing an Informix® engine, a &quot;root dbspace&quot; is created to store information about all databases, including storage information (chunks used, other dbspaces, etc.). The &quot;physical log&quot; is a set of continuous disk pages where the engine stores &quot;before-images&quot; of data that has been modified during processing. The &quot;logical log&quot; is a set of &quot;logical-log files&quot; used to record logical operations during on-line processing. All transaction information is stored in the logical log files if a database has been created with transaction log. Informix® combines &quot;physical log&quot; and &quot;logical log&quot; information when doing fast recovery. Saved &quot;logical logs&quot; can be used to restore a database from tape.</td>
<td>Each IBM® DB2® database uses a set of &quot;control files&quot; to store internal information. These files are located in a dedicated directory: &quot;.../$DB2INSTANCE/NODEEnn&quot; DB2® uses &quot;database log files&quot; to record SQL transactions.</td>
</tr>
</tbody>
</table>

Concurrency management

Data consistency and concurrency concepts

- **Data Consistency** applies to situations when readers want to access data currently being modified by writers.
- **Concurrent Data Access** applies to situations when several writers are accessing the same data for modification.
- **Locking Granularity** defines the amount of data concerned when a lock is set (for example, row, page, table).

**Informix®**

Informix® uses a locking mechanism to handle data consistency and concurrency. When a process changes database information with **UPDATE**, **INSERT** or **DELETE**, an **exclusive lock** is set on the touched rows. The lock remains active until the end of the transaction. Statements performed outside a transaction are treated as a transaction containing a single operation and therefore release the locks immediately after execution. SELECT statements can set **shared locks**, depending on **isolation level**. In case of locking conflicts (for example, when two processes want to acquire an exclusive lock on the same row for modification, or when a writer is trying to modify data protected by a shared lock), the behavior of a process can be changed by setting the **lock wait mode**.

Control:
• Lock wait mode: `SET LOCK MODE TO ...`
• Isolation level: `SET ISOLATION TO ...`
• Locking granularity: `CREATE TABLE ... LOCK MODE {PAGE | ROW}`
• Explicit exclusive lock: `SELECT ... FOR UPDATE`

Defaults:
• The default isolation level is `READ COMMITTED`.
• The default lock wait mode is `NOT WAIT`.
• The default locking granularity is `PAGE`.

IBM® DB2®
As in Informix®, IBM® DB2® uses locks to manage data consistency and concurrency. The database manager sets _exclusive locks_ on the modified rows and _shared locks_ when data is read, based on the _isolation level_. The locks are held until the end of the transaction. When several processes want to modify the same data, the latest processes must wait until the first finishes its transaction. Readers do not have to wait for writers: Row versioning is supported, to allow data to be committed before the modification in progress. The _lock granularity_ is at the row or table level. For more details, see DB2's Administration Guide, "Application Consideration".

Control:
• Lock wait mode: Always `WAIT`. Only the Lock Timeout can be changed, but this is a global database parameter.
• Isolation level: Can be set through an API function call or with a database client configuration parameter.
• Locking granularity: Row level or Table level.
• Explicit locking: `SELECT ... FOR UPDATE`

Defaults:
• The default isolation level is Cursor Stability (readers cannot see uncommitted data, no shared lock is set when reading data).

Solution
The `SET ISOLATION TO ...` Informix® syntax is replaced by an ODBC API call setting the `SQL_ATTR_TXN_ISOLATION` connection attribute. The next table shows the isolation level mappings applied by the database driver:

Table 204: Isolation level mappings done by the IBM® DB2® LUW database driver

<table>
<thead>
<tr>
<th>SET ISOLATION instruction in program</th>
<th>ODBC SQL_ATTR_TXN_ISOLATION connection attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET ISOLATION TO DIRTY READ</td>
<td><code>SQL_TXN_READ_UNCOMMITTED</code></td>
</tr>
<tr>
<td>SET ISOLATION TO COMMITTED READ [READ COMMITTED] [RETAIN UPDATE LOCKS]</td>
<td><code>SQL_TXN_READ_COMMITTED</code></td>
</tr>
<tr>
<td>SET ISOLATION TO CURSOR STABILITY</td>
<td><code>SQL_TXN_REPEATABLE_READ</code></td>
</tr>
<tr>
<td>SET ISOLATION TO REPEATABLE READ</td>
<td><code>SQL_TXN_SERIALIZABLE</code></td>
</tr>
</tbody>
</table>

For portability, it is recommended that you work with Informix® in the read committed isolation level, to make processes wait for each other (lock mode wait) and to create tables with the "lock mode row" option.

See Informix® and IBM® DB2® documentation for more details about data consistency, concurrency and locking mechanisms.

Related concepts
Concurrent data access on page 599
Understanding concurrent data access and data consistency.

**Optimistic locking** on page 624
Implementing optimistic locking to handle access concurrently to the same database records.

**Cursors WITH HOLD** on page 631
Programming WITH HOLD cursors using SELECT with and without FOR UPDATE clause.

**Transactions handling**

**Informix®**
With the Informix® native mode (non ANSI):

- Transactions blocks start with `BEGIN WORK` and terminate with `COMMIT WORK` or `ROLLBACK WORK`.
- Statements executed outside a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.

```
UPDATE tab1 SET ... -- auto-committed
BEGIN WORK            -- start of TX block
UPDATE tab1 SET ...
UPDATE tab2 SET ...
...                  -- end of TX block
COMMIT WORK
```

Informix® version 11.50 introduces savepoints:

```
SAVEPOINT name [UNIQUE]
ROLLBACK [WORK] TO SAVEPOINT _name_1
RELEASE SAVEPOINT name
```

**IBM® DB2®**
Transactions in IBM® DB2®:

- Beginning of transactions are implicit; two transactions are delimited by COMMIT or ROLLBACK.
- DDL statements can be executed (and canceled) in transactions.

Savepoints in IBM® DB2®:

- Savepoints must be declared with the `ON ROLLBACK RETAIN CURSORS` clause.
- Rollback must always specify the savepoint name.

**Solution**
The Informix® behavior is simulated with an autocommit mode in the IBM® DB2® interface. A switch to the explicit commit mode is done when a BEGIN WORK is performed by the BDL program. Regarding the transaction control instructions, the BDL applications do not have to be modified in order to work with IBM® DB2®.

**Note:** If you want to use savepoints, always specify the savepoint name in `ROLLBACK TO SAVEPOINT`.

See also **SELECT FOR UPDATE**

**Related concepts**
Database transactions on page 600
Database transactions define a set of SQL instructions to be executed as a whole, or rolled back as a whole.

Database users

Informix®

Until version 11.70.xC2, Informix® database users must be created at the operating system level and must be members of the 'informix' group.

Starting with 11.70.xC2, Informix® supports database-only users with the CREATE USER instruction, as in most other db servers.

Any database user must have sufficient privileges to connect and use resources of the database; user rights are defined with the GRANT command.

IBM® DB2®

With DB2, users must be defined as operating system users with a specific DB2® environment.

The database administrator must grant the CONNECT authority to these users.

• Database authorities involve actions on a database as a whole. When a database is created, some authorities are automatically granted to anyone who accesses the database. For example, CONNECT, CREATETAB, BINDADD and IMPLICIT_SCHEMA authorities are granted to all users.

• Database privileges involve actions on specific objects within the database. When a database is created, some privileges are automatically granted to anyone who accesses the database. For example, SELECT privilege is granted on catalog views and EXECUTE and BIND privilege on each successfully bound utility is granted to all users.

Together, privileges and authorities act to control access to an instance and its database objects. Users can access only those objects for which they have the appropriate authorization, that is, the required privilege or authority.

Solution

Set up the IBM® DB2® environment for each user as described in the documentation.

Related concepts

Database users and security on page 608

Properly identifying database users allows to use database security and audit features.

Setting privileges

Informix®

Informix® users must have at least the CONNECT privilege to access the database:

GRANT CONNECT TO username

Application administration users need the RESOURCE privilege to create tables:

GRANT RESOURCE TO username

Since version 7.20, Informix® supports database roles:

GRANT rolename TO username

IBM® DB2®

IBM® DB2® supports the concept of roles to grant or revoke permissions to a group of users.
IBM® DB2® users must have at least the CONNECT authority to access the database:

```sql
GRANT CONNECT ON DATABASE TO (PUBLIC | user | group)
```

See IBM® DB2® documentation for more details.

**Solution**

Informix® and IBM® DB2® user privileges management are quite similar.

See also Temporary Tables

**Data dictionary**

IBM® DB2® related data dictionary topics.

**Data type conversion table: Informix to DB2**

**Table 205: Data type conversion table (Informix to DB2)**

<table>
<thead>
<tr>
<th>Informix® data types</th>
<th>DB2® data types</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR (n)</td>
<td>CHAR (n) (max is 254 chars)</td>
</tr>
<tr>
<td>VARCHAR (n[,m])</td>
<td>VARCHAR (n) (max is 32762 chars)</td>
</tr>
<tr>
<td>LVARCHAR (n)</td>
<td>VARCHAR (n) (max is 32762 chars)</td>
</tr>
<tr>
<td>NCHAR (n)</td>
<td>N/A</td>
</tr>
<tr>
<td>NVARCHAR (n[,m])</td>
<td>N/A</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>BOOLEAN (CHAR(1) with DB2 &lt; 11.1)</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>BIGINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>INT8</td>
<td>BIGINT</td>
</tr>
<tr>
<td>SERIAL[(start)]</td>
<td>INTEGER (see note 1)</td>
</tr>
<tr>
<td>BIGSERIAL[(start)]</td>
<td>BIGINT (see note 1)</td>
</tr>
<tr>
<td>SERIAL8[(start)]</td>
<td>BIGINT (see note 1)</td>
</tr>
<tr>
<td>DOUBLE PRECISION / FLOAT[(n)]</td>
<td>FLOAT[(n)] / DOUBLE</td>
</tr>
<tr>
<td>REAL / SMALLFLOAT</td>
<td>REAL</td>
</tr>
<tr>
<td>DECIMAL(p,s)</td>
<td>DECIMAL (p, s) (max is 31 digits)</td>
</tr>
<tr>
<td>DECIMAL(p) with p&lt;=15</td>
<td>DECIMAL (p, s) (max is 31 digits)</td>
</tr>
<tr>
<td>DECIMAL(p) with p&gt;15</td>
<td>DECIMAL (p, s) (max is 31 digits)</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL (p, s) (max is 31 digits)</td>
</tr>
<tr>
<td>MONEY(p,s)</td>
<td>DECIMAL (p, s) (max is 31 digits)</td>
</tr>
<tr>
<td>MONEY(p)</td>
<td>DECIMAL (p, s) (max is 31 digits)</td>
</tr>
<tr>
<td>MONEY</td>
<td>DECIMAL (16, 2)</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME HOUR TO MINUTE</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME HOUR TO SECOND</td>
<td>TIME</td>
</tr>
</tbody>
</table>
### Informix® data types vs. DB2® data types

<table>
<thead>
<tr>
<th>Informix® data types</th>
<th>DB2® data types</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATETIME HOUR TO FRACTION (n)</td>
<td>TIMESTAMP (for fraction storage)</td>
</tr>
<tr>
<td>DATETIME YEAR TO MONTH</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>DATETIME YEAR TO DAY</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>DATETIME YEAR TO HOUR</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>DATETIME YEAR TO MINUTE</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>DATETIME YEAR TO SECOND</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>DATETIME YEAR TO FRACTION (n)</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>INTERVAL q1 TO q2</td>
<td>CHAR (50)</td>
</tr>
<tr>
<td>TEXT</td>
<td>CLOB (of 500K)</td>
</tr>
<tr>
<td>BYTE</td>
<td>BLOB (of 500K)</td>
</tr>
</tbody>
</table>

Notes:

1. For more details about serial emulation, see SERIAL and BIGSERIAL data types on page 774.

### BOOLEAN data type

**Informix®**

Informix® supports the BOOLEAN data type, which can store 't' or 'f' values.

Genero BDL implements the BOOLEAN data type in a different way: A BOOLEAN variable stores integer values 1 or 0 (for TRUE or FALSE). This type is designed to hold the result of a boolean expression.

**IBM® DB2®**

Starting with version 11.1.1.1, IBM® DB2® supports the BOOLEAN SQL type.

Possible values are the TRUE/FALSE constants, '1'/0' string literals and 1/0 numbers.

**Solution**

The database interface converts the BOOLEAN SQL type to CHAR (1) and stores '1' or '0' values in the column.

With versions before DB2® 11.1, the database interface converts the BOOLEAN SQL type to CHAR (1) and stores '1' or '0' values in the column.

Starting with DB2® 11.1, the database uses BOOLEAN keyword as is and the TRUE/FALSE keywords can be used.

The BOOLEAN type translation can be controlled with the following FGLPROFILE entry:

```plaintext
dbi.database.dsname.ifxemul.datatype.boolean = [ true | false ]
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

- Using portable data types on page 614

Only a limited set of data types are really portable across several database engines.

### CHAR and VARCHAR data types

**Informix®**

Informix® supports the following character data types:
Table 206: Informix® character data types

<table>
<thead>
<tr>
<th>Informix® data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR (n)</td>
<td>SBCS and MBCS character data (max is 32767 bytes)</td>
</tr>
<tr>
<td>VARCHAR (n, m]</td>
<td>SBCS and MBCS character data (max is 255 bytes)</td>
</tr>
<tr>
<td>NCHAR (n)</td>
<td>Same as CHAR, with specific collation order</td>
</tr>
<tr>
<td>NVARCHAR (n, m]</td>
<td>Same as VARCHAR, with specific collation order</td>
</tr>
<tr>
<td>LVARCHAR (n)</td>
<td>max size varies depending on the IDS version</td>
</tr>
</tbody>
</table>

With Informix®, both CHAR/VARCHAR and NCHAR/NVARCHAR data types can be used to store single-byte or multibyte encoded character strings. The only difference between CHAR/VARCHAR and NCHAR/NVARCHAR is in how they use sorting: N[VAR]CHAR types use the collation order, while [VAR]CHAR types use the byte order.

The character set used to store strings in CHAR/VARCHAR/NCHAR/NVARCHAR columns is defined by the DB_LOCALE environment variable.

The character set used by applications is defined by the CLIENT_LOCALE environment variable.

Informix® uses Byte Length Semantics (the size N that you specify in [VAR]CHAR(N) is expressed in bytes, not characters as in some other databases).

IBM® DB2®

IBM® DB2® supports following data types to store character data:

Table 207: IBM® DB2® character data types

<table>
<thead>
<tr>
<th>IBM® DB2® data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR (n)</td>
<td>SBCS character data using the database character set, where n is specified in bytes (max is 255 bytes)</td>
</tr>
<tr>
<td>VARCHAR (n)</td>
<td>SBCS character data using the database character set, where n is specified in bytes (max is 32672 bytes)</td>
</tr>
<tr>
<td>GRAPHIC (n)</td>
<td>UNICODE/UCS-2 character data, where n is specified in characters (max is 127 characters)</td>
</tr>
<tr>
<td>VARGRAPHIC (n)</td>
<td>UNICODE/UCS-2 character data, where n is specified in characters (max is 16336 characters)</td>
</tr>
</tbody>
</table>

Like Informix®, IBM® DB2® uses Byte Length Semantics to define the length of CHAR/VARCHAR columns. However, GRAPHIC and VARGRAPHIC lengths are specified in characters (i.e. max number of double-byte characters).

The character set used by DB2® to store CHAR and VARCHAR data is defined in the database locale section when creating a new database. If your application uses UTF-8, consider creating the DB2® database with the UTF-8 codeset.

DB2® can automatically convert from/to the client and server characters sets. In the client applications, you define the character set with the DB2CODEPAGE profile variable.

**Solution**

Informix® CHAR (N) types must be mapped to DB2® CHAR (N) types, and Informix® VARCHAR (N) or LVARCHAR (N) columns must be mapped to DB2® VARCHAR (N).

**Important:**
• DB2® does not support NCHAR/NVARCHAR types. If your programs create tables with these types, you must review your code. The DB2® driver does not automatically convert the NCHAR/NVARCHAR Informix® types to GRAPHIC/VARGRAPHIC, because the meaning of the length is different.
• Check that your database schema does not use CHAR or VARCHAR types with a length exceeding the DB2® limits. Especially, the Informix® CHAR type has a very long size limit compared to DB2® CHAR.

When using a multibyte character set (such as UTF-8), if the DB2® database was created with the appropriate codeset (UTF-8), you can use the CHAR/VARCHAR columns, and user byte length semantics in programs. If the database code set is non multibyte, you must use the GRAPHIC and VARGRAPHIC data types to store multibyte character data, and use character length semantics in BDL programs with FGL_LENGTH_SEMANTICS=CHAR.

When extracting a database schema from a DB2® database, the fgldbsch schema extractor uses the size of the column in characters, not the octet length. If you have created a CHAR(10 (characters) ) column a in DB2® database using the UTF-8 character set, the .sch file will get a size of 10, that will be interpreted by FGL_LENGTH_SEMANTICS as either a number of bytes or characters.

Do not forget to properly define the database client character set, which must correspond to the runtime system character set.

See also the section about Localization.

The CHAR/VARCHAR type translation can be controlled with the following FGLPROFILE entries:

```
dbi.database.dsnname.ifxemul.datatype.char = | true | false |
dbi.database.dsnname.ifxemul.datatype.varchar = | true | false |
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

CHAR and VARCHAR types on page 616
Using the CHAR and VARCHAR data types with different databases.

**Numeric data types**

**Informix®**

Informix® supports several data types to store numbers:

**Table 208: Informix® numeric data types**

<table>
<thead>
<tr>
<th>Informix® data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>16 bit signed integer</td>
</tr>
<tr>
<td>INTEGER</td>
<td>32 bit signed integer</td>
</tr>
<tr>
<td>BIGINT</td>
<td>64 bit signed integer</td>
</tr>
<tr>
<td>INT8</td>
<td>64 bit signed integer (replaced by BIGINT)</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>Equivalent to DECIMAL(16)</td>
</tr>
<tr>
<td>DECIMAL(p)</td>
<td>Floating-point decimal number (max precision is 32)</td>
</tr>
<tr>
<td>DECIMAL(p, s)</td>
<td>Fixed-point decimal number (max precision is 32)</td>
</tr>
<tr>
<td>MONEY</td>
<td>Equivalent to DECIMAL(16,2)</td>
</tr>
<tr>
<td>MONEY(p)</td>
<td>Equivalent to DECIMAL(p, 2) (max precision is 32)</td>
</tr>
<tr>
<td>MONEY(p, s)</td>
<td>Equivalent to DECIMAL(p, s) (max precision is 32)</td>
</tr>
<tr>
<td>REAL / SMALLFLOAT</td>
<td>32-bit floating point decimal (C float)</td>
</tr>
</tbody>
</table>
IBM® DB2® supports the following numeric data types:

**Note:** DB2® V 9.1 introduces the DECFLOAT(16) and DECFLOAT(34) floating point decimal types to store large decimals. The next table lists the Informix® types and DB2® equivalents.

**Table 209: IBM® DB2® numeric data types**

<table>
<thead>
<tr>
<th>IBM® DB2® data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>16 bit signed integer</td>
</tr>
<tr>
<td>INTEGER</td>
<td>32 bit signed integer</td>
</tr>
<tr>
<td>BIGINT</td>
<td>64 bit signed integer</td>
</tr>
<tr>
<td>DECFLOAT(16)</td>
<td>16 bit signed floating point decimal</td>
</tr>
<tr>
<td>DECFLOAT(34)</td>
<td>32 bit signed floating point decimal</td>
</tr>
<tr>
<td>DECIMAL(p, s)</td>
<td>Fixed point decimal</td>
</tr>
<tr>
<td>REAL</td>
<td>32-bit floating point decimal (C float)</td>
</tr>
<tr>
<td>FLOAT[(n)] (DOUBLE)</td>
<td>64-bit floating point decimal (C double)</td>
</tr>
</tbody>
</table>

**Solution**

Use the following conversion rules to map Informix numeric types to DB2 numeric types:

**Table 210: Informix® numeric data types and DB2® equivalents**

<table>
<thead>
<tr>
<th>Informix® data type</th>
<th>IBM® DB2® equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>INT8 / BIGINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>DECIMAL(p&lt;=16)</td>
<td>DECFLOAT(16)</td>
</tr>
<tr>
<td>DECIMAL(p&gt;16)</td>
<td>DECFLOAT(34)</td>
</tr>
<tr>
<td>DECIMAL(p&lt;=31, s)</td>
<td>DECIMAL(p, s)</td>
</tr>
<tr>
<td>DECIMAL(32, s)</td>
<td>No equivalent</td>
</tr>
<tr>
<td>MONEY</td>
<td>DECIMAL(16,2)</td>
</tr>
<tr>
<td>MONEY(p)</td>
<td>DECIMAL(p,2)</td>
</tr>
<tr>
<td>MONEY(p,s)</td>
<td>DECIMAL(p,s)</td>
</tr>
<tr>
<td>SMALLFLOAT</td>
<td>REAL</td>
</tr>
<tr>
<td>FLOAT[(n)]</td>
<td>FLOAT[(n)] (DOUBLE)</td>
</tr>
</tbody>
</table>

SQL scripts to create databases must be converted manually. Tables created from BDL programs do not have to be converted; the database interface detects the MONEY data type and uses the DECIMAL type for DB2®.
**Important:** The maximum precision for DB2® decimals is 31 digits, while Informix® supports 32 digits.

The DECIMAL (p) data type is converted to DECFLOAT (16) (for DECIMAL (p<=16)) or DECFLOAT (34) (for DECIMAL (p>16)) to store floating point decimals. If you create tables with DECFLOAT columns, you will lose the original DECIMAL precision when extracting the schema with fgl dbsch, because IBM® DB2® supports only two precision specifications (16 or 34). Note also the DECFLOAT (34) will be extracted as DECIMAL (32), since the Genero DECIMAL type has a maximum precision of 32 digits.

The numeric types translation can be controlled with the following FGLPROFILE entries:

```
<table>
<thead>
<tr>
<th>FGLPROFILE entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbi.database.dsname.ifxemul.datatype.smallint = {true</td>
</tr>
<tr>
<td>dbi.database.dsname.ifxemul.datatype.integer = {true</td>
</tr>
<tr>
<td>dbi.database.dsname.ifxemul.datatype.bigint = {true</td>
</tr>
<tr>
<td>dbi.database.dsname.ifxemul.datatype.int8 = {true</td>
</tr>
<tr>
<td>dbi.database.dsname.ifxemul.datatype.decimal = {true</td>
</tr>
<tr>
<td>dbi.database.dsname.ifxemul.datatype.money = {true</td>
</tr>
<tr>
<td>dbi.database.dsname.ifxemul.datatype.float = {true</td>
</tr>
<tr>
<td>dbi.database.dsname.ifxemul.datatype.smallfloat = {true</td>
</tr>
</tbody>
</table>
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

**Using portable data types** on page 614

Only a limited set of data types are really portable across several database engines.

**DATE and DATETIME data types**

**Informix®**

Informix® provides two data types to store date and time information:

- **DATE** = for year, month and day storage.
- **DATETIME** = for year to fraction (1-5) storage.

The DATE type is stored as an INTEGER with the number of days since 1899/12/31.

The DATETIME type can be defined with various time units, by specifying a start and end qualifier. For example, you can define a datetime to store an hour-to-second time value with DATETIME HOUR TO SECOND.

The values of Informix® DATETIME can be represented with a character string literal, or as DATETIME () literals:

```
'2017-12-24 15:45:12.345'  -- a DATETIME YEAR TO FRACTION(3)
'15:45'   -- a DATETIME HOUR TO MINUTE
DATETIME(2017-12-24 12:45) YEAR TO MINUTE
DATETIME(12:45:56.333) HOUR TO FRACTION(3)
```

Informix® is able to convert quoted strings to DATE/DATETIME data, if the string contains matching environment parameters. The string to date conversion rules for DATE is defined by the DBDATE environment variable. The string to datetime format for DATETIME is defined by the GL_DATETIME environment variable.

**Note:** Within Genero programs, the string representation for DATETIME values is always ISO (YYYY-MM-DD hh:mm:ss.ffffff)

Informix® supports date arithmetic on DATE and DATETIME values. The result of an arithmetic expression involving dates/times is an INTEGER number of days when only DATE values are used, and an INTERVAL value if a DATETIME is used in the expression.

Informix® automatically converts an INTEGER to a DATE when the integer is used to set a value of a date column.

Informix® provides the CURRENT ↓ q1 TO q2 ↓ operator, to get the system date/time on the server where the current database is located. When no qualifiers are specified, CURRENT returns a DATETIME YEAR TO FRACTION (3). Informix also supports the SYSDATE operator, which returns the current system time as a DATETIME YEAR TO FRACTION (5).
**Note:** The `USEOSTIME` configuration parameter must be set to 1 in order to get the subsecond precision in `CURRENT` and `SYSDATE` operators. See Informix documentation for more details.

**IBM® DB2®**

IBM® DB2® provides the following data type to store date and time data:

**Table 211: IBM® DB2® date/time data types**

<table>
<thead>
<tr>
<th>IBM® DB2® data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>for year, month, day storage</td>
</tr>
<tr>
<td>TIME</td>
<td>for hour, minutes, seconds storage</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>for year, month, day, hour, minutes, seconds, fraction(12) storage</td>
</tr>
</tbody>
</table>

Like Informix®, IBM® DB2® can convert quoted strings to dates, times or timestamps. Only one format is possible: 'yyyy-mm-dd' for dates, 'hh:mm:ss' for times and 'yyyy-mm-dd hh:mm:ss:f' for timestamps.

In IBM® DB2®, the result of a date/time arithmetic expression involving `DATE` values is a number of days with a decimal part representing the fraction of the day (0.5 = 12H00, 2.00694444 = (2 + (10/1440)) = 2 days and 10 minutes). The result of a date/time arithmetic expression involving `TIME` or `TIMESTAMP` values is a number of seconds with a decimal part representing the fraction of seconds.

**Solution**

Use the following conversion rules to map Informix® date/time types to IBM® DB2® date/time types:

**Table 212: Informix® data types and IBM® DB2® equivalents**

<table>
<thead>
<tr>
<th>Informix® data type</th>
<th>IBM® DB2® data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME HOUR TO MINUTE</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME HOUR TO SECOND</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME HOUR TO FRACTION(n)</td>
<td>TIMESTAMP (for fraction storage)</td>
</tr>
<tr>
<td>DATETIME YEAR TO MONTH</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>DATETIME YEAR TO DAY</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>DATETIME YEAR TO HOUR</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>DATETIME YEAR TO MINUTE</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>DATETIME YEAR TO SECOND</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>DATETIME YEAR TO FRACTION(n)</td>
<td>TIMESTAMP</td>
</tr>
</tbody>
</table>

The `DATE` and `DATETIME` types translation can be controlled with the following FGLPROFILE entries:

```sql
 dbi.database.dsname.ifxemul.datatype.date = ↓ true ↓ false ↓
 dbi.database.dsname.ifxemul.datatype.datetime = ↓ true ↓ false ↓
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

DB2® and Informix® `DATE` data type are equivalent and store year, month, day values.

DB2® `TIME` data type can be used to store Informix® `DATETIME HOUR TO SECOND` and `DATETIME HOUR TO MINUTE` values, and any other `DATETIME` type with qualifiers `HOUR`, `MINUTE`, `SECOND`, except `FRACTION(n)`.,
because the DB2 TIME native type does not store fraction of seconds. Missing time parts default to 00:00:00. For example, when using a DATETIME MINUTE TO SECOND with the value of "45:23", the DB2® TIME value will be "00:45:23".

Informix® DATETIME values with any qualifiers from YEAR to FRACTION(5) can be stored in DB2® TIMESTAMP columns. Missing date or time parts default to 1900-01-01 00:00:00.0. For example, when using a DATETIME DAY TO MINUTE with the value of "23 11:45", the DB2® TIMESTAMP value will be "1900-01-23 11:45:00.0".

Important:
- Using integers as a number of days in an expression with dates is not supported by IBM® DB2®. Check your code to detect where you are using integers with DATE columns.
- Literal DATETIME and INTERVAL expressions (i.e. DATETIME (1999-10-12) YEAR TO DAY) are not converted.
- It is strongly recommended that you use BDL variables in dynamic SQL statements instead of quoted strings representing DATE values. For example:

```sql
LET stmt = "SELECT ... FROM customer WHERE creat_date >'", adate,"'
```

is not portable, use a question mark place holder instead and OPEN the cursor USING adate:

```sql
LET stmt = "SELECT ... FROM customer WHERE creat_date > ?"
```

- DATE arithmetic expressions using SQL parameters (USING variables) are not fully supported. The next example generates an error at PREPARE time:

```sql
SELECT ... WHERE datecol < ? +1
```

- SQL Statements using expressions with TODAY / CURRENT / EXTEND must be reviewed and adapted to the native syntax.

Date/time SQL functions

Table 213: Informix® and IBM® DB2® date/time SQL functions

<table>
<thead>
<tr>
<th>Informix®</th>
<th>IBM® DB2®</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>current date</td>
</tr>
<tr>
<td>current hour to second</td>
<td>current time</td>
</tr>
<tr>
<td>current year to fraction(5)</td>
<td>current timestamp</td>
</tr>
</tbody>
</table>

Related concepts
Date/time literals in SQL statements on page 635
Good practices for date and time handling in SQL.

INTERVAL data type

Informix®

Informix® provides the INTERVAL data type to store a value that represents a span of time. INTERVAL types are divided into two classes:

- **year-month** intervals. For example: INTERVAL YEAR(5) TO MONTH
- **day-time** intervals. For example: INTERVAL DAY(9) TO SECOND

INTERVAL columns can be defined with various time units, by specifying a start and end qualifier. For example, you can define an interval to store a number of hours and minutes with INTERVAL HOUR(n) TO MINUTE, where n defines the maximum number of digits for the hours unit.
The values of Informix® INTERVAL can be represented with a character string literal, or as INTERVAL() literals:

```
'-9834 15:45:12.345' -- an INTERVAL DAY(6) TO FRACTION(3)
'7623-11' -- an INTERVAL YEAR(9) TO MONTH
INTERVAL(18734:45) HOUR(5) TO MINUTE
INTERVAL(-7634-11) YEAR(5) TO MONTH
```

**IBM® DB2®**

IBM® DB2® does not provide a data type corresponding the Informix® INTERVAL data type.

**Solution**

The INTERVAL data type and values are converted CHAR (50) column with IBM® DB2®.

INTERVAL values can be stored and retrieved from the database. However, since IBM® DB2® does not support a native interval type, arithmetics cannot be performed on the database side in SQL statements.

The INTERVAL types translation can be controlled with the following FGLPROFILE entry:

```

dbi.database.dsnname.ifxemul.datatype.interval = \t\true\t\false
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

*Using portable data types* on page 614

Only a limited set of data types are really portable across several database engines.

**SERIAL and BIGSERIAL data types**

**Informix®**

Informix® supports the SERIAL, BIGSERIAL data types to produce automatic integer sequences:

- SERIAL can produce 32 bit integers (INTEGER)
- BIGSERIAL can produce 64 bit integers (BIGINT)
- SERIAL8 is a synonym for BIGSERIAL

Steps to use serials with Informix®:

1. Create the table with a column using SERIAL, or BIGSERIAL.
2. To generate a new serial, no value or a zero value is specified in the INSERT statement:

   ```
   INSERT INTO tab1 ( c ) VALUES ( 'aa' )
   INSERT INTO tab1 ( k, c ) VALUES ( 0, 'aa' )
   ```

3. After INSERT, the new value of a SERIAL column is provided in SQLCA.SQLERRD[2], while the new value of a BIGSERIAL value must be fetched with a `SELECT dbinfo('bigserial')` query.

Informix® allows you to insert rows with a value different from zero for a serial column. Using an explicit value will automatically increment the internal serial counter, to avoid conflicts with future INSERT statements that are using a zero value:

```
CREATE TABLE tab ( k SERIAL); -- internal counter = 0
INSERT INTO tab VALUES ( 0 ); -- internal counter = 1
INSERT INTO tab VALUES ( 10 ); -- internal counter = 10
INSERT INTO tab VALUES ( 0 ); -- internal counter = 11
DELETE FROM tab; -- internal counter = 11
INSERT INTO tab VALUES ( 0 ); -- internal counter = 12
```
**IBM® DB2®**

IBM® DB2® supports **IDENTITY** columns:

```sql
CREATE TABLE tab ( k INTEGER GENERATED ALWAYS AS IDENTITY, name VARCHAR(50) )
```

To get the last generated **IDENTITY** value after an **INSERT**, DB2® provides the **IDENTITY_VAL_LOCAL()** function.

**IBM® DB2®** supports **SEQUENCES**:

```sql
CREATE SEQUENCE sq1 START WITH 100
```

To create a new sequence number, you must use the **NEXTVAL FOR** operator:

```sql
INSERT INTO table VALUES ( NEXTVAL FOR sq1, ... )
```

To get the last generated sequence number, you must use the **PREVVAL FOR** operator:

```sql
SELECT PREVVAL FOR sq1 ...
```

**Solution**

**Note**: For best SQL portability when using different types of databases, consider using sequences as described in **Solution 3: Use native SEQUENCE database objects** on page 628.

To emulate Informix® serials with IBM® DB2®, you can use **IDENTITY** columns (1), or insert triggers using sequences (2). The first solution is faster, but does not allow explicit serial value specification in insert statements; the second solution is slower but allows explicit serial value specification.

**Important**: The trigger-based solution is provided to simplify the conversion from Informix, but is slower as the solution using identity columns. We strongly recommend that you use native **IDENTITY** columns instead to get best performances.

The method used to emulate **SERIAL** types is defined by the **ifxemul.datatype.serial.emulation** FGLPROFILE parameter:

```sql
dbi.database.dbname.ifxemul.datatype.serial.emulation = {"native","trigseq"}
```

- native: uses **IDENTITY** columns.
- trigseq: uses insert triggers with sequences.

The default emulation technique is "native".

The serial types emulation can be enabled or disabled with the following FGLPROFILE entries:

```sql
dbi.database.dbname.ifxemul.datatype.serial = {true,false}
dbi.database.dbname.ifxemul.datatype.serial8 = {true,false}
dbi.database.dbname.ifxemul.datatype.bigserial = {true,false}
```

**Disabling automatic serial retrieval for SQLCA.SQLERRD[2]**

**SERIAL** emulation can be totally disabled by setting the **ifxemul.datatype.serial** FGLPROFILE entry to false:

```sql
dbi.database.dbname.ifxemul.datatype.serial = false
```

For Informix® compatibility, after an **INSERT** statement, the ODI drivers automatically execute another SQL query (or do a DB client API call when possible), to get the last generated serial, and fill the **SQLCA.SQLERRD[2]**
When SERIAL emulation is required (to create temp tables with a serial column during program execution), and the SQLCA.SQLERRD[2] register does not need to be filled, (typically because you use your own method to retrieve the last generated serial), you can set the ifxemul.datatype.serial.sqlerrd2 FGLPROFILE entry to false. This will avoid the automatic retrieval of last serial value to fill SQLCA.SQLERRD[2]:

\[\text{dbi.database.dbname.ifxemul.datatype.serial.sqlerrd2 = false}\]

See also \texttt{db_get_last_serial()} on page 2308.

Using the native serial emulation

In database creation scripts, all SERIAL\[(n)\] data types must be converted by hand to:

\[\text{INTEGER GENERATED ALWAYS AS IDENTITY[\(\text{START WITH n, INCREMENT BY 1}\)]}\]

while the SERIAL8 and BIGSERIAL\[(n)\] types must be converted to:

\[\text{BIGINT GENERATED ALWAYS AS IDENTITY[\(\text{START WITH n, INCREMENT BY 1}\)]}\]

Tables created from the BDL programs can use the SERIAL data type: When a BDL program executes a CREATE [TEMP] TABLE with a SERIAL column, the database interface automatically converts the SERIAL\[(n)\] data type to an IDENTITY specification.

In BDL, the new generated SERIAL value is available from the SQLCA.SQLERRD[2] variable. The database interface performs a call to the IDENTITY**VAL**_LOCAL() function. However, SQLCA.SQLERRD[2] is defined as an INTEGER, it cannot hold values from BIGINT identity columns. If you are using BIGINT IDENTITY columns, you must use the IDENTITY**VAL**_LOCAL() function.

Since IBM® DB2® does not allow you to specify the value of IDENTITY columns, it is mandatory to convert all INSERT statements to remove the SERIAL column from the list. For example, the following statement:

\[\text{INSERT INTO tab (col1, col2) VALUES (0, p_value)}\]

must be converted to:

\[\text{INSERT INTO tab (col2) VALUES (p_value)}\]

Static SQL INSERT using records defined from the schema file must also be reviewed:

\[\text{DEFINE rec LIKE tab.}\ast\text{ INSERT INTO tab VALUES (rec.}\ast\text{) -- will use the serial column}\]

must be converted to:

\[\text{INSERT INTO tab VALUES rec.}\ast\text{ -- without parentheses, serial column is removed}\]

Using the trigseq serial emulation

In database creation scripts, all SERIAL\[(n)\] data types must be converted to INTEGER data types, SERIAL8/ BIGSERIAL must be converted to BIGINT, and you must create a sequence and a trigger for each table using a SERIAL.

To know how to write those triggers, you can create a small Genero program that creates a table with a SERIAL column. Set the FGLSQLDEBUG environment variable and run the program. The debug output will show you the native SQL commands to create the sequence and the trigger.
Tables created from the BDL programs can use the SERIAL data type: When a BDL program executes a CREATE [TEMP] TABLE with a SERIAL column, the database interface automatically converts the SERIAL[(n)] data type to INTEGER and creates the sequence and the insert trigger.

Note:
IBM® DB2® performs NOT NULL data controls before the execution of triggers. If the serial column must be NOT NULL (for example, because it is part of the primary key), you cannot specify a NULL value for that column in INSERT statements.

For example, the following statement:

```
INSERT INTO tab VALUES (NULL,p_value)
```

must be converted to:

```
INSERT INTO tab (col2) VALUES (p_value)
```

Important:
• IBM® DB2® triggers are not automatically dropped when the corresponding table is dropped. They become inoperative instead. Database administrators must take care of this behavior when managing schemas.
• With IBM® DB2®, INSERT statements using NULL for the SERIAL column will produce a new serial value:

```
INSERT INTO tab ( col_serial, col_data ) VALUES ( NULL, 'data' )
```

This behavior is mandatory in order to support INSERT statements which do not use the serial column:

```
INSERT INTO tab (col_data) VALUES ('data')
```

Check if your application uses tables with a SERIAL column that can contain a NULL value.
• With DB2®, trigger creation is not allowed on temporary tables. Therefore, the "trigseq" method cannot work with temporary tables using serials.

Related concepts
Auto-incremented columns (serials) on page 626
How to implement automatic record keys.

ROWID columns

Informix®
When creating a table, Informix® automatically adds a ROWID integer column (applies to non-fragmented tables only).

The ROWID column is auto-filled with a unique number and can be used like a primary key to access a given row.

Note: Informix® ROWID usage was a common practice in the early days of Informix® 4GL programming. Today it is recommended to define all your database tables with a PRIMARY KEY to uniquely identify rows.

With Informix®, the SQLCA.SQLERRD[6] register contains the ROWID of the last row affected by an INSERT, UPDATE or DELETE statement.

IBM® DB2®
IBM® DB2® ROWID columns were introduced in version 9.7. Unlike Informix® integer row ids, DB2® row ids are based on VARCHAR(16) FOR BIT DATA (128 bit integer) that are usually represented as a 32 char hexadecimal representation of the value. The IBM® DB2® ROWID is actually an alternative syntax for RID_BIT(), and a qualified reference to ROWID like tablename.ROWID is equivalent to RID_BIT(tablename).
For example:

```
x'070000000000000000000065CE770000'
```

In DB2® SQL, to find a row with a rowid, you must specify the rowid value as an hexadecimal value:

```
SELECT * FROM customer WHERE ROWID = x'070000000000000000000065CE770000'
```

or convert the ROWID to an hexadecimal representation and then you can compare to a simple string:

```
SELECT * FROM customer WHERE HEX(ROWID) = '070000000000000000000065CE770000'
```

**Solution**

If your Genero BDL application uses rowid columns, review the program logic to use primary keys instead. If the database table does not define a primary key, it should be added. All references to SQLCA.SQLERRD[6] must be removed, because this variable will not hold the ROWID of the last modified row.

The DB2® database driver will convert the ROWID keyword to HEX(ROWID), so it can be used as a VARCHAR(32) with the hexadecimal representation of the BIT DATA. You need however to replace all INTEGER variable definitions by VARCHAR(32) or CHAR(32).

To emulate Informix® integer ROWIDs, you can also use the DB2® GENERATE_UNIQUE built-in function, or the IDENTITY attribute of the INTEGER or BIGINT data types.

For databases where the keyword of the rowid pseudo-column is different than "ROWID", the translation can be controlled with the following FGLPROFILE entry:

```
dbi.database.dsnname.ifxemul.rowid = {true|false}
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

Using ROWID columns on page 644

Automatic ROWID columns is not a common database feature.

**TEXT and BYTE (LOB) types**

**Informix®**

Informix® provides the TEXT, BYTE, CLOB and BLOB data types to store very large texts or binary data.

Legacy Informix® 4GL applications typically use the TEXT and BYTE types.

Genero BDL does not support the Informix® CLOB and BLOB types.

**IBM® DB2®**

IBM® DB2® supports the LONG VARCHAR/CLOB and BLOB/VARGRAPHIC/DBCLOB types for large objects storage

**Solution**

The DB2® database interface can convert BDL TEXT data to DB2® CLOB and BYTE data to DB2® BLOB.

Note that DB2® CLOB and BLOB columns are created with a size of 500K, while Genero TEXT/BYTE program variables have a limit of 2 gigabytes. Make sure that the large object data does not exceed this limit.

The TEXT and BYTE types translation can be controlled with the following FGLPROFILE entries:

```
dbi.database.dsnname.ifxemul.text = {true|false}
```
For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

**Using portable data types** on page 614

Only a limited set of data types are really portable across several database engines.

**Table constraints**

**Informix®**

Informix® supports primary key, unique, foreign key, default and check constraints.

The constraint naming syntax is different in Informix® and most other databases: Informix expects the constraint name after the constraint definition:

```sql
CREATE TABLE emp (  
...  
   emp_code CHAR(10) UNIQUE CONSTRAINT pk_emp,  
...  
)
```

While other SQL database brands require to specify the constraint name before the constraint definition:

```sql
CREATE TABLE emp (  
...  
   emp_code CHAR(10) CONSTRAINT pk_emp UNIQUE,  
...  
)
```

**IBM® DB2®**

IBM® DB2® supports primary key, unique, foreign key, default and check constraints.

**Constraint naming**

The constraint naming clause must be placed before the constraint specification.

**Primary keys**

Like Informix®, DB2® creates an index to enforce PRIMARY KEY constraints (some RDBMS do not create indexes for constraints). Using CREATE UNIQUE INDEX to define unique constraints is obsolete (use primary keys or a secondary key instead).

**Note:** DB2® primary key constraints do not allow NULLs; make sure your tables do not contain NULLs in the primary key columns.

**Unique constraints**

Like Informix®, DB2® creates an index to enforce UNIQUE constraints (some RDBMS do not create indexes for constraints).

**Note:** DB2® unique constraints do not allow NULLs; make sure your tables do not contain NULLs in the unique columns.

**Foreign keys**

Both Informix® and DB2® support the ON DELETE CASCADE option.
Check constraints

The check condition may be any valid expression that can be evaluated to TRUE or FALSE, including functions and literals. You must verify that the expression is not Informix-specific.

Null constraints

Informix® and DB2® support NOT NULL constraints, but Informix® does not allow you to give a name to NOT NULL constraints.

Solution

The database interface does not convert constraint naming expressions when creating tables from BDL programs. Review the database creation scripts to adapt the constraint naming clauses for DB2®.

Related concepts

Data definition statements on page 613

It is recommended to avoid use of DDL in programs.

Name resolution of SQL objects

Informix®

Informix® uses the following form to identify an SQL object:

database[@dbservername]:[{owner|"owner"}.]identifier

The ANSI convention is to use double quotes for identifier delimiters (For example: "customer"."cust_name").

Informix® database object names are not case-sensitive in non-ANSI databases. When using double-quoted identifiers, Informix® becomes case sensitive.

With non-ANSI Informix® databases, you do not have to give a schema name before the tables when executing an SQL statement:

```
SELECT ... FROM customer WHERE ...
```

In Informix® ANSI compliant databases:

- The table name must include "owner", unless the connected user is the owner of the database object.
- The database server shifts the owner name to uppercase letters before the statement executes, unless the owner name is enclosed in double quotes.

IBM® DB2®

IBM® DB2® database object names are case-sensitive. When a name is used without double quotes, it is automatically converted to uppercase letters. When using double quotes, the names are not converted:

```
CREATE TABLE tab1 ( Key INT, Col1 CHAR(20) )
-- Table name is "TAB1", column names are "KEY" and "COL1"
CREATE TABLE "Tab1" ( "Key" INT, "Col1" CHAR(20) )
-- Table name is "Tab1", column names are "Key" and "Col1"
```

In an IBM® DB2® database, tables always belong to a database schema. When executing a SQL statement, a schema name must be used as the high-order part of a two-part object name, unless the current schema corresponds to the table's schema.

The default (implicit) schema is the current user's name but it can be changed with the SET SCHEMA instruction.
Example: The table "TAB1" belongs to the schema "SCH1". User "MARK" (implicit schema is "MARK") wants to access "TAB1" in a SELECT statement:

```sql
SELECT ... FROM TAB1 WHERE ...
-- Error "MARK".TAB1 is an undefined name. SQLSTATE=42704
SELECT ... FROM SCH1.TAB1 WHERE ...
-- OK.
SET SCHEMA SCH1
-- Changes the current schema to SCH1.
SELECT ... FROM TAB1 WHERE ...
-- OK.
```

**Note:** When executing the `SET SCHEMA` instruction, the database interface does not use double quotes around the schema name (= name is converted to uppercase letters). Make sure that the schema name is created with uppercase letters in the database.

DB2® provides *aliases*, but they cannot be used to make a database object name public because aliases belong to schemas also.

**Solution**

To write portable SQL, regarding database object names:

1. Use simple database object names (without any owner/schema prefix)
2. Do not use double quotes to surround database object identifiers.
3. If needed, define public synonyms to reference database objects in others databases/schema.
4. Specify database object identifiers in lowercase.

See also [Naming database objects](#) on page 637.

Without double quotes around the database object names, all names will be converted to uppercase letters by DB2® before executing the SQL.

The DB2® schema concept:

After a connection, the database interface can automatically execute a `SET SCHEMA name` instruction if the following FGLPROFILE entry is defined:

```ini
dbi.database.dbname.db2.schema= "name"
```

Here `dbname` identifies the database name used in the BDL program (`DATABASE `dbname`) and `name` is the schema name to be used in the `SET SCHEMA` instruction. If this entry is not defined, no `SET SCHEMA` instruction is executed and the current schema defaults to the user's name.

**Examples:**

```ini
dbi.database.stores.db2.schema= "STORES1"
dbi.database.acnts.db2.schema= "ACCSCH"
```

**Note:** DB2® does not check the schema name when the `SET SCHEMA` instruction is executed. Setting a wrong schema name results in "undefined name" errors when performing subsequent SQL instructions like `SELECT`, `UPDATE`, `INSERT`.

In accordance with this automatic schema selection, you must create a DB2® schema for your application:

1. Connect as a user with the DBADM authority.
2. Create an administrator user dedicated to your application. For example, "STORESADM". Make sure this user has the `IMPLICIT_SCHEMA` privilege (this is the default in DB2®).
3. Connect as the application administrator "STORESADM" to create all database objects (tables, indexes, ...). In our example, a "STORESADM" schema will be created implicitly and all database objects will belong to this schema.
As a second option you can create a specific schema with the following SQL command:

```
CREATE SCHEMA "name" AUTHORIZATION "appadmin"
```

See the IBM DB2 manuals for more details about schemas.

**Data manipulation**
IBM DB2 related data manipulation topics.

**Reserved words**

**Informix®**
With Informix®, it is possible to create database objects with reserved words.

For example:

```
CREATE TABLE table ( char CHAR(10) );
```

Indeed this is not good practice, but Informix® SQL allows this to be backward compatible when introducing a new keyword in the SQL syntax.

Most other database systems do not allow reserved words as database identifiers. If your legacy code is using SQL reserved words of the target database SQL syntax, an error will be thrown at CREATE TABLE execution.

**IBM® DB2®**
Even if IBM® DB2® allows SQL reserved keywords as SQL object names, verify this with your existing database schema and make sure that you do not use DB2® SQL words. An example of a common word which is part of DB2® SQL grammar is 'alias'.

**Solution**
See IBM® DB2® documentation for reserved keywords.

**Outer joins**

**Informix® OUTER() syntax**
In Informix® SQL, outer joins can be defined in the FROM clause with the OUTER keyword:

```
SELECT ... FROM a, OUTER (b)
WHERE a.key = b.akey
```

```
SELECT ... FROM a, OUTER(b, OUTER(c))
WHERE a.key = b.akey
AND b.key1 = c.bkey1 AND b.key2 = c.bkey2
```

Informix® also supports the ANSI OUTER join syntax, which is the recommended way to specify outer joins with recent SQL database engines:

```
SELECT ... FROM cust LEFT OUTER JOIN order
ON cust.key = order.custno
WHERE ...
```

**IBM® DB2®**
IBM® DB2® supports the ANSI outer join syntax:

```
SELECT ...
FROM cust LEFT OUTER JOIN order
```
**Solution**

The Genero database drivers can convert Informix® OUTER specifications to ANSI outer joins.

**Note:** For better SQL portability, use the ANSI outer join syntax instead of the old Informix® OUTER syntax.

The outer join translation can be controlled with the following FGLPROFILE entry:

```
  dbi.database.dsname.ifxemul.outers = true | false
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

- **Prerequisites:**
  1. In the FROM clause, the main table must be the first item and the outer tables must be listed from left to right in the order of outer levels.

     Example which does not work:

     ```
     ... FROM OUTER(tab2), tab1
     ```

  2. The outer join in the WHERE clause must use the table name as prefix:

     ```
     ... WHERE tab1.col1 = tab2.col2
     ```

- **Restrictions:**
  1. Statements composed by 2 or more SELECT instructions are not supported:

     ```
     SELECT ... UNION SELECT ...
     ```

     or:

     ```
     SELECT ... WHERE col IN (SELECT...) 
     ```

  2. Additional conditions on outer table columns cannot be detected and therefore are not supported:

     ```
     ... FROM tab1, OUTER(tab2)
     WHERE tab1.col1 = tab2.col2
     AND tab2.colx > 10
     ```

  3. Using subscript in outer conditions:

     ```
     ... FROM tab1, OUTER(tab2)
     WHERE tab1.col[1,3] = tab2.col[1,3]
     ```

- **Notes:**
  1. Table aliases are detected in OUTER expressions.

     OUTER example with table alias:

     ```
     ... OUTER(tab1 alias1) ...
     ```

  2. In the outer join, `outertab.col` can be placed on both right or left sides of the equal sign:

     ```
     ... WHERE outertab.col1 = maintab.col2
     ```

```sql
LEFT OUTER JOIN item
ON order.key = item.ordno
ON cust.key = order.custno
WHERE order.cdate > current date
```
3. Table names detection is not case-sensitive:

```
SELECT ... FROM tab1, TAB2
    WHERE tab1.col1 = tab2.col2
```

4. Temporary tables are supported in OUTER specifications:

```
CREATE TEMP TABLE tt1 ( ... )
SELECT ... FROM tab1, OUTER(tt1) ...
```

Related concepts

**Outer joins** on page 643
Use standard ISO outer join syntax instead of the old IBM® Informix® OUTER() syntax.

**Transactions handling**

**Informix®**

With the Informix® native mode (non ANSI):

- Transactions blocks start with `BEGIN WORK` and terminate with `COMMIT WORK` or `ROLLBACK WORK`.
- Statements executed outside a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.

```
UPDATE tab1 SET ...   -- auto-committed
BEGIN WORK            -- start of TX block
UPDATE tab1 SET ...
UPDATE tab2 SET ...
...
COMMIT WORK           -- end of TX block
```

Informix® version 11.50 introduces savepoints:

```
SAVEPOINT name [UNIQUE]
ROLLBACK [WORK] TO SAVEPOINT [name]
RELEASE SAVEPOINT name
```

**IBM® DB2®**

Transactions in IBM® DB2®:

- Beginning of transactions are implicit; two transactions are delimited by `COMMIT` or `ROLLBACK`.
- DDL statements can be executed (and canceled) in transactions.

Savepoints in IBM® DB2®:

- Savepoints must be declared with the `ON ROLLBACK RETAIN CURSORS` clause.
- Rollback must always specify the savepoint name.

**Solution**

The Informix® behavior is simulated with an autocommit mode in the IBM® DB2® interface. A switch to the explicit commit mode is done when a `BEGIN WORK` is performed by the BDL program. Regarding the transaction control instructions, the BDL applications do not have to be modified in order to work with IBM® DB2®.

**Note:** If you want to use savepoints, always specify the savepoint name in `ROLLBACK TO SAVEPOINT`.

See also **SELECT FOR UPDATE**

Related concepts

**Database transactions** on page 600
Database transactions define a set of SQL instructions to be executed as a whole, or rolled back as a whole.

**Temporary tables**

**Informix®**

Informix® temporary tables are created with the `CREATE TEMP TABLE` DDL instruction or with `SELECT ... INTO TEMP` statement:

```sql
CREATE TEMP TABLE tt1 ( pkey INT, name VARCHAR(50) )
CREATE TEMP TABLE tt2 ( pkey INT, name VARCHAR(50) ) WITH NO LOG
SELECT * FROM tab1 WHERE pkey > 100 INTO TEMP tt2
```

Temporary tables are automatically dropped when the SQL session ends, but they can also be dropped with the `DROP TABLE` command. There is no name conflict when several users create temporary tables with the same name.

BDL REPORTs can create a temporary table when the rows are not sorted externally (by the source SQL statement).

Informix® allows you to create indexes on temporary tables. No name conflict occurs when several users create an index on a temporary table by using the same index identifier.

When creating temporary tables in Informix®, the `WITH NO LOG` clause can be used to avoid the overhead of recording DML operations in transaction logs.

**IBM® DB2®**

IBM® DB2® supports the `DECLARE GLOBAL TEMPORARY TABLE` instruction.

DB2® global temporary tables are quite similar to Informix® temporary tables with some exceptions:

- A user temporary table space must exist for the database.
- Users must have ‘USE’ privilege on a ‘user temporary table space’.
- For usage, the temporary table name must be prefixed by ‘SESSION’.
- No constraints or indexes can be created on temporary tables.

**Note:** IBM® DB2® version 11.x supports the `CREATE TEMP TABLE` syntax for compatibility with Netezza. However, IBM recommends to use the `DECLARE GLOBAL TEMPORARY TABLE` syntax, to create a temporary table.

For more details, see the DB2® documentation.

**Solution**

In accordance with some prerequisites, temporary tables creation in BDL programs can be supported by the database interface.

The general FGLPROFILE entry to control temporary table emulation is:

```
  dbi.database.dsname.ifxemul.temptables = true false
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**How does it work ?**

- Informix-specific statements involving temporary table creation are automatically converted to IBM® DB2® `DECLARE GLOBAL TEMPORARY TABLE` statements.
- Once the temporary table has been created, all other SQL statements performed in the current SQL session are parsed to add the `SESSION` prefix to the table name automatically.
Prerequisites

• Fulfill the DB2\textsuperscript{®} prerequisites to create global temporary tables, at minimum you must create a user temporary table space and grant the usage to database users:

```sql
CREATE USER TEMPORARY TABLESPACE tempspace01 MANAGED BY AUTOMATIC STORAGE
GRANT USE OF TABLESPACE tempspace01 TO PUBLIC
```

See DB2\textsuperscript{®} documentation for more details.

Limitations

• Tokens matching the original table names are converted to unique names in all SQL statements. Make sure you are not using a temp table name for other database objects, like columns. The following example illustrates this limitation:

```sql
CREATE TEMP TABLE tmp1 ( col1 INTEGER, col2 CHAR(20) )
SELECT tmp1 FROM table_x WHERE ...
```

• Only the ‘native’ serial emulation mode is supported with temporary tables. See the issue about SERIALs for more details.

Related concepts
Temporary tables on page 641
Syntax for temporary table creation is not unique across all database engines.

Substrings in SQL

Informix\textsuperscript{®}

Informix\textsuperscript{®} SQL statements can use subscripts on columns defined with the character data type:

```sql
SELECT ... FROM tab1 WHERE col1[2,3] = 'RO'
SELECT ... FROM tab1 WHERE col1[10] = 'R' -- Same as col1[10,10]
UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...
SELECT ... FROM tab1 ORDER BY col1[1,3]
```

**Important:** With other database servers as Informix\textsuperscript{®}, when the subscript notation is used to modify column values in UPDATE statement, or as ORDER BY element, you will get an SQL error:

```sql
UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...
SELECT ... FROM tab1 ORDER BY col1[1,3]
```

**IBM® DB2®**

IBM® DB2\textsuperscript{®} provides different functions (SUBSTR, SUSTR2, SUBSTRING), to extract a substring from a string expression:

```sql
SELECT .... FROM tab1 WHERE SUBSTR(col1,2,2) = 'RO'
SELECT SUBSTR('Some text',6,3) ...
SELECT SUBSTRING(col1,1,3,CODEUNITS32) ...
```

**Solution**

Replace all Informix\textsuperscript{®} col[x,y] right-value expressions by SUBSTRING(col,x,y-x+1,CODEUNITS32).

Rewrite UPDATE and ORDER BY clauses using col[x,y] expressions.
The translation of \( \text{col}[x,y] \) expressions can be controlled with the following FGLPROFILE entry:

```plaintext
dbi.database.dsname.ifxemul.colsubs = \_true \_false
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

Substring expressions on page 644
Handle substrings expressions with different database engines.

**String delimiters**

**Informix**

The ANSI SQL string delimiter character is the single quote (\'string\'), while double quotes are used to delimit database object names:

```sql
SELECT ... WHERE "tabname"."colname" = 'a string value'
```

In Informix databases created in native mode (non-ANSI), you can use double quotes as string delimiters:

```sql
SELECT ... WHERE tabname.colname = 'a string value'
```

This is important, since many BDL programs use that character to delimit the strings in SQL commands.

**Note:** This problem concerns only double quotes within SQL statements. Double quotes used in pure BDL string expressions are not subject to SQL compatibility problems.

**IBM® DB2®**

IBM® DB2® follows the ANSI SQL specification, using single quotes for string delimiters and double quotes for database object names.

**Solution**

When using Static SQL statements, the fglcomp compiler converts string literals using double quotes to string literals with single quotes:

```bash
$ cat s.4gl
MAIN
  DEFINE n INT
  SELECT COUNT(*) INTO n FROM tab1 WHERE col1 = "abc"
END MAIN
$ fglcomp -S s.4gl
s.4gl^3^SELECT COUNT(*) FROM tab1 WHERE col1 = 'abc'
```

However, SQL statements created dynamically are not modified by the Genero compiler.

The Genero database interface can automatically replace all double quotes by single quotes in SQL statements. This applies to static and dynamic SQL statements.

The The translation of double quoted expression to single quoted expressions can be controlled with the following FGLPROFILE entry:

```plaintext
dbi.database.dbname.ifxemul.dbquotes = \_true \_false
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.
However, database object names must not be delimited by double quotes, because the database interface cannot determine the difference between a database object name and a quoted string! For example, if the program executes the SQL statement:

```
... WHERE "tabname"."colname" = "a string value"
```

replacing all double quotes by single quotes would produce:

```
... WHERE 'tabname'.'colname' = 'a string value'
```

This would produce an error since 'tabname'.'colname' is not allowed by ORACLE.

Escaped string delimiters can be used inside strings like the following:

```
'This is a single quote: '''
'This is a single quote: \\
"This is a double quote: '"'
"This is a double quote: \\
```

Although double quotes are replaced automatically in SQL statements, it is recommended that you use only single quotes to enforce portability.

**Related concepts**

- **String literals in SQL statements** on page 634
  
  Single quotes is the standard for delimiting string literals in SQL.

**Single row SELECT**

**Informix®**

With Informix®, you must use the system table with a condition on the table id:

```
SELECT user FROM systables WHERE tabid=1
```

**IBM® DB2®**

To retrieve a single row with IBM® DB2®, use the following syntax:

```
SELECT user FROM SYSIBM.SYSTABLES WHERE NAME='SYSTABLE'
```

**Solution**

Check the BDL sources for "FROM systables WHERE tabid=1" and use dynamic SQL to resolve this problem.

Consider writing a **FUNCTION** which produces the **FROM** and **WHERE** part, depending on the target database type.

**MATCHES and LIKE**

**Informix®**

Informix® supports **MATCHES** and **LIKE** operators in SQL statements.

**MATCHES** expects * and ? wild-card characters, while **LIKE** uses the % and _ wild-cards as equivalents.

```
( col MATCHES 'Smi*' AND col NOT MATCHES 'R?x' )
( col LIKE 'Smi%' AND col NOT LIKE 'R_x' )
```
MATCHES accepts also brackets notation, to specify a set of matching characters at a given position:

```
( col MATCHES '[Pp]aris' )
( col MATCHES '[0-9][a-z]+' )
```

**IBM® DB2®**

IBM® DB2® does not provide an equivalent of the Informix® MATCHES operator.

The LIKE operator is supported.

**Important:** With IBM® DB2®, columns defined as CHAR (N) are blank padded, and trailing blanks are significant in the LIKE expressions. As result, with a CHAR(5) value such as 'abc ' (with 2 trailing blanks), the expression (colname LIKE 'ab_') will not match. To workaround this behavior, you can do (RTRIM(colname) LIKE 'pattern'). However, consider adding the condition AND (colname LIKE 'pattern%') to force the DB server to optimize the query of the column is indexed. The CONSTRUCT instruction uses this technique when the entered criteria does not end with a * star wildcard.

**Solution**

The database driver is able to translate Informix® MATCHES expressions to LIKE expressions, when no [ ] bracket character ranges are used in the MATCHES operand.

The MATCHES to LIKE expression translation is controlled by the following FGLPROFILE entry:

```
  dbi.database.dbname.ifxemul.matches = [true | false]
```

**Important:** Only [NOT] MATCHES followed by a search pattern provided as a string literal can be converted by ODI drivers. A [NOT] MATCHES followed by a ? question mark parameter place holder is not translated!

For maximum portability, consider replacing the MATCHES expressions with LIKE expressions in all SQL statements.

Avoid using CHAR (N) types for variable length character data (such as name, address).

**Related concepts**

MATCHES and LIKE operators on page 645
Use the standard LIKE operator instead of the MATCHES operator.

**SELECT * (asterisk)**

**Informix®**

Informix® allows you to use the star character in the select list along with other expressions:

```
SELECT coll, * FROM tab1 ...
```

**IBM® DB2®**

IBM® DB2® does not support the asterisk notation after another expression in the SELECT list.

Use the table name as a prefix to the star:

```
SELECT coll, tab1.* FROM tab1 ...
```

**Solution**

Always use the table name with stars.
The LENGTH() function

**Informix®**

Informix® provides the LENGTH() function to count the number of bytes of a character string expression:

```
SELECT LENGTH("aaa"), LENGTH(col1) FROM table
```

Informix® LENGTH() does not count the trailing blanks for CHAR or VARCHAR expressions, while Oracle counts the trailing blanks.

Informix® LENGTH() returns 0 when the given string is empty. That means, LENGTH('') = 0.

**IBM® DB2®**

IBM® DB2® supports the LENGTH() function, but there are some differences with Informix® LENGTH().

The IBM® DB2® LENGTH() function counts trailing blanks. When using a CHAR column, values are blank padded, and the function returns the size of the CHAR column. When using a VARCHAR column, trailing blanks are significant, and the function returns the number of characters, including trailing blanks.

**Solution**

Check if the trailing blanks are significant when using the LENGTH() SQL function in your application.

To count the number of characters by ignoring the trailing blanks, you must use the RTRIM() function:

```
SELECT LENGTH(RTRIM(col1)) FROM table
```

The translation of LENGTH() expressions can be controlled with the following FGLPROFILE entry:

```
dbi.database.dsn.name.ifxemul.length = true | false
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

The LENGTH() function in SQL on page 647

The semantics of the LENGTH() SQL function differs according to the database engine.

**Row limiting clause**

**Informix®**

Informix® SQL supports the SKIP and FIRST/LIMIT keywords to limit the number of rows of a result set:

```
SELECT SKIP 10 FIRST 20 customer.* FROM customer ... ORDER BY cust_name
```

This Informix SQL syntax is not portable.

Recent database engines support the row limiting clause syntax defined by the SQL standard:

```
SELECT ... OFFSET n ROWS FETCH FIRST m ROWS ONLY
```

This should be the preferred syntax to be used, if all target database types support this SELECT clause.

The ODI database drivers can convert the Informix SQL SKIP/FIRST row limiting clause to a native SQL equivalent, if the row limiting clause parameters are simple integer literals (the clause is not translated when using SQL parameters / program variables).
**Important:** In addition to the `SKIP/FIRST` clause of the projection clause, Informix SQL supports also a `LIMIT` clause after the `ORDER BY` clause:

```sql
SELECT customer.* FROM customer ... ORDER BY cust_name LIMIT 10
```

This Informix SQL syntax construction is not converted by the ODI drivers. To benefit from the conversion, review the code to use the Informix SQL `SKIP/FIRST` clause instead.

**IBM® DB2®**

IBM® DB2® supports the following row limiting clause:

```
SELECT ... ORDER BY ... OFFSET n ROWS FETCH FIRST m ROWS ONLY
```

**Solution**

The Informix SQL row limiting clause can be converted by the IBM® DB2® driver to the native SQL equivalent clause, when the parameters are simple integer literals.

**Note:** The row limiting clause must not use SQL parameters. Only row limiting clauses using integer constants will be converted.

The translation of the Informix SQL row limiting clause can be controlled with the following FGLPROFILE entry:

```
dbi.database.dsnname.ifxemul.rowlimiting = true false
```

For more details see [IBM Informix emulation parameters in FGLPROFILE](on page 668).

**Related concepts**

- [Row limiting clause (SELECT)](on page 650)
- [How to use the right clause to limit the number of rows produced by a SELECT statement?](BDL programming)

**INSERT cursors**

**Informix®**

Informix® provides *insert cursors* to optimize row creation in a database. An insert cursor is declared as a cursor, and rows as added with the `PUT` instruction. The rows are buffered and sent to the database server when executing a `FLUSH` instruction, or when the cursor is closed with `CLOSE`. When using transactions in Informix®, the `OPEN`, `PUT` and `FLUSH` instructions must be executed within a transaction block.

```sql
DECLARE c1 CURSOR FOR INSERT INTO tab1 ...
BEGIN WORK
OPEN c1
WHILE ...
    PUT c1 USING var-list
END WHILE
CLOSE c1
COMMIT WORK
```

**IBM® DB2®**

IBM® DB2® does not support insert cursors.

**Solution**

Insert cursors are emulated by the database interface, using basic `INSERT` SQL instructions.
The performances might be not as good as with Informix®, but the feature is fully supported.

**Related concepts**
- [Insert cursors](#) on page 633
- Using insert cursors with non-Informix databases.

**Cursors WITH HOLD**

**Informix®**

Informix® closes opened cursors automatically when a transaction ends, unless the **WITH HOLD** option is used in the **DECLARE** instruction:

```sql
DECLARE c1 CURSOR WITH HOLD FOR SELECT ... 
OPEN c1
BEGIN WORK
FETCH c1 ...
COMMIT WORK
FETCH c1 ...
CLOSE c1
```

**IBM® DB2®**

IBM® DB2® can keep cursors open across transactions, when the transaction is terminated with **COMMIT**.

When a transaction is canceled with a **ROLLBACK**, DB2® automatically closes all cursors.

**Solution**

Normally, BDL cursors declared **WITH HOLD** remain open even after terminating a transaction with a **COMMIT WORK** or **ROLLBACK WORK**.

Since IBM® DB2® closes cursors when a transaction is rolled back, you must check that your source code does not use **WITH HOLD** cursors after transactions canceled with **ROLLBACK WORK**.

For consistency with other database brands, database cursors that are not declared **WITH HOLD** are automatically closed, when a **COMMIT WORK** or **ROLLBACK WORK** is performed.

**Important**: Opening a **WITH HOLD** cursor declared with a **SELECT FOR UPDATE** results in an SQL error; in the same conditions, this does not normally appear with Informix®. Review the program logic in order to find another way to set locks.

**Related concepts**
- [Cursors WITH HOLD](#) on page 631
- Programming WITH HOLD cursors using **SELECT** with and without **FOR UPDATE** clause.

**SELECT ... FOR UPDATE**

**Informix®**

Legacy BDL programs typically use a cursor with **SELECT FOR UPDATE** to implement pessimistic locking and avoid several users editing the same rows:

```sql
DECLARE cc CURSOR FOR 
SELECT ... FROM tab WHERE ... FOR UPDATE 
OPEN cc
FETCH cc <-- lock is acquired 
... 
CLOSE cc <-- lock is released 
```

The row must be fetched in order to set the lock.
If the cursor is local to a transaction, the lock is released when the transaction ends. If the cursor is declared WITH HOLD, the lock is released when the cursor is closed.

Informix® provides the `SET LOCK MODE` instruction to define the lock wait timeout:

```
SET LOCK MODE TO \[ \[ \[ WAIT \| NOT WAIT \| WAIT \[ seconds \] \] \] \] \]
```

The default mode is NOT WAIT.

**IBM® DB2®**

IBM® DB2® locks are released when closing the cursor or when the transaction ends.

IBM® DB2®'s locking granularity is at the row level.

IBM® DB2® has no equivalent for `SET LOCK MODE TO NOT WAIT`: The system lock timeout parameter can be changed, but this is a database parameter, global to all SQL processes.

**Solution**

The database interface is based on an emulation of an Informix® engine using transaction logging. Therefore, opening a `SELECT ... FOR UPDATE` cursor declared outside a transaction will raise an SQL error -255 (not in transaction).

When used, you must review pessimistic locking implementation, because it is based on the NOT WAIT mode, which is not supported by IBM® DB2®.

**Related concepts**

- **Cursors WITH HOLD** on page 631
- Programming WITH HOLD cursors using `SELECT` with and without `FOR UPDATE` clause.

**UPDATE/DELETE ... WHERE CURRENT OF**

**Informix®**

Informix® allows positioned UPDATEs and DELETEs with the "WHERE CURRENT OF cursor" clause, if the cursor has been DECLARED with a `SELECT ... FOR UPDATE` statement.

**IBM® DB2®**

`UPDATE/DELETE ... WHERE CURRENT OF` is supported by the IBM® DB2® CLI interface.

**Solution**

With IBM® DB2®, `UPDATE/DELETE ... WHERE CURRENT OF` instructions are executed as is without SQL translation.

As a replacement of WHERE CURRENT OF, if the database table is defined with a primary key column, use the value fetched from the `SELECT [FOR UPDATE]` cursor in the WHERE clause of the `UPDATE/DELETE` statement.

**Related concepts**

- Positioned UPDATE/DELETE on page 631
- Using positioned updates/deletes with named database cursors.

**SQL parameters limitation**

The IBM® DB2® SQL parser does not allow some uses of the '?' SQL parameter marker.

The following SQL expressions are not supported:

```
? IS [NOT] NULL
? operator ?
```
SQL instructions containing these expressions raise an error during the statement preparation.

Solution

Check that your BDL programs do not use these types of conditional expressions.

If you really need to test a BDL variable during the execution of a SQL statement, you must use the CAST() function for DB2® only:

```sql
... WHERE CAST( ? AS INTEGER ) IS NULL
```

See the DB2® documentation for more details.

LOAD and UNLOAD

Informix®

Informix® provides two SQL instructions to export / import data from / into a database table:

The UNLOAD instruction copies rows from a database table into a text file:

```sql
UNLOAD TO "filename.unl" SELECT * FROM tab1 WHERE ..
```

The LOAD instructions insert rows from a text file into a database table:

```sql
LOAD FROM "filename.unl" INSERT INTO tab1
```

IBM® DB2®

IBM® DB2® does not provide LOAD and UNLOAD instructions.

Solution

LOAD and UNLOAD instruction are implemented in the Genero BDL runtime system with basic INSERT (for LOAD) or SELECT (for UNLOAD) SQL commands. The LOAD and UNLOAD instruction can be supported with various database servers.

However, LOAD and UNLOAD require the description of the column types in order to work, that can lead to some differences in the data formatting.

Note: If no transaction is started, the LOAD instruction will automatically execute a BEGIN WORK and COMMIT WORK when finished, or ROLLBACK WORK if a row insertion failed while loading. Terminating a transaction will automatically close cursors not defined WITH HOLD option. To workaround this situation, see more details in the LOAD on page 736 reference topic.

The LOAD and UNLOAD BDL instructions are supported with IBM® DB2®.

Related concepts

LOAD and UNLOAD instructions on page 639

The LOAD and UNLOAD instructions can produce different data formats depending on the database server type.

SQL Interruption

Informix®

With Informix®, it is possible to interrupt a long running query if the SQL INTERRUPT ON option.
IBM® DB2®

IBM® DB2® supports SQL Interruption: The db client must issue an SQLCancel () ODBC call to interrupt a query.

Solution

The IBM® DB2® database driver supports SQL interruption and converts the native SQL error code -952 to the Informix® error code -213.

Related concepts

Using SQL interruption on page 601
Interrupt long running SQL queries, or interrupt queries waiting for locked data.

Scrollable cursors

Informix®

Informix® SQL and Genero BDL support scrollable cursors when you specify the SCROLL clause in the DECLARE cursor instruction:

```
DECLARE c1 SCROLL CURSOR FOR SELECT ...
```

Important: Informix does not allow to fetch TEXT/BYTE columns with scrollable cursors. If you declare a scroll cursor with a SELECT containing TEXT/BYTE columns, Informix will produce the SQL error -611 when executing the OPEN instruction.

IBM® DB2®

IBM® DB2® supports native scrollable cursors with the CLI SQL_CURSOR_STATIC statement attribute.

Solution

The IBM® DB2® database driver uses the native scrollable cursors by setting the CLI statement attribute SQL_ATTR_CURSOR_TYPE to SQL_CURSOR_STATIC.

Important: With IBM DB2 is it NOT possible to use LOB columns in a scrollable cursor. If TEXT/BYTE columns are used with a scrollable cursor, the OPEN instruction will produce the SQL error -611 (as with Informix).

Related concepts

Scrollable cursors on page 623
How scrollable cursors can be supported on different databases.

Stored procedure calls

IBM® DB2® supports stored procedures as a saved collection of SQL statements, which can accept and return user-supplied parameters.

IBM® DB2® stored procedures can also produce one or more result sets.

To create a stored procedure in an IBM® DB2® database, use the CREATE PROCEDURE statement. See reference documentation for more details.

Stored procedures with output parameters

IBM® DB2® stored procedures must be called with the input and output parameters specification in the USING clause of the EXECUTE, OPEN or FOREACH instruction. As in normal dynamic SQL, parameters must correspond by position and the IN/OUT/INOUT options must match the parameter definition of the stored procedure.

To execute the stored procedure, you must use the CALL SQL instruction:

```
PREPARE stmt FROM "call procl(?,?,?)"
```
Here is a complete example creating and calling a stored procedure with output parameters:

```sql
MAIN
  DEFINE n INTEGER
  DEFINE d DECIMAL(6,2)
  DEFINE c VARCHAR(200)
  DATABASE test1
  EXECUTE IMMEDIATE
    "create procedure proc1("
    "  in p1 int,"
    "  out p2 decimal(6,2),"
    "  inout p3 varchar(20)"
    " )"
    " language sql begin"
    "  set p2 = p1 + 0.23;"
    "  set p3 = 'Value = ' || char(p1);"
    "end"
PREPARE stmt FROM "call proc1(?,?,?)"
LET n = 111
EXECUTE stmt USING n IN, d OUT, c INOUT
DISPLAY d
DISPLAY c
END MAIN
```

**Stored procedures with result set**

With IBM® DB2®, you can execute stored procedures returning a result set. To do so, you must declare a cursor and fetch the rows:

```sql
MAIN
  DEFINE i, n INTEGER
  DEFINE d DECIMAL(6,2)
  DEFINE c VARCHAR(200)
  DATABASE test1
  CREATE TABLE tab1 ( c1 INTEGER, c2 DECIMAL(6,2), c3 VARCHAR(200) )
  INSERT INTO tab1 VALUES ( 1, 123.45, 'aaaaaa' )
  INSERT INTO tab1 VALUES ( 2, 123.66, 'bbbbbbbb' )
  INSERT INTO tab1 VALUES ( 3, 444.77, 'cccccc' )
  EXECUTE IMMEDIATE "create procedure proc2( in key integer )"
    " result sets 1"
    " language sql"
    " begin"
    " declare c1 cursor with return for"
    "  select * from tab1 where c1 > key;"
    " open c1;"
    " end"
DECLARE curs CURSOR FROM "call proc2(?)"
LET i = 1
FOREACH curs USING i INTO n, d, c
  DISPLAY n, d, c
END FOREACH
END MAIN
```

**Stored procedures with output parameters and result set**

It is possible to execute IBM® DB2® stored procedures with output parameters and a result set. The output parameter values are available after the OPEN cursor instruction:

```sql
OPEN curs USING n IN, d OUT, c INOUT
FETCH curs INTO rec.*
```
**IBM® Netezza®**

**Support versions**
Genero BDL supports the following IBM Netezza versions:

- IBM Netezza 6.x
- IBM Netezza 7.x

**Installation (Runtime Configuration)**
IBM® Netezza® related installation topics.

**Install IBM® Netezza® and create a database - database configuration/design tasks**
If you are tasked with installing and configuring the database, here is a list of steps to be taken:

1. **An IBM® Netezza® appliance (the server) must be available.**
2. **Install the IBM® Netezza® client software with the IBM® Netezza® ODBC driver on the application server.**
3. **Create an IBM® Netezza® database with the nzsq1 utility.**
   
   You must connect to the "system" database:

   ```
   $ nzsq1 -h hostname system username password
   ```

4. **Create your database with the following SQL command:**

   ```
   CREATE DATABASE mydatabase
   ```

   **Note:** The default character set for CHAR/VARCHAR types will be latin9. Unicode/UTF-8 character string data must be stored in NCHAR/NVARCHAR columns.

5. **Create a database user dedicated to the administration of the new database and grant privileges:**

   ```
   CREATE USER myadmin WITH PASSWORD 'password' ...
   GRANT ALL PRIVILEGES on mydatabase TO myadmin;
   GRANT ALL ADMIN TO myadmin;
   ```

6. **Create the application tables.**
   
   Convert Informix® data types to Netezza® data types. See [Data type conversion table: Informix to Netezza](#) on page 802 for more details.

7. **If you plan to use the SERIAL emulation, you must prepare the database.**

   See [SERIAL and BIGSERIAL data types](#) on page 810 for more details.

**Prepare the runtime environment - connecting to the database**

1. **In order to connect to IBM® Netezza®, you must have the “dbmnt.z” driver in $FGLDIR/dbdrivers.**
2. **The IBM® Netezza® client software with ODBC driver is required to connect to a server.**
   
   Check if the ODBC client library (libnzodbc.*) is installed on the machine where the BDL programs run.

3. **Make sure that the IBM® Netezza® client environment variables are properly set.**
   
   Check for example NZ_DIR (the path to the installation directory), NZ_ODBC_INI_PATH (the path to the ODBC data source file), etc. See IBM® Netezza® documentation for more details.

4. **Verify the environment variable defining the search path for Netezza database client shared libraries (libnzodbc.so on UNIX™, ODBC32.DLL on Windows™).**
Table 214: Shared library environment setting for IBM® Netezza®

<table>
<thead>
<tr>
<th>IBM® Netezza® version</th>
<th>Shared library environment setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM® Netezza® 6 and higher</td>
<td><em>UNIX</em>: Add $NZ_DIR/lib (for 32 bit) or $NZ_DIR/lib64 (for 64 bit) to LD_LIBRARY_PATH (or its equivalent).&lt;br&gt;<em>Windows</em>: Add %NZ_DIR%\bin to PATH.</td>
</tr>
</tbody>
</table>

5. Check the database client locale settings.

With IBM® Netezza®, there is no configuration setting to defined database client: When using CHAR/VARCHAR columns, the application character set (LC_ALL, LANG) must be Latin-9 / ISO-8859-15. When using NCHAR/NVARCHAR columns, the application character set must be UTF-8.

6. You can test the client environment by trying to connect to the server with the SQL command line tool:

```
$ nzsql -h hostname system username password
```

7. Set up the FGLPROFILE entries for database connections.

   a) Define the Netezza database driver:

   ```
dbi.database.dbname.driver = "dbmntz"
```

   b) The "source" parameter defines the name of the ODBC source.

   ```
dbi.database.dbname.source = "test1"
```

---

**Database concepts**

IBM® Netezza® related database concepts topics.

**Database concepts**

Like Informix® servers, Netezza® can handle multiple database entities. Tables created by a user can be accessed without the owner prefix by other users as long as they have access privileges to these tables.

**Solution**

Create a Netezza® database for each Informix® database.

**Tip:** If you have several Informix database entities, migrating from the Informix database to another database it is a good opportunity to centralize all tables in a single database. To avoid conflicts with table names, use a prefix when needed.

**Concurrency management**

**Data consistency and concurrency concepts**

- *Data Consistency* applies to situations when readers want to access data currently being modified by writers.
- *Concurrent Data Access* applies to situations when several writers are accessing the same data for modification.
- *Locking Granularity* defines the amount of data concerned when a lock is set (for example, row, page, table).

**Informix®**

Informix® uses a locking mechanism to handle data consistency and concurrency. When a process changes database information with UPDATE, INSERT or DELETE, an exclusive lock is set on the touched rows. The lock remains active until the end of the transaction. Statements performed outside a transaction are treated as a transaction containing a single operation and therefore release the locks immediately after execution. SELECT statements can set shared locks, depending on isolation level. In case of locking conflicts (for example, when two processes want to
acquire an exclusive lock on the same row for modification, or when a writer is trying to modify data protected by a shared lock), the behavior of a process can be changed by setting the lock wait mode.

Control:

- Lock wait mode: SET LOCK MODE TO ...
- Isolation level: SET ISOLATION TO ...
- Locking granularity: CREATE TABLE ... LOCK MODE {PAGE|ROW}
- Explicit exclusive lock: SELECT ... FOR UPDATE

Defaults:

- The default isolation level is READ COMMITTED.
- The default lock wait mode is NOT WAIT.
- The default locking granularity is PAGE.

**Netezza®**

Netezza® servers are designed for Data Warehouse applications, not for OLTP applications: Concurrent data access is not the best thing that a Netezza® server can do. There are a bunch of limitations that you must be aware of. You must not expect to be able to migrate an existing OLTP application running against Informix® or Oracle® to a Netezza® database server. The purpose of a Netezza-based application is mostly to do queries, with few insert or updates. Typically a Netezza® database is fed with data by using tools such as nzload, not by Genero BDL programs.

Some limitations of Netezza®:

- An application can only execute one cursor (or statement handle) at a time.
- Singular data modification statements (INSERT, UPDATE, DELETE) are much slower than with traditional OLTP database servers. Netezza® is, however, very good when it comes to loading a huge amount of data with special tools like the nzload utility.
- SELECT ... FOR UPDATE is not supported. Regular SELECT statement never lock rows.
- Locks can only be set for an entire table with LOCK TABLE.
- A maximum of 31 concurrent INSERT processes are allowed (Netezza® V6), and there must be only INSERT statements in a transaction block.
- UPDATE/DELETE statements lock the entire table, but don’t prevent SELECT statements. Other processes doing UPDATE/DELETE statements will wait until the first session has committed.
- Netezza® (V6) understands the SET TRANSACTION ISOLATION statement, but currently implements only the SERIALIZABLE level.
- There is no way to define the LOCK WAIT mode. With Netezza®, processes always wait for locks to be released.

**Solution**

Understand that the main difference with Informix® is that Netezza® is not good at concurrent data modification. Note also that readers do not have to wait for writers in Netezza®.

It is recommended that Genero applications mainly do queries against a Netezza® server. You must review your program logic that modifies data, keeping in mind that only one process can modify a table at the time. Note however, that if you write short transactions, this is not visible to the end users, except that an INSERT / UPDATE / DELETE of a single row takes more time than with another database server.

The SET ISOLATION TO ... Informix® syntax is replaced by SET TRANSACTION ISOLATION LEVEL ... in Netezza®. However, only the REPEATABLE READ level is supported with Netezza®.

The table shows the isolation level mappings done by the Netezza® database driver:

**Table 215: Isolation level mappings done by the Netezza® database driver**

<table>
<thead>
<tr>
<th>SET ISOLATION instruction in program</th>
<th>Native SQL command</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET ISOLATION TO DIRTY READ</td>
<td>Not supported (SQL Error)</td>
</tr>
<tr>
<td>SET ISOLATION instruction in program</td>
<td>Native SQL command</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>SET ISOLATION TO COMMITTED READ [READ COMMITTED] [RETAIN UPDATE LOCKS]</td>
<td>Not supported (SQL Error)</td>
</tr>
<tr>
<td>SET ISOLATION TO CURSOR STABILITY</td>
<td>Not supported (SQL Error)</td>
</tr>
<tr>
<td>SET ISOLATION TO REPEATABLE READ</td>
<td>SET TRANSACTION ISOLATION LEVEL SERIALIZABLE</td>
</tr>
</tbody>
</table>

**Important:** Since Netezza® does not support the lock wait mode, you must check that your programs do not include a SET LOCK MODE instruction. This instruction will fail with error -6370 if it is executed when connected to Netezza®.

See the Informix® and Netezza® documentation for more details about data consistency, concurrency and locking mechanisms.

**Related concepts**

Concurrent data access on page 599
Understanding concurrent data access and data consistency.

Optimistic locking on page 624
Implementing optimistic locking to handle access concurrently to the same database records.

Cursors WITH HOLD on page 631
Programming WITH HOLD cursors using SELECT with and without FOR UPDATE clause.

**Transactions handling**

**Informix®**

With the Informix® native mode (non ANSI):

- Transactions blocks start with BEGIN WORK and terminate with COMMIT WORK or ROLLBACK WORK.
- Statements executed outside a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.

```plaintext
UPDATE tab1 SET ... -- auto-committed
BEGIN WORK           -- start of TX block
UPDATE tab1 SET ...
UPDATE tab2 SET ...
...                  -- end of TX block
COMMIT WORK
```

**Informix®** version 11.50 introduces savepoints:

```plaintext
SAVEPOINT name [UNIQUE]
ROLLBACK [WORK] TO SAVEPOINT [name]  
RELEASE SAVEPOINT name
```

**Netezza®**

- Transactions are started with BEGIN WORK.
- Transactions are validated with COMMIT WORK.
- Transactions are canceled with ROLLBACK WORK.
- Statements executed outside of a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.
- If an SQL error occurs in a transaction, the whole transaction is aborted.
- A transaction must only contain INSERTs if you want concurrent processes to insert rows at the same time (UPDATEs/DELETEs lock the whole table).
Only the **SERIALIZABLE** isolation level is implemented by Netezza®.

Netezza® cancels the entire transaction if an SQL error occurs in one of the statements executed inside the transaction. The following code example illustrates this difference:

```sql
CREATE TABLE tab1 ( k INT PRIMARY KEY, c CHAR(10) )
WHENEVER ERROR CONTINUE
BEGIN WORK
INSERT INTO tab1 (1, 'abc')
SELECT FROM unexisting WHERE key = 123  -- unexisting table = sql error
COMMIT WORK
```

With Informix®, the above code will leave the table with one row inside, since the first INSERT statement succeeded. With Netezza®, the table will remain empty after executing this piece of code, because the server will rollback the whole transaction.

**Solution**

Regarding the transaction control instructions, the BDL applications do not have to be modified in order to work with Netezza®: Informix® transaction handling commands are automatically converted to Netezza® instructions to start, validate or cancel transactions. However, since Netezza® is not designed for OLTP applications, you must review any code doing complex data modifications. See the concurrency topic for more details.

You must review the SQL statements inside `BEGIN WORK / COMMIT WORK` instruction and check if these can raise an SQL error. To get the same behavior in case of error when connected to a different database than Netezza®, you must issue a `ROLLBACK` to cancel all the SQL statements that succeeded in the transaction, for example with a `TRY/CATCH` block.

```sql
TRY
  BEGIN WORK
  ...
  COMMIT WORK
CATCH
  ROLLBACK WORK
END TRY
```

**Related concepts**

- **Database transactions** on page 600
  Database transactions define a set of SQL instructions to be executed as a whole, or rolled back as a whole.

**Database users**

**Informix®**

Until version 11.70.xC2, Informix® database users must be created at the operating system level and must be members of the 'informix' group.

Starting with 11.70.xC2, Informix® supports database-only users with the `CREATE USER` instruction, as in most other db servers.

Any database user must have sufficient privileges to connect and use resources of the database; user rights are defined with the `GRANT` command.

**Netezza®**

Netezza® users must be registered in the database with the `CREATE USER` command, for example:

```sql
CREATE USER name WITH PASSWORD 'pswd' IN GROUP ...
```

See the Netezza® documentation for more details about user creation and database access/security.
Solution

Depending on the application logic (is it a multiuser application?), you have to create one or several Netezza® users.

Related concepts

(Database users and security on page 608

Properly identifying database users allows to use database security and audit features.

Data dictionary

(IBM® Netezza® related data dictionary topics.

Data type conversion table: Informix to Netezza

Table 216: Data type conversion table (Informix to Netezza)

<table>
<thead>
<tr>
<th>Informix® data types</th>
<th>Netezza® data types</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR(n)</td>
<td>CHAR(n) if SBCS, NCHAR(n) if UTF-8</td>
</tr>
<tr>
<td>VARCHAR(n[,m])</td>
<td>VARCHAR(n) if SBCS, NVARCHAR(n) if UTF-8</td>
</tr>
<tr>
<td>NCHAR(n)</td>
<td>NCHAR(n) (UTF-8)</td>
</tr>
<tr>
<td>NVARCHAR(n[,m])</td>
<td>NVARCHAR(n) (UTF-8)</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>BIGINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>INT8</td>
<td>BIGINT</td>
</tr>
<tr>
<td>SERIAL[(start)]</td>
<td>N/A (see note 1)</td>
</tr>
<tr>
<td>BIGSERIAL[(start)]</td>
<td>N/A (see note 1)</td>
</tr>
<tr>
<td>SERIAL8[(start)]</td>
<td>N/A (see note 1)</td>
</tr>
<tr>
<td>DOUBLE PRECISION / FLOAT[(n)]</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>REAL / SMALLFLOAT</td>
<td>REAL</td>
</tr>
<tr>
<td>DECIMAL(p,s)</td>
<td>DECIMAL(p,s)</td>
</tr>
<tr>
<td>DECIMAL(p)</td>
<td>DECIMAL(p*2,p)</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL(32,16)</td>
</tr>
<tr>
<td>MONEY(p,s)</td>
<td>DECIMAL(p,s)</td>
</tr>
<tr>
<td>MONEY(p)</td>
<td>DECIMAL(p,2)</td>
</tr>
<tr>
<td>MONEY</td>
<td>DECIMAL(16,2)</td>
</tr>
<tr>
<td>TEXT</td>
<td>N/A</td>
</tr>
<tr>
<td>BYTE</td>
<td>N/A</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME HOUR TO MINUTE</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME HOUR TO SECOND</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME HOUR TO FRACTION(n)</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME YEAR TO MONTH</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>DATETIME YEAR TO DAY</td>
<td>TIMESTAMP</td>
</tr>
</tbody>
</table>
### Informix® data types vs. Netezza® data types

<table>
<thead>
<tr>
<th>Informix® data types</th>
<th>Netezza® data types</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATETIME YEAR TO HOUR</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>DATETIME YEAR TO MINUTE</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>DATETIME YEAR TO SECOND</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>DATETIME YEAR TO FRACTION(n)</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>INTERVAL YEAR[(p)] TO MONTH</td>
<td>CHAR(50)</td>
</tr>
<tr>
<td>INTERVAL YEAR[(p)] TO YEAR</td>
<td>CHAR(50)</td>
</tr>
<tr>
<td>INTERVAL MONTH[(p)] TO MONTH</td>
<td>INTERVAL</td>
</tr>
<tr>
<td>INTERVAL DAY[(p)] TO FRACTION(n)</td>
<td>INTERVAL</td>
</tr>
<tr>
<td>INTERVAL DAY[(p)] TO SECOND</td>
<td>INTERVAL</td>
</tr>
<tr>
<td>INTERVAL DAY[(p)] TO MINUTE</td>
<td>INTERVAL</td>
</tr>
<tr>
<td>INTERVAL DAY[(p)] TO HOUR</td>
<td>INTERVAL</td>
</tr>
<tr>
<td>INTERVAL DAY[(p)] TO DAY</td>
<td>INTERVAL</td>
</tr>
<tr>
<td>INTERVAL HOUR[(p)] TO FRACTION(n)</td>
<td>INTERVAL</td>
</tr>
<tr>
<td>INTERVAL HOUR[(p)] TO SECOND</td>
<td>INTERVAL</td>
</tr>
<tr>
<td>INTERVAL HOUR[(p)] TO MINUTE</td>
<td>INTERVAL</td>
</tr>
<tr>
<td>INTERVAL HOUR[(p)] TO HOUR</td>
<td>INTERVAL</td>
</tr>
<tr>
<td>INTERVAL MINUTE[(p)] TO FRACTION(n)</td>
<td>INTERVAL</td>
</tr>
<tr>
<td>INTERVAL MINUTE[(p)] TO SECOND</td>
<td>INTERVAL</td>
</tr>
<tr>
<td>INTERVAL MINUTE[(p)] TO MINUTE</td>
<td>INTERVAL</td>
</tr>
<tr>
<td>INTERVAL SECOND[(p)] TO FRACTION(n)</td>
<td>INTERVAL</td>
</tr>
<tr>
<td>INTERVAL SECOND[(p)] TO SECOND</td>
<td>INTERVAL</td>
</tr>
<tr>
<td>INTERVAL FRACTION[(p)] TO FRACTION(n)</td>
<td>INTERVAL</td>
</tr>
</tbody>
</table>

Notes:

1. For more details about serial emulation, see [SERIAL and BIGSERIAL data types](#) on page 810.

### BOOLEAN data type

**Informix®**

Informix® supports the BOOLEAN data type, which can store 't' or 'f' values.

Genero BDL implements the BOOLEAN data type in a different way: A BOOLEAN variable stores integer values 1 or 0 (for TRUE or FALSE). This type is designed to hold the result of a boolean expression.

**Netezza®**

Netezza® supports the BOOLEAN data type and stores 't' or 'f' values for TRUE and FALSE representation.

**Note:** It is not possible to insert the integer values 1 or 0 into Netezza BOOLEAN columns: Values must be true, false, 't', 'f', '1' or '0'.

**Solution**

The Netezza® database interface converts the BOOLEAN integer values to a CHAR(1) values '1' or '0'.
The BOOLEAN type translation can be controlled with the following FGLPROFILE entry:

```
dbi.database.dsnname.ifxemul.datatype.boolean = [ true | false ]
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

- **Using portable data types** on page 614

Only a limited set of data types are really portable across several database engines.

### CHAR and VARCHAR data types

**Informix®**

Informix® supports the following character data types:

**Table 217: Informix® character data types**

<table>
<thead>
<tr>
<th>Informix® data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR (n)</td>
<td>SBCS and MBCS character data (max is 32767 bytes)</td>
</tr>
<tr>
<td>VARCHAR (n,m]</td>
<td>SBCS and MBCS character data (max is 255 bytes)</td>
</tr>
<tr>
<td>NCHAR (n)</td>
<td>Same as CHAR, with specific collation order</td>
</tr>
<tr>
<td>NVARCHAR (n,m]</td>
<td>Same as VARCHAR, with specific collation order</td>
</tr>
<tr>
<td>LVARCHAR (n)</td>
<td>max size varies depending on the IDS version</td>
</tr>
</tbody>
</table>

With Informix®, both CHAR/VARCHAR and NCHAR/NVARCHAR data types can be used to store single-byte or multibyte encoded character strings. The only difference between CHAR/VARCHAR and NCHAR/NVARCHAR is in how they use sorting: N[VAR]CHAR types use the collation order, while [VAR]CHAR types use the byte order.

The character set used to store strings in CHAR/VARCHAR/NCHAR/NVARCHAR columns is defined by the DB_LOCALE environment variable.

The character set used by applications is defined by the CLIENT_LOCALE environment variable.

Informix® uses Byte Length Semantics (the size N that you specify in [VAR]CHAR (N) is expressed in bytes, not characters as in some other databases)

**Netezza®**

Netezza® supports following data types to store character data:

**Table 218: Netezza® character data types**

<table>
<thead>
<tr>
<th>Netezza® data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR (n)</td>
<td>SBCS character data using the database character set, where n is specified in bytes (max is 64000 bytes)</td>
</tr>
<tr>
<td>VARCHAR (n)</td>
<td>SBCS character data using the database character set, where n is specified in bytes (max is 64000 bytes)</td>
</tr>
<tr>
<td>NCHAR (n)</td>
<td>Unicode/UTF-8 character data, where n is specified in characters (max is 16000 characters)</td>
</tr>
<tr>
<td>NVARCHAR (n)</td>
<td>Unicode/UTF-8 character data, where n is specified in characters (max is 16000 characters)</td>
</tr>
</tbody>
</table>
Netezza® uses the Latin-9 (ISO-8859-15) code set for CHAR/VARCHAR columns, and UTF-8 for NCHAR/NVARCHAR columns.

No automatic character set conversion is done by the Netezza® client software. When using CHAR/VARCHAR columns, the client application character set (LC_ALL, LANG) must match the Latin-9/ISO-8859-15 character set. When using NCHAR/NVARCHAR columns, the client application character set must be UTF-8.

Solution

When your application uses a Latin-9 / ISO-8859-15 locale, create tables with the CHAR/VARCHAR SQL types. To store UNICODE (UTF-8) character strings, use the NCHAR/NVARCHAR SQL types instead. In program sources you can use CHAR/VARCHAR; these types can hold single and multibyte character sets, based on the C POSIX locale.

Important: Netezza® (V6 while writing these lines) supports only the latin9 database character set for CHAR/VARCHAR types. Since character set conversion is not supported, your application can only use Latin-9 or UTF-8.

When using a multibyte character set (such as UTF-8), define database columns as NCHAR and NVARCHAR, with the size in character units, and use character length semantics in BDL programs with FGL_LENGTH_SEMANTICS=CHAR.

When extracting a database schema from a Netezza® database, the fglbsch schema extractor uses the size of the column in characters, not the octet length. If you have created a CHAR(10 (characters) ) column a in Netezza® database using the UTF-8 character set, the .sch file will get a size of 10, that will be interpreted by FGL_LENGTH_SEMANTICS as a number of bytes or characters.

With Netezza, it is not possible to defined the database client client character set: The locale used by programs must match the database locale.

See also the section about Localization.

The CHAR/VARCHAR type translation can be controlled with the following FGLPROFILE entries:

```plaintext
dbi.database.dsnname.ifxemul.datatype.char = { true | false }
dbi.database.dsnname.ifxemul.datatype.varchar = { true | false }
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

Related concepts

CHAR and VARCHAR types on page 616
Using the CHAR and VARCHAR data types with different databases.

Numeric data types

Informix®

Informix® supports several data types to store numbers:

Table 219: Informix® numeric data types

<table>
<thead>
<tr>
<th>Informix® data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>16 bit signed integer</td>
</tr>
<tr>
<td>INTEGER</td>
<td>32 bit signed integer</td>
</tr>
<tr>
<td>BIGINT</td>
<td>64 bit signed integer</td>
</tr>
<tr>
<td>INT8</td>
<td>64 bit signed integer (replaced by BIGINT)</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>Equivalent to DECIMAL(16)</td>
</tr>
<tr>
<td>DECIMAL(p)</td>
<td>Floating-point decimal number (max precision is 32)</td>
</tr>
<tr>
<td>DECIMAL(p,s)</td>
<td>Fixed-point decimal number (max precision is 32)</td>
</tr>
</tbody>
</table>
**Informix® data type** | **Description**
---|---
MONEY | Equivalent to DECIMAL(16,2)
MONEY (p) | Equivalent to DECIMAL (p, 2) (max precision is 32)
MONEY (p, s) | Equivalent to DECIMAL (p, s) (max precision is 32)
REAL / SMALLFLOAT | 32-bit floating point decimal (C float)
DOUBLE PRECISION / FLOAT[(n)] | 64-bit floating point decimal (C double)

**Netezza®**

Netezza® supports the following data types to store numbers:

**Table 220: Netezza® numeric data types**

<table>
<thead>
<tr>
<th>Netezza® data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTEINT</td>
<td>8-bit value with the range -128 to 127</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>16 bit signed integer</td>
</tr>
<tr>
<td>INTEGER</td>
<td>32 bit signed integer</td>
</tr>
<tr>
<td>BIGINT</td>
<td>64 bit signed integer</td>
</tr>
<tr>
<td>DECIMAL (p, s)</td>
<td>Exact decimal number with p significant digits and s decimals (1&lt;=p&lt;=38)</td>
</tr>
<tr>
<td>DECIMAL (p)</td>
<td>Integer with precision p (1&lt;=p&lt;=38)</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>Integer, same as NUMERIC(18,0)</td>
</tr>
<tr>
<td>FLOAT (p) with 1 &lt;= p &lt;= 6</td>
<td>16 bit approx floating point (C float)</td>
</tr>
<tr>
<td>FLOAT (p) with 7 &lt;= p &lt;= 15</td>
<td>32 bit approx floating point (C double)</td>
</tr>
<tr>
<td>REAL</td>
<td>Same as FLOAT (6)</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>Same as FLOAT (15)</td>
</tr>
</tbody>
</table>

**Solution**

Use the following conversion rules to map Informix numeric types to Netezza® numeric types:

**Table 221: Informix® numeric data types and Netezza® equivalents**

<table>
<thead>
<tr>
<th>Informix® data type</th>
<th>Netezza® equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>INT8 / BIGINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>DECIMAL (p, s)</td>
<td>DECIMAL (p, s)</td>
</tr>
<tr>
<td>DECIMAL (p)</td>
<td>DECIMAL (p*2, p)</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL (32, 16)</td>
</tr>
<tr>
<td>MONEY (p, s)</td>
<td>DECIMAL (p, s)</td>
</tr>
<tr>
<td>MONEY (p)</td>
<td>DECIMAL (p, 2)</td>
</tr>
<tr>
<td>MONEY</td>
<td>DECIMAL (16, 2)</td>
</tr>
<tr>
<td>Informix® data type</td>
<td>Netezza® equivalent</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>SMALLFLOAT</td>
<td>REAL</td>
</tr>
<tr>
<td>FLOAT[(n)]</td>
<td>FLOAT[(n)] (DOUBLE)</td>
</tr>
</tbody>
</table>

**Important:**

There is no Netezza® equivalent for the Informix® DECIMAL (p) floating point decimal (i.e. without a scale). If your application uses such data types, you must review the database schema in order to use Netezza® compatible types.

To workaround the Netezza® limitation, the NTZ database drivers converts DECIMAL (p) types to a DECIMAL( 2*p, p ). to store all possible numbers that an Informix® DECIMAL(p) can store. However, the original Informix® precision cannot exceed 19, since the Netezza® maximum DECIMAL precision is 38 (2*19). If the original precision is bigger than 19, a CREATE TABLE statement executed from a Genero program will fail with an SQL error.

The numeric types translation can be controlled with the following FGLPROFILE entries:

```plaintext
dbi.database.dsname.ifxemul.datatype.smallint = { true | false }
dbi.database.dsname.ifxemul.datatype.integer = { true | false }
dbi.database.dsname.ifxemul.datatype.bigint = { true | false }
dbi.database.dsname.ifxemul.datatype.int8 = { true | false }
dbi.database.dsname.ifxemul.datatype.decimal = { true | false }
dbi.database.dsname.ifxemul.datatype.money = { true | false }
dbi.database.dsname.ifxemul.datatype.float = { true | false }
dbi.database.dsname.ifxemul.datatype.smallfloat = { true | false }
```

For more details see [IBM Informix emulation parameters in FGLPROFILE](#) on page 668.

**Related concepts**

**Using portable data types** on page 614

Only a limited set of data types are really portable across several database engines.

### DATE and DATETIME data types

**Informix®**

Informix® provides two data types to store date and time information:

- DATE = for year, month and day storage.
- DATETIME = for year to fraction (1-5) storage.

The DATE type is stored as an INTEGER with the number of days since 1899/12/31.

The DATETIME type can be defined with various time units, by specifying a start and end qualifier. For example, you can define a datetime to store an hour-to-second time value with `DATETIME HOUR TO SECOND`.

The values of Informix® DATETIME can be represented with a character string literal, or as DATETIME () literals:

```plaintext
'2017-12-24 15:45:12.345' -- a DATETIME YEAR TO FRACTION(3)
'15:45' -- a DATETIME HOUR TO MINUTE
DATETIME(2017-12-24 12:45) YEAR TO MINUTE
DATETIME(12:45:56.333) HOUR TO FRACTION(3)
```

Informix® is able to convert quoted strings to DATE/DATETIME data, if the string contains matching environment parameters. The string to date conversion rules for DATE is defined by the DBDATE environment variable. The string to datetime format for DATETIME is defined by the GL_DATETIME environment variable.

**Note:** Within Genero programs, the string representation for DATETIME values is always ISO (YYYY-MM-DD hh:mm:ss.ffffff)
Informix® supports date arithmetic on DATE and DATETIME values. The result of an arithmetic expression involving dates/times is an INTEGER number of days when only DATE values are used, and an INTERVAL value if a DATETIME is used in the expression.

Informix® automatically converts an INTEGER to a DATE when the integer is used to set a value of a date column.

Informix® provides the CURRENT \[ q1 TO q2 \] operator, to get the system date/time on the server where the current database is located. When no qualifiers are specified, CURRENT returns a DATETIME YEAR TO FRACTION(3). Informix also supports the SYSDATE operator, which returns the current system time as a DATETIME YEAR TO FRACTION(5).

Note: The USEOSTIME configuration parameter must be set to 1 in order to get the subsecond precision in CURRENT and SYSDATE operators. See Informix documentation for more details.

Netezza®

Netezza® supports the following data types to store date/time values:

Table 222: Netezza® date/time data types

<table>
<thead>
<tr>
<th>Netezza® data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>for year, month, day storage</td>
</tr>
<tr>
<td>TIME</td>
<td>for hour, minute, second, fraction with (6 decimal positions)</td>
</tr>
<tr>
<td>TIME WITH TIME ZONE / TIMEZ</td>
<td>same as TIME, with time zone information</td>
</tr>
<tr>
<td>TIMetz</td>
<td>for year, month, day, hour, minute, second, fraction (with 6 decimal positions)</td>
</tr>
</tbody>
</table>

Like Informix®, Netezza® can convert quoted strings to date time data. Netezza® accepts different date formats, including ISO date time strings, and you can specify the cast operator (:date, :time, :timestamp) after the string literal.

With Netezza®, the result of an arithmetic expression involving DATE values is an INTEGER representing a number of days.

Complex DATETIME expressions (involving INTERVAL values for example) are Informix® specific and have no equivalent in Netezza®.

Solution

Use the following conversion rules to map Informix date/time types to Netezza® date/time types:

Table 223: Informix® data types and Netezza® equivalents

<table>
<thead>
<tr>
<th>Informix® data type</th>
<th>Netezza® data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME HOUR TO MINUTE</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME HOUR TO SECOND</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME HOUR TO FRACTION(n)</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME YEAR TO MONTH</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>DATETIME YEAR TO DAY</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>DATETIME YEAR TO HOUR</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>DATETIME YEAR TO MINUTE</td>
<td>TIMESTAMP</td>
</tr>
</tbody>
</table>
The `DATE` and `DATETIME` types translation can be controlled with the following FGLPROFILE entries:

```
dbi.database.dsname.ifxemul.datatype.date = [true | false]
dbi.database.dsname.ifxemul.datatype.datetime = [true | false]
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

Netezza® and Informix® `DATE` data type are equivalent and store year, month, day values.

Netezza® `TIME` data type can be used to store Informix® `DATETIME HOUR TO FRACTION(n)`, `DATETIME HOUR TO SECOND` and `DATETIME HOUR TO MINUTE` values, and any other `DATETIME` type with qualifiers `HOUR`, `MINUTE`, `SECOND`, `FRACTION(n)`. Missing time parts default to 00:00:00.0. For example, when using a `DATETIME MINUTE TO SECOND` with the value of "23:11:45", the Netezza® `TIME` value will be "00:23:11:45.0".

Informix® `DATETIME` values with any precision from `YEAR` to `FRACTION(5)` can be stored in Netezza® `TIMESTAMP` columns. The database interface makes the conversion automatically. Missing date or time parts default to 1900-01-01 00:00:00.0. For example, when using a `DATETIME DAY TO MINUTE` with the value of "23 11:45", the Netezza® `TIMESTAMP` value will be "1900-01-23 11:45:00.0".

**Note:** Informix® supports implicit `DATE/DATETIME` conversions, for example you can use a `DATE` variable when the target column is a `DATETIME`. This is not possible with Netezza®: The type of the SQL parameter must match the type of the column in the database table. Make sure that you are using the same type for the SQL parameter and the target column, `DATE/DATETIME` implicit conversion is not supported by Netezza®.

**Related concepts**

- Date/time literals in SQL statements on page 635
- Good practices for date and time handling in SQL.

**INTERVAL data type**

**Informix®**

Informix® provides the `INT`ERVAL data type to store a value that represents a span of time.

INTERVAL types are divided into two classes:

- `year-month` intervals. For example: `INTERVAL YEAR(5) TO MONTH`
- `day-time` intervals. For example: `INTERVAL DAY(9) TO SECOND`

INTERVAL columns can be defined with various time units, by specifying a start and end qualifier. For example, you can define an interval to store a number of hours and minutes with `INTERVAL HOUR(n) TO MINUTE`, where `n` defines the maximum number of digits for the hours unit.

The values of Informix® INTERVAL can be represented with a character string literal, or as `INTERVAL()` literals:

```
'-9834 15:45:12.345'  -- an INTERVAL DAY(6) TO FRACTION(3)
'7623-11'  -- an INTERVAL YEAR(9) TO MONTH
INTERVAL(18734:45) HOUR(5) TO MINUTE
INTERVAL(-7634-11) YEAR(5) TO MONTH
```

**Netezza®**

Netezza® implements the `INTERVAL` data type in a different way than Informix® does:
• Netezza® allows you to specify interval qualifiers (YEAR, MONTH, DAY, ...) but internally it always uses the same base type, storing values of any combination of units. Thus, there is no way to distinguish year-month intervals and day-time intervals with Netezza®.
• The precision of Netezza® intervals includes fraction of seconds with up to 6 significant digits. However, it is not possible to specify the scale of a Netezza® interval as with the Informix® FRACTION(p) qualifier.
• With Netezza®, interval literals must be include the units, as "−923 days 11 hours 22 minutes". While Informix® interval literals have the form INTERVAL(999−99...) start-qualifier TO end-qualifier.
• Netezza® normalizes all INTERVAL values to units of seconds, and considers a month to be thirty days for the purpose of interval comparisons. This approximation can lead to inaccuracies.

Solution

The Informix® INTERVAL types of the day-time class can be mapped to the native Netezza® INTERVAL type, for day to second time interval storage.

Since Netezza® does not clearly distinguish year-month interval class, such types are converted to CHAR(50) by the Netezza® driver.

Important: Netezza® (V6 at the time of writing) has several bugs regarding the INTERVAL type; we do not recommend using this type until Netezza® has fixed these problems.

The INTERVAL types translation can be controlled with the following FGLPROFILE entry:

```
dbi.database.dsname.ifxemul.datatype.interval = ↓ true ↓ false ↓
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

Related concepts

Using portable data types on page 614

Only a limited set of data types are really portable across several database engines.

SERIAL and BIGSERIAL data types

Informix®

Informix® supports the SERIAL, BIGSERIAL data types to produce automatic integer sequences:

• SERIAL can produce 32 bit integers (INTEGER)
• BIGSERIAL can produced 64 bit integers (BIGINT)
• SERIAL8 is a synonym for BIGSERIAL

Steps to use serials with Informix®:

1. Create the table with a column using SERIAL, or BIGSERIAL.
2. To generate a new serial, no value or a zero value is specified in the INSERT statement:

```
INSERT INTO tab1 ( c ) VALUES ( 'aa' )
INSERT INTO tab1 ( k, c ) VALUES ( 0, 'aa' )
```

3. After INSERT, the new value of a SERIAL column is provided in SQLCA.SQLERRD[2], while the new value of a BIGSERIAL value must be fetched with a SELECT dbinfo('bigserial') query.

Informix® allows you to insert rows with a value different from zero for a serial column. Using an explicit value will automatically increment the internal serial counter, to avoid conflicts with future INSERT statements that are using a zero value:

```
CREATE TABLE tab ( k SERIAL); -- internal counter = 0
INSERT INTO tab VALUES ( 0 ); -- internal counter = 1
INSERT INTO tab VALUES ( 10 ); -- internal counter = 10
INSERT INTO tab VALUES ( 0 ); -- internal counter = 11
```
DELETE FROM tab; -- internal counter = 11
INSERT INTO tab VALUES ( 0 ); -- internal counter = 12

Netezza

Netezza® does not have a SERIAL data type.

Netezza® Version 6 supports sequences, but not triggers.

Solution

The lack of trigger support in Netezza® makes it impossible to emulate Informix® SERIALs.

Note: For best SQL portability when using different types of databases, consider using sequences as described in Solution 3: Use native SEQUENCE database objects on page 628.

If you are using Informix® SERIAL or BIGSERIAL columns, you must review the application logic and database schema to replace these columns with INTEGER/BIGINT columns, and generate the new keys from a SEQUENCE, as described in the SQL Programming page.

Related concepts

Auto-incremented columns (serials) on page 626
How to implement automatic record keys.

ROWID columns

Informix

When creating a table, Informix® automatically adds a ROWID integer column (applies to non-fragmented tables only).

The ROWID column is auto-filled with a unique number and can be used like a primary key to access a given row.

Note: Informix® ROWID usage was a common practice in the early days of Informix® 4GL programming. Today it is recommended to define all your database tables with a PRIMARY KEY to uniquely identify rows.

With Informix®, the SQLCA.SQLERRD[6] register contains the ROWID of the last row affected by an INSERT, UPDATE or DELETE statement.

Netezza

Netezza® implements ROWIDs like Informix®, except that the rowids are stored in a 64 bit integer.

Solution

If your Genero BDL application uses rowid columns, review the program logic to use primary keys instead. If the database table does no define a primary key, it should be added. All references to SQLCA.SQLERRD[6] must be removed, because this variable will not hold the ROWID of the last modified row.

ROWIDs can be used with Netezza® as with Informix®, as long as you fetch rowid values into a BIGINT variable. However, the SQLCA.SQLERRD[6] register cannot be supported, because Netezza® rowids are 64 bit integers (BIGINT) while SQLCA.SQLERRD[6] is a 32 bit integer (INTEGER).

For databases where the keyword of the rowid pseudo-column is different than "ROWID", the translation can be controlled with the following FGLPROFILE entry:

dbi.database.dsnname.ifxemul.rowid = □ true □ false □

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

Related concepts

Using ROWID columns on page 644
Automatic ROWID columns is not a common database feature.

**Indexes**

Like most database servers, Informix® supports index creation on table columns. Indexes can be used to make the server find rows rapidly:

```
CREATE INDEX cust_ix1 ON customer (cust_name)
```

Netezza® does not support index creation on tables. There is no need for indexes in a Netezza® database because performance is achieved by distributing data rows over several disks. Netezza® tracks min/max values of each column per disk extent to ignore extents which do not contain the values the query is looking for. See Netezza® documentation for more details.

**Solution**

You must remove all CREATE INDEX instructions from your programs and SQL scripts that create database tables.

**TEXT and BYTE (LOB) types**

**Informix®**

Informix® provides the TEXT, BYTE, CLOB and BLOB data types to store very large texts or binary data.

Legacy Informix® 4GL applications typically use the TEXT and BYTE types.

Genero BDL does not support the Informix® CLOB and BLOB types.

**Netezza®**

Netezza® does not support large objects in the database.

**Solution**

If your application need to store large objects with TEXT and BYTE data types, you cannot use a Netezza® server.

**Related concepts**

[Using portable data types](#) on page 614.

Only a limited set of data types are really portable across several database engines.

**Table constraints**

**Informix®**

Informix® supports primary key, unique, foreign key, default and check constraints.

The constraint naming syntax is different in Informix® and most other databases: Informix expects the constraint name after the constraint definition:

```
CREATE TABLE emp (  
    ...  
    emp_code CHAR(10) UNIQUE CONSTRAINT pk_emp,  
    ...  
)
```

While other SQL database brands require to specify the constraint name before the constraint definition:

```
CREATE TABLE emp (  
    ...  
    emp_code CHAR(10) CONSTRAINT pk_emp UNIQUE,  
    ...  
)
Netezza® supports primary key, unique, foreign key, default and check constraints in a limited manner.

**Important:** Netezza® allows you to create tables with the `UNIQUE` and `PRIMARY KEY` and `FOREIGN KEY` syntax, but the constraints are not enforced.

### Constraint naming syntax

The constraint naming clause must be placed **before** the constraint specification.

### Solution

The database interface does not convert constraint naming expressions when creating tables from BDL programs. Review the database creation scripts to adapt the constraint-naming clauses for Netezza®.

Since Netezza® does not enforce constraints, you must test for unique values and foreign key references at the program level.

**Related concepts**

- [Data definition statements](#) on page 613
- It is recommended to avoid use of DDL in programs.

### Name resolution of SQL objects

#### Informix®

Informix® uses the following form to identify an SQL object:

```sql
database[@dbservername]:[][{owner|"owner"}.]identifier
```

The ANSI convention is to use double quotes for identifier delimiters (For example: "customer"."cust_name").

Informix® database object names are not case-sensitive in non-ANSI databases. When using double-quoted identifiers, Informix® becomes case sensitive.

With non-ANSI Informix® databases, you do not have to give a schema name before the tables when executing an SQL statement:

```sql
SELECT ... FROM customer WHERE ...
```

In Informix® ANSI compliant databases:

- The table name must include "owner", unless the connected user is the owner of the database object.
- The database server shifts the owner name to uppercase letters before the statement executes, unless the owner name is enclosed in double quotes.

#### Netezza®

With Netezza®, an object name takes the following form:

```sql
[database.[schema].]identifier
```

### Solution

To write portable SQL, regarding database object names:
1. Use simple database object names (without any owner/schema prefix)
2. Do not use double quotes to surround database object identifiers.
3. If needed, define public synonyms to reference database objects in others databases/schema.
4. Specify database object identifiers in lowercase.

See also Naming database objects on page 637.

**Data manipulation**

IBM® Netezza® related data manipulation topics.

**Reserved words**

**Informix®**

With Informix®, it is possible to create database objects with reserved words.

For example:

```sql
CREATE TABLE table ( char CHAR(10) );
```

Indeed this is not good practice, but Informix® SQL allows this to be backward compatible when introducing a new keyword in the SQL syntax.

Most other database systems do not allow reserved words as database identifiers. If your legacy code is using SQL reserved words of the target database SQL syntax, an error will be thrown at CREATE TABLE execution.

**Netezza®**

In Netezza®, SQL object names like table and column names cannot be SQL reserved keywords.

**Solution**

Table or column names which are Netezza® reserved keywords must be renamed.

See the Netezza® SQL Reference guide for a list of reserved keywords.

**Outer joins**

**Informix® OUTER() syntax**

In Informix® SQL, outer joins can be defined in the FROM clause with the OUTER keyword:

```sql
SELECT ... FROM a, OUTER (b) 
  WHERE a.key = b.akey

SELECT ... FROM a, OUTER(b, OUTER(c)) 
  WHERE a.key = b.akey 
  AND b.key1 = c.bkey1 AND b.key2 = c.bkey2
```

Informix® also supports the ANSI OUTER join syntax, which is the recommended way to specify outer joins with recent SQL database engines:

```sql
SELECT ... FROM cust LEFT OUTER JOIN order 
  ON cust.key = order.custno 
  WHERE ...
```

**Netezza®**

Netezza® supports the ANSI outer join syntax:

```sql
SELECT ...
```
Solution

The Genero database drivers can convert Informix® OUTER specifications to ANSI outer joins.

**Note:** For better SQL portability, use the ANSI outer join syntax instead of the old Informix® OUTER syntax.

The outer join translation can be controlled with the following FGLPROFILE entry:

```
  dbi.database.dsnname.ifxemulouters = true false
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

- **Prerequisites:**
  1. In the FROM clause, the main table must be the first item and the outer tables must be listed from left to right in the order of outer levels.

    Example which does not work:

    ```
    ... FROM OUTER(tab2), tab1
    ```

  2. The outer join in the WHERE clause must use the table name as prefix:

    ```
    ... WHERE tab1.col1 = tab2.col2
    ```

- **Restrictions:**
  1. Statements composed by 2 or more SELECT instructions are not supported:

    ```
    SELECT ... UNION SELECT ...
    ```

    or:

    ```
    SELECT ... WHERE col IN (SELECT...)
    ```

  2. Additional conditions on outer table columns cannot be detected and therefore are not supported:

    ```
    ... FROM tab1, OUTER(tab2)
    WHERE tab1.col1 = tab2.col2
    AND tab2.colx > 10
    ```

  3. Using subscript in outer conditions:

    ```
    ... FROM tab1, OUTER(tab2)
    WHERE tab1.col1[1,3] = tab2.col2[1,3]
    ```

- **Notes:**

  1. Table aliases are detected in OUTER expressions.

    OUTER example with table alias:

    ```
    ... OUTER(tab1 alias1) ...
    ```

  2. In the outer join, outertab.col can be placed on both right or left sides of the equal sign:

    ```
    ... WHERE outertab.col1 = maintab.col2
    ```
3. Table names detection is not case-sensitive:

```sql
SELECT ... FROM tab1, TAB2
  WHERE tab1.col1 = tab2.col2
```

4. Temporary tables are supported in OUTER specifications:

```sql
CREATE TEMP TABLE tt1 ( ... )
SELECT ... FROM tab1, OUTER(tt1) ...
```

**Related concepts**

*Outer joins* on page 643

Use standard ISO outer join syntax instead of the old IBM® Informix® OUTER () syntax.

**Transactions handling**

**Informix®**

With the Informix® native mode (non ANSI):

- Transactions blocks start with `BEGIN WORK` and terminate with `COMMIT WORK` or `ROLLBACK WORK`.
- Statements executed outside a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.

```sql
UPDATE tab1 SET ... -- auto-committed
BEGIN WORK -- start of TX block
UPDATE tab1 SET ...
UPDATE tab2 SET ...
...
COMMIT WORK -- end of TX block
```

Informix® version 11.50 introduces savepoints:

```sql
SAVEPOINT name [UNIQUE]
ROLLBACK [WORK] TO SAVEPOINT [name]
RELEASE SAVEPOINT name
```

**Netezza®**

- Transactions are started with `BEGIN WORK`.
- Transactions are validated with `COMMIT WORK`.
- Transactions are canceled with `ROLLBACK WORK`.
- Statements executed outside of a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.
- If an SQL error occurs in a transaction, the whole transaction is aborted.
- A transaction must only contain INSERTs if you want concurrent processes to insert rows at the same time (UPDATEs/DELETEs lock the whole table).
- Only the `SERIALIZABLE` isolation level is implemented by Netezza®.

Netezza® cancels the entire transaction if an SQL error occurs in one of the statements executed inside the transaction. The following code example illustrates this difference:

```sql
CREATE TABLE tab1 ( k INT PRIMARY KEY, c CHAR(10) )
WHENEVER ERROR CONTINUE
BEGIN WORK
  INSERT INTO tab1 ( 1, 'abc' )
SELECT FROM unexisting WHERE key = 123 -- unexisting table = sql error
COMMIT WORK
```
With Informix®, the above code will leave the table with one row inside, since the first INSERT statement succeeded. With Netezza®, the table will remain empty after executing this piece of code, because the server will rollback the whole transaction.

**Solution**

Regarding the transaction control instructions, the BDL applications do not have to be modified in order to work with Netezza®: Informix® transaction handling commands are automatically converted to Netezza® instructions to start, validate or cancel transactions. However, since Netezza® is not designed for OLTP applications, you must review any code doing complex data modifications. See the concurrency topic for more details.

You must review the SQL statements inside BEGIN WORK / COMMIT WORK instruction and check if these can raise an SQL error. To get the same behavior in case of error when connected to a different database than Netezza®, you must issue a ROLLBACK to cancel all the SQL statements that succeeded in the transaction, for example with a TRY/CATCH block:

```
TRY
  BEGIN WORK
  ...
  COMMIT WORK
CATCH
  ROLLBACK WORK
END TRY
```

**Related concepts**

- Database transactions on page 600
  Database transactions define a set of SQL instructions to be executed as a whole, or rolled back as a whole.

**Temporary tables**

- **Informix®**
  Informix® temporary tables are created with the CREATE TEMP TABLE DDL instruction or with SELECT ... INTO TEMP statement:

```
CREATE TEMP TABLE tt1 ( pkey INT, name VARCHAR(50) )
CREATE TEMP TABLE tt2 ( pkey INT, name VARCHAR(50) ) WITH NO LOG
SELECT * FROM tab1 WHERE pkey > 100 INTO TEMP tt2
```

Temporary tables are automatically dropped when the SQL session ends, but they can also be dropped with the DROP TABLE command. There is no name conflict when several users create temporary tables with the same name.

BDL REPORTs can create a temporary table when the rows are not sorted externally (by the source SQL statement). Informix® allows you to create indexes on temporary tables. No name conflict occurs when several users create an index on a temporary table by using the same index identifier.

When creating temporary tables in Informix®, the WITH NO LOG clause can be used to avoid the overhead of recording DML operations in transaction logs.

- **Netezaa**
  Netezza® support temporary tables as Informix®, with a little syntax difference in the SELECT INTO TEMP instruction:

```
SELECT * INTO TEMP temptab FROM source
```

**Solution**

Temporary tables are well supported with native Netezza® temp tables.
**Important:**

Simple Informix-style SQL statement creating temporary tables can be converted to a native SQL equivalent instruction. However, complex SQL statements such as `SELECT .. INTO TEMP` with subqueries may fail. In such cases, create a view from the complex query and then create the temp table from the view. Or, disable Informix emulation and use the native SQL syntax to create the temporary table (EXECUTE IMMEDIATE "/* fglhint_no_ifxemul */ ...")

With Informix SQL, if the source table has a column defined as SERIAL or BIGSERIAL, a `SELECT ... INTO TEMP` will produce a new temp table with an auto-incremented serial column. With the `SELECT ... INTO TEMP` emulation for non-Informix databases, not using the native sequence generators (such as IDENTITY columns in SQL Server), the resulting temporary table will get a simple INTEGER or BIGINT column, instead of an auto-incremented column.

The general FGLPROFILE entry to control temporary table emulation is:

```
|dbi.database.dsname.ifxemul.temptables = true |false |
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

Temporary tables on page 641  
Syntax for temporary table creation is not unique across all database engines.

**Substrings in SQL**

**Informix**

Informix® SQL statements can use subscripts on columns defined with the character data type:

```
SELECT ... FROM tab1 WHERE col1[2,3] = 'RO'  
SELECT ... FROM tab1 WHERE col1[10] = 'R' -- Same as col1[10,10]  
UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...  
SELECT ... FROM tab1 ORDER BY col1[1,3]
```

**Important:** With other database servers as Informix®, when the subscript notation is used to modify column values in UPDATE statement, or as ORDER BY element, you will get and SQL error:

```
UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...  
SELECT ... FROM tab1 ORDER BY col1[1,3]
```

**Netezza**

Netezza® provides the `SUBSTRING( ... from ... to ... )` function, to extract a substring from a string expression:

```
SELECT .... FROM tab1 WHERE SUBSTRING(col1 from 2 for 2) = 'RO'  
SELECT SUBSTRING('Some text' from 6 for 3) ... -- Gives 'tex'
```

**Solution**

Replace all Informix® `col[x,y]` right-value expressions by `SUBSTRING( col from x for (y-x+1) ).`  
Rewrite UPDATE and ORDER BY clauses using `col[x,y]` expressions.

The translation of `col[x,y]` expressions can be controlled with the following FGLPROFILE entry:

```
|dbi.database.dsname.ifxemul.colsubs = true |false |
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.
Related concepts
Substring expressions on page 644
Handle substrings expressions with different database engines.

The LENGTH() function

Informix®
Informix® provides the LENGTH() function to count the number of bytes of a character string expression:

```
SELECT LENGTH("aaa"), LENGTH(col1) FROM table
```

Informix® LENGTH() does not count the trailing blanks for CHAR or VARCHAR expressions, while Oracle counts the trailing blanks.
Informix® LENGTH() returns 0 when the given string is empty. That means, LENGTH ('') = 0.

Netezza®
Netezza® supports LENGTH() and CHARACTER_LENGTH() functions, but these count the number of characters (not bytes), and trailing blanks are significant.
The Netezza LENGTH() function ignores trailing blanks.
Netezza returns NULL if the LENGTH() parameter is NULL.

Solution
Review the program logic and make sure you do not pass NULL values to the LENGTH() SQL function.
The translation of LENGTH() expressions can be controlled with the following FGLPROFILE entry:

```
dbi.database.dsname.ifxemul.length = {true | false}
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

Related concepts
The LENGTH() function in SQL on page 647
The semantics of the LENGTH() SQL function differs according to the database engine.

Name resolution of SQL objects

Informix®
Informix® uses the following form to identify an SQL object:

```
database[@dbservername]:][{owner|"owner"}.]identifier
```

The ANSI convention is to use double quotes for identifier delimiters (For example: "customer"."cust_name").
Informix® database object names are not case-sensitive in non-ANSI databases. When using double-quoted identifiers, Informix® becomes case sensitive.

With non-ANSI Informix® databases, you do not have to give a schema name before the tables when executing an SQL statement:

```
SELECT ... FROM customer WHERE ...
```

In Informix® ANSI compliant databases:
- The table name must include "owner", unless the connected user is the owner of the database object.
• The database server shifts the owner name to uppercase letters before the statement executes, unless the owner name is enclosed in double quotes.

Netezza®

With Netezza®, an object name takes the following form:

```
[database.[schema].]identifier
```

**Solution**

To write portable SQL, regarding database object names:

1. Use simple database object names (without any owner/schema prefix)
2. Do not use double quotes to surround database object identifiers.
3. If needed, define public synonyms to reference database objects in others databases/schema.
4. Specify database object identifiers in lowercase.

See also Naming database objects on page 637.

**String delimiters**

**Informix®**

The ANSI SQL string delimiter character is the single quote (`'string'`), while double quotes are used to delimit database object names:

```
SELECT ... WHERE "tabname"."colname" = 'a string value'
```

In Informix® databases created in native mode (non-ANSI), you can use double quotes as string delimiters:

```
SELECT ... WHERE tabname.colname = 'a string value'
```

This is important, since many BDL programs use that character to delimit the strings in SQL commands.

**Note:** This problem concerns only double quotes within SQL statements. Double quotes used in pure BDL string expressions are not subject to SQL compatibility problems.

**Netezza®**

Netezza® follows the ANSI SQL specification, using single quotes for string delimiters and double quotes for database object names.

**Solution**

When using Static SQL statements, the fglcomp compiler converts string literals using double quotes to string literals with single quotes:

```
$ cat s.4gl
MAIN
   DEFINE n INT
   SELECT COUNT(*) INTO n FROM tab1 WHERE col1 = "abc"
END MAIN

$ fglcomp -S s.4gl
s.4gl^3^SELECT COUNT(*) FROM tab1 WHERE col1 = 'abc'
```

However, SQL statements created dynamically are not modified by the Genero compiler.
The Genero database interface can automatically replace all double quotes by single quotes in SQL statements. This applies to static and dynamic SQL statements.

The translation of double quoted expression to single quoted expressions can be controlled with the following FGLPROFILE entry:

```
  dbi.database.dbname.ifxemul.dblquotes = { true | false }
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

However, database object names must not be delimited by double quotes, because the database interface cannot determine the difference between a database object name and a quoted string! For example, if the program executes the SQL statement:

```
... WHERE "tabname"."colname" = "a string value"
```

replacing all double quotes by single quotes would produce:

```
... WHERE 'tabname'.'colname' = 'a string value'
```

This would produce an error since 'tabname'.'colname' is not allowed by ORACLE.

Escaped string delimiters can be used inside strings like the following:

```
'This is a single quote: '''
'This is a single quote: \\''
"This is a double quote: '""
"This is a double quote: \""
```

Although double quotes are replaced automatically in SQL statements, it is recommended that you use only single quotes to enforce portability.

**Related concepts**

*String literals in SQL statements* on page 634

Single quotes is the standard for delimiting string literals in SQL.

**MATCHES and LIKE**

**Informix®**

Informix® supports MATCHES and LIKE operators in SQL statements.

MATCHES expects * and ? wild-card characters, while LIKE uses the % and _ wild-cards as equivalents.

```
( col MATCHES 'Smi*' AND col NOT MATCHES 'R?x' )
( col LIKE 'Smi%' AND col NOT LIKE 'R_x' )
```

MATCHES accepts also brackets notation, to specify a set of matching characters at a given position:

```
( col MATCHES '[Pp]aris' )
( col MATCHES '[0-9][a-z]*' )
```

**Netezza®**

Netezza® does not provide an equivalent of the Informix® MATCHES operator.

The LIKE operator is supported.

The Netezza® ~ operator expects regular expressions as follows: ( col ~ 'a.*' )
**Important:** With Netezza®, columns defined as CHAR(N) are blank padded, and trailing blanks are significant in the LIKE expressions. As result, with a CHAR(5) value such as 'abc ' (with 2 trailing blanks), the expression (colname LIKE 'ab_') will not match. To workaround this behavior, you can do (RTRIM(colname) LIKE 'pattern'). However, consider adding the condition AND (colname LIKE 'pattern%') to force the DB server to optimize the query of the column is indexed. The CONSTRUCT instruction uses this technique when the entered criteria does not end with a * star wildcard.

**Solution**

The database driver is able to translate Informix® MATCHES expressions to LIKE expressions, when no [ ] bracket character ranges are used in the MATCHES operand.

The MATCHES to LIKE expression translation is controlled by the following FGLPROFILE entry:

```
  dbi.database.dbname.ifxemul.matches = { true | false }
```

**Important:** Only [NOT] MATCHES followed by a search pattern provided as a string literal can be converted by ODI drivers. A [NOT] MATCHES followed by a ? question mark parameter place holder is not translated!

For maximum portability, consider replacing the MATCHES expressions with LIKE expressions in all SQL statements.

Avoid using CHAR(N) types for variable length character data (such as name, address).

**Related concepts**

MATCHES and LIKE operators on page 645

Use the standard LIKE operator instead of the MATCHES operator.

**Row limiting clause**

**Informix®**

Informix® SQL supports the SKIP and FIRST/LIMIT keywords to limit the number of rows of a result set:

```
  SELECT SKIP 10 FIRST 20 customer.* FROM customer ... ORDER BY cust_name
```

This Informix SQL syntax is not portable.

Recent database engines support the row limiting clause syntax defined by the SQL standard:

```
  SELECT ... OFFSET n ROWS FETCH FIRST m ROWS ONLY
```

This should be the preferred syntax to be used, if all target database types support this SELECT clause.

The ODI database drivers can convert the Informix SQL SKIP/FIRST row limiting clause to a native SQL equivalent, if the row limiting clause parameters are simple integer literals (the clause is not translated when using SQL parameters / program variables).

**Important:** In addition to the SKIP/FIRST clause of the projection clause, Informix SQL supports also a LIMIT clause after the ORDER BY clause:

```
  SELECT customer.* FROM customer ... ORDER BY cust_name LIMIT 10
```

This Informix SQL syntax construction is not converted by the ODI drivers. To benefit from the conversion, review the code to use the Informix SQL SKIP/FIRST clause instead.
Netezza®

Netezza® supports the following row limiting clause:

```
SELECT ... ORDER BY ... LIMIT m OFFSET n
```

**Solution**

The Informix SQL row limiting clause can be converted by the Netezza® driver to the native SQL equivalent clause, when the parameters are simple integer literals.

**Note:** The row limiting clause must not use SQL parameters. Only row limiting clauses using integer constants will be converted.

The translation of the Informix SQL row limiting clause can be controlled with the following FGLPROFILE entry:

```
dbi.database.dsname.ifxemul.rowlimiting = \true \false
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

- [Row limiting clause (SELECT)](on page 650)
- How to use the right clause to limit the number of rows produced by a SELECT statement?

**BDL programming**

IBM® Netezza® related programming topics.

**UPDATE limitations in Netezza**

Netezza® has some limitations regarding the UPDATE statement:

- Like DELETE, an UPDATE statement locks the entire table.
- It is not possible to UPDATE distribution columns:
  - Netezza® database tables get distributed across all of the nodes using the distribution column. You can specify the distribution column(s) when you create the table. See Netezza® documentation for more details.
  - If you try to update a distribution column, you get error 46 "Attempt to UPDATE a distribution column".

**Solution**

Review the program logic if the UPDATE statements in your programs use distribution columns, and keep in mind that an UPDATE will lock the entire table.

**INSERT cursors**

**Informix®**

Informix® provides insert cursors to optimize row creation in a database. An insert cursor is declared as a cursor, and rows are added with the PUT instruction. The rows are buffered and sent to the database server when executing a FLUSH instruction, or when the cursor is closed with CLOSE. When using transactions in Informix®, the OPEN, PUT and FLUSH instructions must be executed within a transaction block.

```
DECLARE c1 CURSOR FOR INSERT INTO tab1 ...
BEGIN WORK
OPEN c1
WHILE ...
  PUT c1 USING var-list
END WHILE
CLOSE c1
COMMIT WORK
```
Netezza®

Netezza® does not support insert cursors.

**Solution**

Insert cursors are emulated by the database interface, using basic `INSERT` SQL instructions. The performances might be not as good as with Informix®, but the feature is fully supported.

**Related concepts**

- [Insert cursors](page 633)
- Using insert cursors with non-Informix databases.

**Cursors WITH HOLD**

**Informix®**

Informix® closes opened cursors automatically when a transaction ends, unless the `WITH HOLD` option is used in the `DECLARE` instruction:

```sql
DECLARE c1 CURSOR WITH HOLD FOR SELECT ...
OPEN c1
BEGIN WORK
FETCH c1 ...
COMMIT WORK
FETCH c1 ...
CLOSE c1
```

**Netezza®**

Netezza® does not close cursors when a transaction ends. However, cursors declared with a `SELECT FOR UPDATE` are not supported with Netezza®.

**Solution**

BDL cursors declared `WITH HOLD` remain open even after terminating a transaction with a `COMMIT WORK` or `ROLLBACK WORK`.

For consistency with other database brands, database cursors that are not declared `WITH HOLD` are automatically closed, when a `COMMIT WORK` or `ROLLBACK WORK` is performed.

**Important:** Opening a `WITH HOLD` cursor declared with a `SELECT FOR UPDATE` results in an SQL error; in the same conditions, this does not normally appear with Informix®. Review the program logic in order to find another way to set locks.

**Related concepts**

- [Cursors WITH HOLD](page 631)
- Programming `WITH HOLD` cursors using `SELECT` with and without `FOR UPDATE` clause.

**SELECT … FOR UPDATE**

**Informix®**

Legacy BDL programs typically use a cursor with `SELECT FOR UPDATE` to implement pessimistic locking and avoid several users editing the same rows:

```sql
DECLARE cc CURSOR FOR
SELECT ... FROM tab WHERE ... FOR UPDATE
OPEN cc
FETCH cc <-- lock is acquired
```
The row must be fetched in order to set the lock.

If the cursor is local to a transaction, the lock is released when the transaction ends. If the cursor is declared WITH HOLD, the lock is released when the cursor is closed.

Informix® provides the SET LOCK MODE instruction to define the lock wait timeout:

```
SET LOCK MODE TO { WAIT | NOT WAIT | WAIT seconds }
```

The default mode is NOT WAIT.

**Netezza®**

With Netezza®, locks are released when closing the cursor or when the transaction ends.

Netezza® does not support SELECT FOR UPDATE statements: Netezza® systems are designed for data warehouse applications, not for OLTP applications. In a DW context, concurrent data access is not required or a priority.

**Solution**

Review the program logic when using SELECT FOR UPDATE statements.

**Related concepts**

Cursors WITH HOLD on page 631
Programming WITH HOLD cursors using SELECT with and without FOR UPDATE clause.

**UPDATE/DELETE ... WHERE CURRENT OF**

**Informix®**

Informix® allows positioned UPDATEs and DELETEs with the "WHERE CURRENT OF cursor" clause, if the cursor has been DECLARED with a SELECT ... FOR UPDATE statement.

**Netezza®**

Netezza® servers do no support SELECT FOR UPDATE, and does not set locks.

Positioned UPDATE/DELETE with the WHERE CURRENT OF cursor clause cannot be supported with Netezza®.

**Solution**

UPDATE/DELETE ... WHERE CURRENT OF is not supported with Netezza.

As a replacement of WHERE CURRENT OF, if the database table is defined with a primary key column, use the value fetched from the SELECT [FOR UPDATE] cursor in the WHERE clause of the UPDATE/DELETE statement.

**Related concepts**

Positioned UPDATE/DELETE on page 631
Using positioned updates/deletes with named database cursors.

**LOAD and UNLOAD**

**Informix®**

Informix® provides two SQL instructions to export / import data from / into a database table:

The UNLOAD instruction copies rows from a database table into a text file:

```
UNLOAD TO "filename.unl" SELECT * FROM tab1 WHERE ..
```
The LOAD instructions insert rows from a text file into a database table:

```
LOAD FROM "filename.unl" INSERT INTO tab1
```

Netezza®
Netezza® does not support LOAD and UNLOAD instructions.
Netezza® provides tools like the nzload utility.

Solution
LOAD and UNLOAD instruction are implemented in the Genero BDL runtime system with basic INSERT (for LOAD) or SELECT (for UNLOAD) SQL commands. The LOAD and UNLOAD instruction can be supported with various database servers.

However, LOAD and UNLOAD require the description of the column types in order to work, that can lead to some differences in the data formatting.

Note: If no transaction is started, the LOAD instruction will automatically execute a BEGIN WORK and COMMIT WORK when finished, or ROLLBACK WORK if a row insertion failed while loading. Terminating a transaction will automatically close cursors not defined WITH HOLD option. To workaround this situation, see more details in the LOAD on page 736 reference topic.

The LOAD and UNLOAD BDL instructions are supported with Netezza®.

Related concepts
LOAD and UNLOAD instructions on page 639
The LOAD and UNLOAD instructions can produce different data formats depending on the database server type.

SQL Interruption

Informix®
With Informix®, it is possible to interrupt a long running query if the SQL INTERRUPT ON option.

Netezza®
Netezza® supports SQL Interruption: The db client must issue an SQLCancel() ODBC call to interrupt a query.

Important: When the statement is interrupted, Netezza® issues an automatic ROLLBACK and returns a "Transaction rolled back by user", SQL error number 46.

Solution
The Netezza® database driver supports SQL interruption and converts the native SQL error 46 to the Informix® error code -213.

Related concepts
Using SQL interruption on page 601
Interrupt long running SQL queries, or interrupt queries waiting for locked data.

Scrollable cursors

Informix®
Informix® SQL and Genero BDL support scrollable cursors when you specify the SCROLL clause in the DECLARE cursor instruction:

```
DECLARE c1 SCROLL CURSOR FOR SELECT ...
```
**Important:** Informix does not allow to fetch TEXT/BYTE columns with scrollable cursors. If you declare a scroll cursor with a SELECT containing TEXT/BYTE columns, Informix will produce the SQL error -611 when executing the OPEN instruction.

**Netezza®**
Netezza® does not support scrollable cursors.

**Solution**
The Netezza® database driver emulates scrollable cursors by fetching rows in a temporary file.

**Related concepts**
[Scrollable cursors](#) on page 623
How scrollable cursors can be supported on different databases.

**Microsoft™ SQL Server**

**Supported versions**

**Important:** When connecting to SQL Server, the ODI driver will check the SQL Server compatibility level, and produce an SQL error, if the compatibility level does not match one of the versions listed in this topic.

Genero BDL supports the following Microsoft SQL Server versions:

- Microsoft SQL Server 2012 (v11)
- Microsoft SQL Server 2014 (v12)
- Microsoft SQL Server 2016 (v13)
- Microsoft SQL Server 2017 (v14)
- Microsoft SQL Server 2019 (v15)

Genero BDL supports the following Microsoft Azure SQL Database versions:

- Microsoft Azure SQL Database v12

**Installation (Runtime Configuration)**
Microsoft™ SQL Server related installation topics.

**Install Microsoft™ SQL Server and create a database - database configuration/design tasks**
If you are tasked with installing and configuring the database, here is a list of steps to be taken:

1. Install Microsoft™ SQL Server, or setup a Microsoft™ Azure Database in the cloud.
   **Important:** The collation of the SQL Server instance defines the collation of the tempdb system database. This can matter if your programs create temporary tables with CREATE TABLE #tabname. For more details, see [Temporary tables](#) on page 861.

2. Create a SQL Server database entity with the SQL Server Management Studio.
   In the database properties:
   a) Define the database "Collation": The database collation defines the character set for CHAR/VARCHAR columns, and the "SC" collation option defines if NCHAR/NVARCHAR columns store UTF-16 characters (or UCS-2 when SC is not used). Choose the right code page / collation to get a case-sensitive database.
   b) The "Compatibility level" must be set to an SQL Server version supported by the Genero BDL.
   c) The "ANSI NULL Default" option must be set to true, if you want to have the same default NULL constraint as in Informix® (a column created without a NULL constraint will allow null values, users must specify NOT NULL to deny nulls).
   d) The "Quoted Identifiers Enabled" option must be set to false, to use database object names without quotes.

3. Create and declare a database user dedicated to your application: the application administrator.
4. If you plan to use SERIAL emulation based on triggers using a registration table, create the SERIALREG table and create the serial triggers for all tables using a SERIAL.

See SERIAL and BIGSERIAL data types on page 847.

5. Create the application tables.

Convert Informix® data types to SQL Server data types. See Data type conversion table: Informix to SQL Server on page 837. In order to make application tables visible to all users, make sure that the tables are created with the 'dbo' owner.

Prepare the runtime environment - connecting to the database

1. Genero BDL provides several database drivers based on different ODBC clients. This list describes each of them:

   Important: Configure your ODBC data source to use the appropriate Microsoft SQL Server driver.

   • On Microsoft® Windows® platforms:
     - Use an SNC (dbmsnc*) driver either with the Microsoft ODBC driver for SQL Server, or with the Microsoft SQL Native Client driver (msdn.microsoft.com):
       - For Microsoft ODBC 17 (MSODBCSQL17.DLL), use dbmsnc_17.
       - For Microsoft ODBC 13 (MSODBCSQL13.DLL), use dbmsnc_13.
       - For Microsoft SQL Native Client 11 (SQLNCLI11.DLL), use dbmsnc_11.

   • On Linux® platforms:
     - With the SNC (dbmsnc*) driver, use the Microsoft ODBC for SQL Server on Linux® client (msdn.microsoft.com):
       - For Microsoft ODBC 17 (libmsodbcsql-17.so), use dbmsnc_17.
       - For Microsoft ODBC 13 (libmsodbcsql-13.so), use dbmsnc_13.

     Minimum Microsoft ODBC for SQL Server on Linux® version: 13.0.

     • With the FTM (dbmftm*) driver, use the FreeTDS ODBC client (www.freetds.org, http://www.freetds.org/files/stable/).

     Minimum FreeTDS version: 1.00.104.

     • With the ESM (dbmesm*) driver, use the Easysoft ODBC driver for SQL Server (www.easysoft.com).

     Minimum Easysoft version 1.5; Version 1.9 is strongly recommended.

2. Check that the Genero distribution package has installed the SQL Server database driver you need. A "dbmsnc", "dbmftm", or "dbmesm" driver must exist in $FGLDIR/dbdrivers.

3. An ODBC data source must be configured to allow the BDL program to establish connections to SQL Server.

   Make sure you select the correct ODBC driver (see step 1).

   Important: When using the FTM (FreeTDS) or ESM (EasySoft) database driver, you have to define the ODBCINI and ODBCINST environment variable to point to the odbc.ini and odbcinst.ini files.

4. Install and configure the database client software:

   a) When using the SNC database driver on Windows®, you must have the "Microsoft ODBC for SQL Server" software installed on the computer running Genero applications (see msdn.microsoft.com).

   On Windows® platforms, the SNC ODI driver is linked with ODBC32.DLL; There is no need to set the PATH environment variable to a specific database client library path: The ODBC32.DLL driver manager will find the MS ODBC driver from the data source settings.

   On Windows®, the MS ODBC database client locale is defined by the Windows® regional settings for non-unicode applications, and must match the BDL application locale. The BDL application locale is usually also defined by the regional settings (the ACP), but it can be set with the LANG environment variable (typically to .fglutf8). UTF-8 can also be enabled in the Windows regional settings (then LANG does not have to be set to .fglutf8).

   On Linux® platforms, the dbmsnc_nn drivers are directly linked to the corresponding libmsodbcsql-nn.so ODBC driver library. There is no need to install the unixODBC software. The SNC...
drivers will be able to connect to SQL Server, as long as the dynamic linker can find the Microsoft ODBC driver library. The `libmsodbcsql-nn.so` shared library is a symbolic link located in `/usr/lib64`, which points to the real ODBC 13 shared library.

On Linux, the MS ODBC database client locale is always UTF-16: The `dbmsnc` ODI driver makes the required character set conversions between the BDL application locale (defined by `LANG` or `LC_ALL`) and UTF-16, for the Microsoft ODBC driver for SQL Server. Therefore, no ODBC configuration is required: Character set conversions and ODBC bindings are automatically deduced from to C application locale (`LANG/LC_ALL`).

**Important:** On both Linux and Windows, ODBC character string bindings is controlled by the `widechar` option, which is automatically selected according to the application locale. For more details see CHAR and VARCHAR data types on page 838.

Unix ODBCINI sample for MS ODBC driver for SQL Server:

```ini
[snc_msvtest1_dirac_utf8]
Driver = /usr/lib64/libmsodbcsql-13.so
Description = SQL Server ODBC 13 / DIRAC SQL Server 2017 / v14
#Server = [protocol:]server[,port]
Server = tcp:dirac,1433
Database = msvtest1
--- Always Encrypted (Column Encryption)
# ColumnEncryption = Enabled
--- Transport encryption with SSL/TLS
# Encrypt = Yes/No
# TrustServerCertificate = xxx
# Trusted_Connection=yes
```

b) When using the FTM database driver, the FreeTDS driver must be installed (see [www.freetds.org](http://www.freetds.org)). There is no need to install unixODBC: The FTM driver is directly linked to `libtdsodbc.so.0`.

Make sure the FreeTDS environment variables are properly set. Check for example FREETDS (the path to the configuration file). See FreeTDS documentation for more details.

With the FTM driver, there is no need to install a driver manager like unixODBC: The FTM database driver is linked directly with the `libtdsodbc.so` shared library. Verify the environment variable (LD_LIBRARY_PATH or equivalent) specifies the search path for that database client shared library.

You must create the `odbc.ini` and `odbcinst.ini` files to defined the data source.

**Important:** Set the TDS protocol version depending on the SQL Server version, by setting the `tds version` parameter in `freetds.conf` or `TDS_Version` in `odbc.ini`. For example, for SQL Server version 2012 and 2014, use `TDS_Version=7.3`. For more details, see the FreeTDS documentation.

Define the client character set for FreeTDS (`client charset` parameter in `freetds.conf` or `ClientCharset` parameter in `odbc.ini`). You may need to link FreeTDS with the `libiconv` library to support character set conversions.

Unix ODBCINI sample for FreeTDS driver:

```ini
[ftm_msvtest1_ida_utf8_2017]
Description = SQL Server 2017
Server = ida
Database = msvtest1
Port = 1433
TDS_Version = 7.3
ClientCharset = UTF-8
dump_file = /tmp/freetds.log
dump_file_append = yes
```

See FreeTDS documentation for more details about installation and data source configuration in ODBC files.
c) When using the ESM database driver, the EasySoft ODBC driver for SQL Server must be installed (see www.easysoft.com). There is no need to install unixODBC: The ESM driver is directly linked to libessqlsrv.so.

Make sure the EasySoft environment variables are properly set. Check for example EASYSOFT_ROOT (the path to the installation directory). See FreeTDS documentation for more details.

With the ESM driver, there is no need to install a driver manager like unixODBC. The ESM database driver is linked directly with the libessqlsrv.so shared library. Verify the environment variable (LD_LIBRARY_PATH or equivalent) specifies the search path for that database client shared library.

You must create the odbc.ini and odbcinst.ini files to define the data source.

Define the client character set for EasySoft with the Client_CSet parameter in odbc.ini. The client character set is an iconv name and must match the locale of your Genero application.

**Note:**

To support all UNICODE characters when using UTF-8 with NCHAR/NVARCHAR columns, you need to define Client_CSet=UTF-8 and Server_UCSet=UTF-16LE.

When using CHAR/VARCHAR types in the database and when the database collation is different from the client locale, you must also set the Server_CSet parameter to an iconv name corresponding to the database collation. Some examples:

- If Client_CSet=ISO-8859-15 and the db collation is Latin1_* (=CP1252), you must set Server_CSet=WINDOWS-1252 (otherwise, the characters €, Š, Ž, Ž, Ž, Ë, Ë which are encoded differently
- If Client_CSet=BIG5 and the db collation is Chinese_Taiwan_Stroke_BIN, you must set Server_CSet=BIG5HKSCS.

You must also set the following DSN parameters:

- AnsiNPW=Yes
- Mars_Connection=No
- QuotedId=No

UNIX® ODBCINI sample for EasySoft ODBC for SQL Server driver:

```ini
[esm_msvtest1_ida_utf8_2017]
Driver=Easysoft ODBC-SQL Server
Description=Easysoft SQL Server ODBC driver
Server=ida
Port=1683
Database=msvtest1
Mars_Connection=No
Logging=No
LogFile=/tmp/odbc.log
#QuotedId=No
AnsiNPW=Yes
Language=
Version7=No
ClientLB=No
Failover_Partner=VarMaxAsLong=No
DisguiseWide=No
DisguiseLong=No
Trusted_Connection=No
Trusted_Domain=
IPv6=No
Client_CSet=UTF-8
Server_UCSet=UTF-16LE
```

See EasySoft documentation for more details about installation and data source configuration in ODBC files.
5. Set up the FGLPROFILE entries for database connections.
   a) Define the SQL Server database driver according to the database client used:

   ```
   dbi.database.dbname.driver = ( "dbmsnc" | "dbmesm" | "dbmftm" )
   ```

   b) The "source" parameter defines the name of the ODBC source.

   ```
   dbi.database.dbname.source = "test1"
   ```

   c) With the SNC driver, set the snc.widechar FGLPROFILE parameter to false, if database columns are defined with the CHAR/VARCHAR/TEXT SQL types, and your application is using a non-UTF-8, multibyte encoding (typically with BIG5). When using ISO8859-? or UTF-8, do not set this parameter: The expected char mode will be used, depending on the current application locale. See CHAR and VARCHAR data types on page 838 for more details.

   ```
   dbi.database.dbname.snc.widechar = false
   ```

   d) If required, define the serial emulation method to "trigseq", when the INSERT statements use all columns of the table, including the serial column. For more details, see SERIALIZ and BIGSERIAL data types on page 847.

   ```
   dbi.database.dbname.ifxemul.datatype.serial.emulation = "trigseq"
   ```

   e) If needed, define the login timeout with the following FGLPROFILE entry:

   ```
   dbi.database.stores.driver-code.logintime = 5
   ```

   f) If needed, define the number of rows to be fetched at once on the application side, for each single FETCH instruction:

   ```
   dbi.database.stores.driver-code.prefetch.rows = 50
   ```

   Note: The default is 10 rows. This is usually sufficient for regular interactive applications. Increase this parameter only in case of batch programs processing large result sets. The bigger this parameter is, the more memory is used by each program.

   g) If needed, add ODBC connection string parameters with the `datasource?options` notation, in the `source` parameter of the connection. You can for example define the SQL client application identifier for SQL Server.

   ```
   dbi.database.dbname.source = "test1?APP=myappid;"
   ```

   Note: The `source` parameter can also be defined at runtime in the database specification of CONNECT TO instruction.

Database concepts
Microsoft™ SQL Server related database concepts topics.

Database concepts
As in Informix®, an SQL SERVER engine can manage multiple database entities. When creating a database object like a table, Microsoft™ SQL SERVER allows you to use the same object name in different databases.

Tip: If you have several Informix database entities, migrating from the Informix database to another database it is a good opportunity to centralize all tables in a single database. To avoid conflicts with table names, use a prefix when needed.

Data storage concepts
When converting from Informix® to Microsoft™ SQL Server the aim is to try to preserve as much of the data storage information as possible in the process. The most important storage decisions made for Informix® database objects (like initial sizes and physical placement) can be applied to the SQL Server database.
Storage concepts are quite similar in Informix® and in Microsoft™ SQL Server, but the names are different. These table compares Informix® storage concepts to Microsoft™ SQL SERVER storage concepts:

Table 224: Physical units of storage

<table>
<thead>
<tr>
<th>Informix®</th>
<th>Microsoft™ SQL SERVER</th>
</tr>
</thead>
<tbody>
<tr>
<td>The largest unit of physical disk space is a &quot;chunk&quot;, which can be allocated either as a cooked file (I/O is controlled by the OS) or as raw device (= UNIX™ partition, I/O is controlled by the database engine). A &quot;dbspace&quot; uses at least one &quot;chunk&quot; for storage. You must add &quot;chunks&quot; to &quot;dbspaces&quot; in order to increase the size of the logical unit of storage. A &quot;page&quot; is the smallest physical unit of disk storage that the engine uses to read from and write to databases. A &quot;chunk&quot; contains a certain number of &quot;pages&quot;. The size of a &quot;page&quot; must be equal to the operating system's block size. An &quot;extent&quot; consists of a collection of continuous &quot;pages&quot; that the engine uses to allocate both initial and subsequent storage space for database tables. When creating a table, you can specify the first extent size and the size of future extents with the EXTENT SIZE and NEXT EXTENT options. For a single table, &quot;extents&quot; can be located in different &quot;chunks&quot; of the same &quot;dbspace&quot;.</td>
<td>SQL Server uses &quot;filegroups&quot;, based on Windows NT™ operating system files and therefore define the physical location of data. As in Informix®, SQL Server stores data in &quot;pages&quot; with a size fixed at 2Kb in V6.5 and 8Kb in V7 and later. An &quot;extent&quot; is a specific number of 8 contiguous pages, obtained in a single allocation. Extents are allocated in the filegroup used by the database.</td>
</tr>
</tbody>
</table>

Table 225: Logical units of storage

<table>
<thead>
<tr>
<th>Informix®</th>
<th>Microsoft™ SQL SERVER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A &quot;table&quot; is a logical unit of storage that contains rows of data values. A &quot;database&quot; is a logical unit of storage that contains table and index data. Each database also contains a system catalog that tracks information about database elements like tables, indexes, stored procedures, integrity constraints and user privileges. Database tables are created in a specific &quot;dbspace&quot;, which defines a logical place to store data. If no dbspace is given when creating the table, Informix® defaults to the current database dbspace. The total disk space allocated for a table is the &quot;tblspace&quot;, which includes &quot;pages&quot; allocated for data, indexes, blobs, tracking page usage within table extents.</td>
<td>Same concept as Informix®. Same concept as Informix®. When creating a &quot;database&quot;, you must specify which &quot;database devices&quot; (V6.5) or &quot;filegroup&quot; (V7) has to be used for physical storage. Database tables are created in a database based on &quot;database devices&quot; (V6.5) or a &quot;filegroup&quot; (V7), which defines the physical storage. No equivalent.</td>
</tr>
</tbody>
</table>
Table 226: Other concepts relating to storage

<table>
<thead>
<tr>
<th>Informix®</th>
<th>Microsoft™ SQL SERVER</th>
</tr>
</thead>
<tbody>
<tr>
<td>When initializing an Informix® engine, a &quot;root dbspace&quot; is created to store information about all databases, including storage information (chunks used, other dbspaces, etc.).</td>
<td>SQL Server uses the &quot;master&quot; database to hold system stored procedures, system messages, SQL Server logins, current activity information, configuration parameters of other databases.</td>
</tr>
<tr>
<td>The &quot;physical log&quot; is a set of continuous disk pages where the engine stores &quot;before-images&quot; of data that has been modified during processing.</td>
<td>Each database has its own &quot;transaction log&quot; that records all changes to the database. The &quot;transaction log&quot; is based on a &quot;database device&quot; (V6.5) or &quot;filegroup&quot; (V7) which is specified when creating the database.</td>
</tr>
<tr>
<td>The &quot;logical log&quot; is a set of &quot;logical-log files&quot; used to record logical operations during on-line processing. All transaction information is stored in the logical log files if a database has been created with transaction log.</td>
<td>SQL Server checks the &quot;transaction logs&quot; for automatic recovery.</td>
</tr>
<tr>
<td>Informix® combines &quot;physical log&quot; and &quot;logical log&quot; information when doing fast recovery. Saved &quot;logical logs&quot; can be used to restore a database from tape.</td>
<td></td>
</tr>
</tbody>
</table>

Concurrent management

Data consistency and concurrency concepts

- Data Consistency applies to situations when readers want to access data currently being modified by writers.
- Concurrent Data Access applies to situations when several writers are accessing the same data for modification.
- Locking Granularity defines the amount of data concerned when a lock is set (for example, row, page, table).

Informix®

Informix® uses a locking mechanism to handle data consistency and concurrency. When a process changes database information with UPDATE, INSERT or DELETE, an exclusive lock is set on the touched rows. The lock remains active until the end of the transaction. Statements performed outside a transaction are treated as a transaction containing a single operation and therefore release the locks immediately after execution. SELECT statements can set shared locks, depending on isolation level. In case of locking conflicts (for example, when two processes want to acquire an exclusive lock on the same row for modification, or when a writer is trying to modify data protected by a shared lock), the behavior of a process can be changed by setting the lock wait mode.

Control:

- Lock wait mode: SET LOCK MODE TO ...
- Isolation level: SET ISOLATION TO ...
- Locking granularity: CREATE TABLE ... LOCK MODE {PAGE|ROW}
- Explicit exclusive lock: SELECT ... FOR UPDATE

Defaults:

- The default isolation level is READ COMMITTED.
- The default lock wait mode is NOT WAIT.
- The default locking granularity is PAGE.

Microsoft™ SQL Server

As in Informix®, SQL Server uses locks to manage data consistency and concurrency. The database manager sets exclusive locks on the modified rows and shared locks or update locks when data is read, based on the isolation level. The locks are held until the end of the transaction. When multiple processes want to access the same data, the latest processes must wait until the first finishes its transaction or the lock timeout occurs. The locking strategy of
SQL Server is row locking with possible promotion to page or table locking. SQL Server dynamically determines the appropriate level at which to place locks for each Transact-SQL statement.

SQL Server supports snapshot isolation level, to force using a copy of the row when it is changed by a transaction. To turn this feature on, you must set the database property ALLOW_SNAPSHOT_ISOLATION ON. Setting the READ_COMMITTED_SNAPSHOT ON option allows access to versioned rows under the default READ COMMITTED isolation level (otherwise, snapshot isolation must be specified by every SQL Session).

Control:

- Lock wait mode: SET LOCK_TIMEOUT milliseconds (returns error 1222 on time out).
- Isolation level: SET TRANSACTION ISOLATION LEVEL ...
- Locking granularity: Row, Page, or Table level (Automatic - See Dynamic Locking).
- Explicit locking: SELECT ... FROM ... WITH (UPDLOCK) (See Locking Hints)

Defaults:

- The default isolation level is READ COMMITTED (readers cannot see uncommitted data).
- The default LOCK_TIMEOUT is -1 (indicates no timeout period, wait forever).

Solution

The SET ISOLATION TO ... in programs is converted to SET TRANSACTION ISOLATION LEVEL ... for SQL Server. The table shows the isolation level mappings applied by the database driver:

Table 227: Isolation level mappings done by the Microsoft™ SQL Server database driver

<table>
<thead>
<tr>
<th>SET ISOLATION instruction in program</th>
<th>Native SQL command</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET ISOLATION TO DIRTY READ</td>
<td>SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED</td>
</tr>
<tr>
<td>SET ISOLATION TO COMMITTED READ [READ COMMITTED] [RETAIN UPDATE LOCKS]</td>
<td>SET TRANSACTION ISOLATION LEVEL READ COMMITTED</td>
</tr>
<tr>
<td>SET ISOLATION TO CURSOR STABILITY</td>
<td>SET TRANSACTION ISOLATION LEVEL REPEATABLE READ</td>
</tr>
<tr>
<td>SET ISOLATION TO REPEATABLE READ</td>
<td>SET TRANSACTION ISOLATION LEVEL SERIALIZABLE</td>
</tr>
</tbody>
</table>

For portability, it is recommended that you work with Informix® in the read committed isolation level, to make processes wait for each other (lock mode wait) and to create tables with the "lock mode row" option.

When using SET LOCK_MODE ... in the programs, it will be converted to a SET LOCK_TIMEOUT instruction for SQL Server:

Table 228: SET LOCK MODE as handled by the Microsoft™ SQL Server database driver

<table>
<thead>
<tr>
<th>SET LOCK MODE instruction in program</th>
<th>Native SQL command</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET LOCK MODE TO WAIT</td>
<td>SET LOCK_TIMEOUT -1 (wait forever)</td>
</tr>
<tr>
<td>SET LOCK MODE TO WAIT seconds</td>
<td>SET LOCK_TIMEOUT (seconds*1000) (wait N milliseconds)</td>
</tr>
<tr>
<td>SET LOCK MODE TO NOT WAIT</td>
<td>SET LOCK_TIMEOUT 0 (do not wait)</td>
</tr>
</tbody>
</table>

See Informix® and SQL Server documentation for more details about data consistency, concurrency and locking mechanisms.
**Related concepts**

- **Concurrent data access** on page 599
  - Understanding concurrent data access and data consistency.

- **Optimistic locking** on page 624
  - Implementing optimistic locking to handle access concurrently to the same database records.

- **Cursors WITH HOLD** on page 631
  - Programming WITH HOLD cursors using SELECT with and without FOR UPDATE clause.

**Transactions handling**

**Informix**

With the Informix® native mode (non ANSI):

- Transactions blocks start with `BEGIN WORK` and terminate with `COMMIT WORK` or `ROLLBACK WORK`.
- Statements executed outside a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.

```sql
UPDATE tab1 SET ...   -- auto-committed
BEGIN WORK            -- start of TX block
UPDATE tab1 SET ...
UPDATE tab2 SET ...
...
COMMIT WORK           -- end of TX block
```

Informix® version 11.50 introduces savepoints:

```sql
SAVEPOINT name [UNIQUE]
ROLLBACK [WORK] TO SAVEPOINT name
RELEASE SAVEPOINT name
```

**Microsoft™ SQL Server**

Microsoft™ SQL Server supports named and nested transactions:

- Transactions are started with `BEGIN TRANSACTION [name]`.
- Transactions are validated with `COMMIT TRANSACTION [name]`.
- Transactions are canceled with `ROLLBACK TRANSACTION [name]`.
- Savepoints can be placed with `SAVE TRANSACTION name`.
- Transactions can be rolled back to a savepoint with `ROLLBACK TRANSACTION TO name`.
- Savepoints can not be released.
- Statements executed outside of a transaction are automatically committed (autocommit mode). This behavior can be changed with "SET IMPLICIT_TRANSACTION ON".
- DDL statements are not supported in transactions blocks.

**Solution**

Informix® transaction handling commands are automatically converted to Microsoft™ SQL Server instructions to start, validate or cancel transactions.

Regarding the transaction control instructions, the BDL applications do not have to be modified to work with Microsoft™ SQL Server.

**Important:** If you want to use savepoints, do not use the UNIQUE keyword in the savepoint declaration, always specify the savepoint name in `ROLLBACK TO SAVEPOINT`, and do not drop savepoints with `RELEASE SAVEPOINT`. 
**Related concepts**

**Database transactions** on page 600

Database transactions define a set of SQL instructions to be executed as a whole, or rolled back as a whole.

**Database users**

**Informix®**

Until version 11.70.xC2, Informix® database users must be created at the operating system level and must be members of the 'informix' group.

Starting with 11.70.xC2, Informix® supports database-only users with the `CREATE USER` instruction, as in most other db servers.

Any database user must have sufficient privileges to connect and use resources of the database; user rights are defined with the `GRANT` command.

**Microsoft™ SQL Server**

Before a user can access an SQL Server database, the system administrator (SA) must add the user’s login to the SQL Server Login list and add a user name for that database. The user name is a name that is assigned to a login ID for the purpose of allowing that user to access a specified database. Database users are members of a user group; the default group is 'public'.

Microsoft™ SQL Server offers two authentication modes:

1. The **SQL Server authentication mode**, which requires a login name and a password
2. The **Windows™ authentication mode**, which uses the security mechanisms within Windows™ when validating login connections. With this mode, user do not have to enter a login ID and password - their login information is taken directly from the network connection.

**Solution**

Both SQL Server and Windows™ authentication methods can be used to allow BDL program users to connect to Microsoft™ SQL Server and access a specific database.

If you don't specify the `USER/USING` clause in the `CONNECT TO` instruction, operating system authentication takes place.

See SQL Server documentation for more details on database logins and users.

**Related concepts**

**Database users and security** on page 608

Properly identifying database users allows to use database security and audit features.

**Setting privileges**

**Informix®**

Informix® users must have at least the `CONNECT` privilege to access the database:

```
GRANT CONNECT TO username
```

Application administration users need the `RESOURCE` privilege to create tables:

```
GRANT RESOURCE TO username
```

Since version 7.20, Informix® supports database roles:

```
GRANT rolename TO username
```
**Microsoft™ SQL Server**

Microsoft™ SQL Server supports *user groups*, to grant or revoke permissions to more than one user at the same time. See SQL Server documentation for more details.

**Solution**

Informix® and Microsoft™ SQL Server user privileges management are quite similar.

**Data dictionary**

Microsoft™ SQL Server related data dictionary topics.

**Data type conversion table: Informix to SQL Server**

**Table 229: Data type conversion table (Informix to SQL Server)**

<table>
<thead>
<tr>
<th>Informix® data types</th>
<th>Microsoft™ SQL Server data types</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR(n)</td>
<td>CHAR(n) if SBCS, NCHAR(n) if UTF-8</td>
</tr>
<tr>
<td>VARCHAR(n[,m])</td>
<td>VARCHAR(n) if SBCS, NVARCHAR(n) if UTF-8</td>
</tr>
<tr>
<td>NVARCHAR(n[,m])</td>
<td>NVARCHAR(n) (UNICODE, max is 4000 chars)</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>BIT</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>BIGINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>INT8</td>
<td>BIGINT</td>
</tr>
<tr>
<td>SERIAL[(start)]</td>
<td>INTEGER (see note 1)</td>
</tr>
<tr>
<td>BIGSERIAL[(start)]</td>
<td>BIGINT (see note 1)</td>
</tr>
<tr>
<td>SERIAL8[(start)]</td>
<td>BIGINT (see note 1)</td>
</tr>
<tr>
<td>DOUBLE PRECISION / FLOAT[(n)]</td>
<td>FLOAT(n)</td>
</tr>
<tr>
<td>REAL / SMALLFLOAT</td>
<td>REAL</td>
</tr>
<tr>
<td>DECIMAL(p,s)</td>
<td>DECIMAL(p,s)</td>
</tr>
<tr>
<td>DECIMAL(p) with p&lt;=19</td>
<td>DECIMAL(2*p,p)</td>
</tr>
<tr>
<td>DECIMAL(p) with p&gt;19</td>
<td>N/A</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL(32,16)</td>
</tr>
<tr>
<td>MONEY(p,s)</td>
<td>DECIMAL(p,s)</td>
</tr>
<tr>
<td>MONEY(p)</td>
<td>DECIMAL(p,2)</td>
</tr>
<tr>
<td>MONEY</td>
<td>DECIMAL(16,2)</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME HOUR TO MINUTE</td>
<td>TIME(0)</td>
</tr>
<tr>
<td>DATETIME HOUR TO SECOND</td>
<td>TIME(0)</td>
</tr>
<tr>
<td>DATETIME HOUR TO FRACTION(n)</td>
<td>TIME(n)</td>
</tr>
<tr>
<td>DATETIME YEAR TO MONTH</td>
<td>DATETIME2(0)</td>
</tr>
<tr>
<td><strong>Informix® data types</strong></td>
<td><strong>Microsoft™ SQL Server data types</strong></td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------------------------------</td>
</tr>
<tr>
<td>DATETIME YEAR TO DAY</td>
<td>DATETIME2 (0)</td>
</tr>
<tr>
<td>DATETIME YEAR TO HOUR</td>
<td>DATETIME2 (0)</td>
</tr>
<tr>
<td>DATETIME YEAR TO MINUTE</td>
<td>DATETIME2 (0)</td>
</tr>
<tr>
<td>DATETIME YEAR TO SECOND</td>
<td>DATETIME2 (0)</td>
</tr>
<tr>
<td>DATETIME YEAR TO FRACTION(n)</td>
<td>DATETIME2 (n)</td>
</tr>
<tr>
<td>INTERVAL q1 TO q2</td>
<td>CHAR (50)</td>
</tr>
<tr>
<td>TEXT</td>
<td>VARCHAR (MAX)</td>
</tr>
<tr>
<td>BYTE</td>
<td>VARBINARY (MAX)</td>
</tr>
</tbody>
</table>

Notes:
1. For more details about serial emulation, see SERIAL and BIGSERIAL data types on page 847.

**BOOLEAN data type**

**Informix®**

Informix® supports the BOOLEAN data type, which can store ‘t’ or ‘f’ values.

Genero BDL implements the BOOLEAN data type in a different way: A BOOLEAN variable stores integer values 1 or 0 (for TRUE or FALSE). This type is designed to hold the result of a boolean expression.

**Microsoft™ SQL Server**

Microsoft™ SQL Server supports the BIT data type to store boolean values.

**Solution**

The SQL Server database interfaces converts BOOLEAN type to BIT columns and stores 1 or 0 values in the column.

The BOOLEAN type translation can be controlled with the following FGLPROFILE entry:

```plaintext
  dbi.database.dsnname.ifxemul.datatype.boolean = { true | false }
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

Using portable data types on page 614

Only a limited set of data types are really portable across several database engines.

**CHAR and VARCHAR data types**

**Informix®**

Informix® supports the following character data types:

**Table 230: Informix® character data types**

<table>
<thead>
<tr>
<th><strong>Informix® data type</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR (n)</td>
<td>SBCS and MBCS character data (max is 32767 bytes)</td>
</tr>
<tr>
<td>VARCHAR (n[,m])</td>
<td>SBCS and MBCS character data (max is 255 bytes)</td>
</tr>
<tr>
<td>NCHAR (n)</td>
<td>Same as CHAR, with specific collation order</td>
</tr>
</tbody>
</table>
With Informix®, both CHAR/VARCHAR and NCHAR/NVARCHAR data types can be used to store single-byte or multibyte encoded character strings. The only difference between CHAR/VARCHAR and NCHAR/NVARCHAR is in how they use sorting: N [VAR] CHAR types use the collation order, while [VAR] CHAR types use the byte order.

The character set used to store strings in CHAR/VARCHAR/NCHAR/NVARCHAR columns is defined by the DB_LOCALE environment variable.

The character set used by applications is defined by the CLIENT_LOCALE environment variable.

Informix® uses Byte Length Semantics (the size N that you specify in [VAR] CHAR(N) is expressed in bytes, not characters as in some other databases)

**Microsoft™ SQL Server**

Microsoft™ SQL Server supports following data types to store character data:

**Table 231: Microsoft™ SQL Server character data types**

<table>
<thead>
<tr>
<th>Microsoft™ SQL Server data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR (n)</td>
<td>SBCS or MBCS encoding using the database collation (starting with SQL Server 2019 it can be UTF-8), where n is specified in bytes (maximum storage size is 8000 bytes)</td>
</tr>
<tr>
<td>VARCHAR (n)</td>
<td>SBCS or MBCS encoding using the database collation (starting with SQL Server 2019 it can be UTF-8), where n is specified in bytes (maximum storage size is 8000 bytes)</td>
</tr>
<tr>
<td>VARCHAR (MAX)</td>
<td>SBCS or MBCS encoding using the database collation (starting with SQL Server 2019 it can be UTF-8), to store large text data (maximum storage size is 2^31-1 bytes)</td>
</tr>
<tr>
<td>NCHAR (n)</td>
<td>UCS-2/UTF-16 encoding, where n is specified in byte-pairs (maximum storage size is 4000 byte-pairs)</td>
</tr>
<tr>
<td>NVARCHAR (n)</td>
<td>UCS-2/UTF-16 encoding, where n is specified in byte-pairs (maximum storage size is 4000 byte-pairs)</td>
</tr>
<tr>
<td>NVARCHAR (MAX)</td>
<td>UCS-2/UTF-16 encoding, to store large text data (maximum storage size is 2^30-1 byte-pairs)</td>
</tr>
</tbody>
</table>

**Note:** To store large text data (LOBs), Microsoft™ SQL Server (2005) provides the VARCHAR (MAX) / NVARCHAR (MAX) type as a replacement for the old TEXT/NTEXT types. See TEXT and BYTE (LOB) types on page 854 for more details.

The use of NCHAR, NVARCHAR character types is the same as CHAR, VARCHAR respectively, except:

- For NCHAR / NVARCHAR, the character encoding is UCS-2 or UTF-16, depending if the database collation is using the SC option.
- The length N in N [VAR] CHAR (N) defines a number of byte-pairs, not bytes. For UCS-2 this corresponds to a number of characters. But with UTF-16 there can be surrogate pairs using 4 bytes (2 byte-pairs).
- The maximum size of NCHAR (N) and NVARCHAR (N) column is 4000 byte-pairs, compared to 8000 bytes for CHAR/VARCHAR using a single-byte character set.
- Unicode string literals are specified with a leading N. For example: N'###'
- The LIKE statement behaves differently with CHAR and NCHAR columns when using the N prefix before the search pattern.
Note that SQL Server uses Byte Length Semantics to define the size of CHAR/VARCHAR columns, while NCHAR and NVARCHAR sizes are expressed in byte-pair units, which for most cases (UCS-2) corresponds to a number of characters (but this is not true for UTF-16 characters encoded in surrogate pairs).

SQL Server defines the character encoding for CHAR and VARCHAR columns with the database collation. The database collation can be specified when creating a new database. Character strings are always stored in the UCS-2 or UTF-16 encoding for NCHAR/NVARCHAR columns. UTF-16 is used when the DB collation has the SC option.

Automatic charset conversion is supported by SQL Server between the client application and the server. The client charset is defined by the Windows® operating system, in the language settings for non-Unicode applications.

**Solution**

Depending on the character set used by your application, you must either use CHAR/VARCHAR or NCHAR/NVARCHAR columns with SQL Server. If the charset is single-byte, you can use CHAR/VARCHAR columns. If the charset set is multibyte or Unicode (i.e. UTF-8), you must use NCHAR/NVARCHAR columns in SQL Server.

Make sure that the regional language settings for non-Unicode applications corresponds to the locale used by Genero programs. For more details about program locale settings, see Localization.

Check that your database tables does not use CHAR or VARCHAR types with a length exceeding the SQL Server character types limits.

When using a multibyte character set (such as UTF-8), define database columns as NCHAR and NVARCHAR, with the size in byte-pair units, and use character length semantics in BDL programs with FGL_LENGTH_SEMANTICS=CHAR. When using a single-byte character set in the programs (like ISO-8859-15), the column types can be CHAR and VARCHAR, and the size in bytes corresponds to the number of characters.

By default, in DDL statements executed by programs (such as CREATE TABLE), the CHAR, VARCHAR and LVARCHAR type names are left untouched. To force the ODI drivers to replace CHAR/VARCHAR/LVARCHAR by NCHAR/NVARCHAR type names when the locals is multibyte, set the `dbi.database.dsname.ifxemul.nationalchars` FGLPROFILE entry to true (default is false):

```
  dbi.database.dsname.ifxemul.nationalchars = true
```

**Note:**

1. With SQL Server ODI drivers, `dbi.database.dsname.ifxemul.nationalchars` will only take effect, if the current application locale is multibyte (typically, UTF-8)
2. The `dbi.database.dsname.ifxemul.nationalchars` parameter is ignored, if the switch corresponding to the character type name (dbi.database.dsname.datatype.{char|varchar|text}) is set to false.
3. With the SNC ODI driver, to have `dbi.database.dsname.ifxemul.nationalchars` take effect, the wide-char mode must be enabled (this is the default with multibyte character sets).

When extracting a database schema from a SQL Server database, the `fgldbsch schema extractor` uses the size of the column in byte-pair units, not the byte length. If you have created an NCHAR(10) column in an SQL Server database, the .sch file will get a size of 10, that will be interpreted as a number of bytes or characters, depending on FGL_LENGTH_SEMANTICS.

Do not forget to properly define the database client character set, which must correspond to the runtime system character set.

The CHAR/VARCHAR type translation can be controlled with the following FGLPROFILE entries:

```
  dbi.database.dsname.ifxemul.datatype.char = {true | false}
  dbi.database.dsname.ifxemul.datatype.varchar = {true | false}
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.
Using the SNC driver

The SNC driver can work in char or in wide-char mode:

The char mode must be used with applications defining database columns with CHAR/VARCHAR/TEXT types. It is not mandatory (i.e. the wide-char mode could be used), but it appears that SQL Server performs better, when char bindings are used for CHAR/VARCHAR/TEXT columns. In char mode, the SNC driver binds SQL parameters (SQLBindParameter) with the SQL_CHAR/SQL_VARCHAR ODBC SQL types. When defining CHAR(N) / VARCHAR(N) columns in SQL Server, you specify N as a number of bytes. Therefore, it is recommended that you use byte length semantics in Genero programs, with FGL_LENGTH_SEMANTICS=BYTE (this is the default).

The wide-char mode must be used for applications defining database columns with NCHAR/NVARCHAR/NTEXT types. These SQL types are used to store UNICODE data. In such case, the runtime system must use a UTF-8 locale, with character length semantics (FGL_LENGTH_SEMANTICS=CHAR). In wide-char mode, the SNC driver binds SQL parameters (SQLBindParameter) with the SQL_WCHAR/SQL_WVARCHAR ODBC SQL types. Furthermore, all string literals of an SQL statement are automatically changed to get the N prefix. Thus, you don't need to add the N prefix by hand in all of your programs. This solution makes your Genero code portable to other databases. When defining NCHAR(N) / NVARCHAR(N) columns in SQL Server, you specify N as a number of byte-pairs that corresponds to a number of characters in UCS-2. Therefore, it is recommended you use char length semantics in Genero programs, with FGL_LENGTH_SEMANTICS=CHAR.

By default, the SNC database driver selects the expected char or wide-char mode, depending on the current application locale, assuming that the database column types fit to the application locale:

- If the application locale defines a single-byte encoding (such as ISO8859-15), the driver assumes that the database columns are defined with CHAR/VARCHAR/TEXT types to store single-byte characters, and the SNC driver will use SQL_[VAR]CHAR.
- If the application locale defines a multibyte encoding (such as UTF-8 or BIG5), the driver assumes that the database columns are defined with NCHAR/NVARCHAR/NTEXT types to store UNICODE characters, and the SNC driver will use SQL_W[VAR]CHAR.

The char / wide-char modes can be forced with the following FGLPROFILE entry:

```
dbi.database.dsname.snc.widechar= { true | false }
```

**Note:** Set the dbi.database.dsname.snc.widechar to false, only if you are using a multibyte encoding such as BIG5, with CHAR/VARCHAR/TEXT column types in the database.

Using the ESM driver

When using the ESM (Easysoft) database driver, SQL Statements are prepared with SQLPrepare(), by using the current character set. Easysoft makes the necessary charset conversions from the client charset to UCS-2 or UTF-16 for the server. ODBC SQL parameters with character string data are bound (SQLBindParameter) with the C type SQL_C_CHAR and with the SQL type SQL_W[VAR]CHAR (=UNICODE) or with SQL_[VAR]CHAR, based on the current locale. The SQL_W[VAR]CHAR type is used if the current locale uses a multibyte encoding. When using a single-byte encoding, parameters are bound with the SQL_[VAR]CHAR type. String literals get the N prefix only if the current locale defines a multibyte encoding. String literals are not touched, if the locale uses a single-byte character set. As a result, the necessary character set conversion is controlled by Easysoft and is optimized for SQL Server when using a single-byte character set.

**Important:** Define the correct client character set in Easysoft configuration files. In the odbc.ini data source definition, the Easysoft client character set is specified with the "Client_CSet" parameter, and the server character set is defined by "Server_CSet" or "Server_UCSet" parameters. For example, to cover all UNICODE characters, define:

```
Client_CSet   = UTF-8
Server_UCSet  = UTF-16LE
```
Using the FTM driver

With the FTM (FreeTDS) database driver, SQL Statements are prepared with SQLPrepare(), by using the current character set. FreeTDS makes the necessary charset conversions from the client charset to UCS-2/UTF-16 before sending the SQL text to the server. ODBC SQL parameters with character string data are bound (SQLBindParameter) with the C type SQL_C_CHAR and with the SQL type SQL_W[VAR]CHAR (=UNICODE) or with SQL_[VAR]CHAR, based on the current locale. The SQL_W[VAR]CHAR type is used if the current locale uses a multibyte encoding. When using a single-byte encoding, parameters are bound with the SQL_[VAR]CHAR type. String literals get the N prefix only if the current locale defines a multibyte encoding. String literals are not touched, if the locale uses a single-byte character set. As a result, the necessary character set conversion is controlled by FreeTDS and is optimized for SQL Server when using a single-byte character set.

Important: Define the correct client character set in FreeTDS configuration files. The FreeTDS client character set is defined with "ClientCharset" parameter in odbc.ini.

Related concepts

Temporary tables on page 861
CHAR and VARCHAR types on page 616
Using the CHAR and VARCHAR data types with different databases.

Numeric data types

Informix®

Informix® supports several data types to store numbers:

Table 232: Informix® numeric data types

<table>
<thead>
<tr>
<th>Informix® data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>16 bit signed integer</td>
</tr>
<tr>
<td>INTEGER</td>
<td>32 bit signed integer</td>
</tr>
<tr>
<td>BIGINT</td>
<td>64 bit signed integer</td>
</tr>
<tr>
<td>INT8</td>
<td>64 bit signed integer (replaced by BIGINT)</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>Equivalent to DECIMAL(16)</td>
</tr>
<tr>
<td>DECIMAL(p)</td>
<td>Floating-point decimal number (max precision is 32)</td>
</tr>
<tr>
<td>DECIMAL(p, s)</td>
<td>Fixed-point decimal number (max precision is 32)</td>
</tr>
<tr>
<td>MONEY</td>
<td>Equivalent to DECIMAL(16, 2)</td>
</tr>
<tr>
<td>MONEY(p)</td>
<td>Equivalent to DECIMAL(p, 2) (max precision is 32)</td>
</tr>
<tr>
<td>MONEY(p, s)</td>
<td>Equivalent to DECIMAL(p, s) (max precision is 32)</td>
</tr>
<tr>
<td>REAL / SMALLFLOAT</td>
<td>32-bit floating point decimal (C float)</td>
</tr>
<tr>
<td>DOUBLE PRECISION / FLOAT[(n)]</td>
<td>64-bit floating point decimal (C double)</td>
</tr>
</tbody>
</table>

Microsoft™ SQL Server

Microsoft™ SQL Server supports the following numeric data types:

Table 233: Microsoft™ SQL Server numeric data types

<table>
<thead>
<tr>
<th>Microsoft™ SQL Server data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>16 bit signed integer</td>
</tr>
</tbody>
</table>
### Microsoft™ SQL Server data type Description

<table>
<thead>
<tr>
<th><strong>Microsoft™ SQL Server data type</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER 32 bit signed integer</td>
<td></td>
</tr>
<tr>
<td>BIGINT 64 bit signed integer</td>
<td></td>
</tr>
<tr>
<td>DECIMAL (p, s) Fixed point decimal.</td>
<td></td>
</tr>
<tr>
<td>SMALLMONEY 32-bit floating point decimal with currency</td>
<td></td>
</tr>
<tr>
<td>MONEY 64-bit floating point decimal with currency</td>
<td></td>
</tr>
<tr>
<td>REAL 32-bit floating point decimal (C float)</td>
<td></td>
</tr>
<tr>
<td>FLOAT[(n)] (DOUBLE) 64-bit floating point decimal (C double)</td>
<td></td>
</tr>
</tbody>
</table>

**Notes about SQL Server DECIMAL type:**
- Without any decimal storage specification, the precision defaults to 18 and the scale defaults to zero:
  - DECIMAL in SQL Server = DECIMAL(18, 0) in Genero BDL.
  - DECIMAL (p) in SQL Server = DECIMAL (p, 0) in Genero BDL.
- The maximum precision is 38.

**Notes about the SQL Server MONEY and SMALLMONEY types:**
- SQL Server provides the MONEY and SMALLMONEY data types, but the currency symbol handling is quite different. Therefore, it is recommended to implement Informix® MONEY columns as DECIMAL columns in SQL Server.

**Solution**

Use the following conversion rules to map Informix® numeric types to SQL Server numeric types:

### Table 234: Informix® data types and Microsoft™ SQL Server equivalents

<table>
<thead>
<tr>
<th>Informix®</th>
<th>Microsoft™ SQL SERVER</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INT / INTEGER</td>
</tr>
<tr>
<td>BIGINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>INT8</td>
<td>BIGINT</td>
</tr>
<tr>
<td>DECIMAL (p, s)</td>
<td>DECIMAL (p, s)</td>
</tr>
<tr>
<td>DECIMAL (p&lt;=19)</td>
<td>DECIMAL (2*p, p)</td>
</tr>
<tr>
<td>DECIMAL (p&gt;19)</td>
<td>N/A</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL (32,16)</td>
</tr>
<tr>
<td>MONEY (p, s)</td>
<td>DECIMAL (p, s)</td>
</tr>
<tr>
<td>MONEY (p)</td>
<td>DECIMAL (p, 2)</td>
</tr>
<tr>
<td>MONEY</td>
<td>DECIMAL (16,2)</td>
</tr>
<tr>
<td>REAL / SMALLFLOAT</td>
<td>REAL</td>
</tr>
<tr>
<td>FLOAT[(n)] / DOUBLE PRECISION</td>
<td>FLOAT (n) (Where n must be from 1 to 15)</td>
</tr>
</tbody>
</table>

When creating tables from BDL programs, the database interface automatically converts Informix® numeric data types to corresponding Microsoft™ SQL Server data types. In database creation scripts, apply the conversion rules as described in the above table.
**Important:** There is no SQL Server equivalent for the Informix® DECIMAL\(\text{(p)}\) floating point decimal (i.e. without a scale). If your application is using such data types, you must review the database schema in order to use SQL Server compatible types. To workaround the SQL Server limitation, the SQL Server database drivers convert DECIMAL\(\text{(p)}\) types to a DECIMAL\(\text{(2*p, p)}\), to store all possible numbers an Informix® DECIMAL\(\text{(p)}\) can store. However, the original Informix® precision cannot exceed 19, since SQL Server maximum DECIMAL precision is 38 \((2*19)\). If the original precision is bigger than 19, a CREATE TABLE statement executed from a Genero program will fail with an SQL Server error 2750.

The numeric types translation can be controlled with the following FGLPROFILE entries:

```plaintext
<table>
<thead>
<tr>
<th>Entry</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbi.database.dsnname.ifxemul.datatype.smallint</td>
<td>true</td>
</tr>
<tr>
<td>dbi.database.dsnname.ifxemul.datatype.integer</td>
<td>true</td>
</tr>
<tr>
<td>dbi.database.dsnname.ifxemul.datatype.bigint</td>
<td>false</td>
</tr>
<tr>
<td>dbi.database.dsnname.ifxemul.datatype.int8</td>
<td>false</td>
</tr>
<tr>
<td>dbi.database.dsnname.ifxemul.datatype.decimal</td>
<td>false</td>
</tr>
<tr>
<td>dbi.database.dsnname.ifxemul.datatype.money</td>
<td>false</td>
</tr>
<tr>
<td>dbi.database.dsnname.ifxemul.datatype.smallfloat</td>
<td>false</td>
</tr>
</tbody>
</table>
```

For more details see [IBM Informix emulation parameters in FGLPROFILE](#) on page 668.

**Related concepts**

- **Using portable data types** on page 614

Only a limited set of data types are really portable across several database engines.

### DATE and DATETIME data types

**Informix®**

Informix® provides two data types to store date and time information:

- **DATE** = for year, month and day storage.
- **DATETIME** = for year to fraction \((1-5)\) storage.

The **DATE** type is stored as an **INTEGER** with the number of days since 1899/12/31.

The **DATETIME** type can be defined with various time units, by specifying a start and end qualifier. For example, you can define a datetime to store an hour-to-second time value with **DATETIME HOUR TO SECOND**.

The values of Informix® **DATETIME** can be represented with a character string literal, or as **DATETIME()** literals:

```plaintext
'2017-12-24 15:45:12.345' -- a DATETIME YEAR TO FRACTION(3)
'15:45'   -- a DATETIME HOUR TO MINUTE
DATETIME(2017-12-24 12:45) YEAR TO MINUTE
DATETIME(12:45:56.333) HOUR TO FRACTION(3)
```

Informix® is able to convert quoted strings to **DATE** / **DATETIME** data, if the string contains matching environment parameters. The string to date conversion rules for **DATE** is defined by the DBDATE environment variable. The string to datetime format for **DATETIME** is defined by the GL_DATETIME environment variable.

**Note:** Within Genero programs, the string representation for **DATETIME** values is always ISO (YYYY-MM-DD hh:mm:ss.fffffff)

Informix® supports date arithmetic on **DATE** and **DATETIME** values. The result of an arithmetic expression involving dates/times is an **INTEGER** number of days when only **DATE** values are used, and an **INTERVAL** value if a **DATETIME** is used in the expression.

Informix® automatically converts an **INTEGER** to a **DATE** when the integer is used to set a value of a date column.

Informix® provides the `CURRENT \[ q1 TO q2 \]` operator, to get the system date/time on the server where the current database is located. When no qualifiers are specified, `CURRENT` returns a **DATETIME YEAR TO FRACTION(3)**. Informix also supports the **SYSDATE** operator, which returns the current system time as a **DATETIME YEAR TO FRACTION(5)**.
Note: The USEOSTIME configuration parameter must be set to 1 in order to get the subsecond precision in CURRENT and SYSDATE operators. See Informix documentation for more details.

Microsoft™ SQL Server

Microsoft™ SQL Server (2008+) provides the following data type to store date and time data:

Table 235: Microsoft™ SQL Server date/time data types

<table>
<thead>
<tr>
<th>Microsoft™ SQL Server data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>for year, month, day storage</td>
</tr>
<tr>
<td>DATETIME</td>
<td>for year, month, day, hour, min, second, fraction(3) storage (from January 1, 1753 through December 31, 9999). Values are rounded to increments of .000, .003, or .007 seconds</td>
</tr>
<tr>
<td>SMALLDATETIME</td>
<td>for year, month, day, hour, minutes storage (from January 1, 1900, through June 6, 2079). Values with 29.998 seconds or lower are rounded down to the nearest minute; values with 29.999 seconds or higher are rounded up to the nearest minute</td>
</tr>
<tr>
<td>TIME (n)</td>
<td>for hour, minute, second and fraction(7) storage. Where n defines the precision of fractional seconds</td>
</tr>
<tr>
<td>DATETIME2 (n)</td>
<td>for year, month, day, hour, minute, second and fraction(7) storage. Where n defines the precision of fractional seconds</td>
</tr>
<tr>
<td>DATETIMEOFFSET (n)</td>
<td>for year, month, day, hour, minute, second, fraction(7) and time zone information storage. Where n defines the precision of fractional seconds</td>
</tr>
</tbody>
</table>

Like Informix®, Microsoft™ SQL Server can convert quoted strings to DATETIME data. The CONVERT() SQL function allows you to convert strings to dates.

Microsoft™ SQL Server does not allow direct arithmetic operations on datetimes; the date handling SQL functions must be used instead (DATEADD and DATEDIFF).

SQL Server provides equivalent functions for Informix® YEAR(), MONTH() and DAY(). Take care with the DAY() function on SQL Server because it begins from January 1, 1900 while Informix® begins from December 31, 1899.

Table 236: Select first day example (Informix® vs. Microsoft™ SQL Server)

<table>
<thead>
<tr>
<th>Informix®</th>
<th>Microsoft™ SQL SERVER</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT day(0), month(0), year(0) FROM systables WHERE tabid=1;</td>
<td>SELECT day(0), month(0), year(0)</td>
</tr>
<tr>
<td>31 12 1899</td>
<td>----------- ----------- 1 1 1900</td>
</tr>
</tbody>
</table>

The SQL Server equivalent for Informix® WEEKDAY() is the DATEPART(dw, date-value) function.

The weekday date part depends on the value set by SET DATEFIRST n, which sets the first day of the week (1=Monday ... 7=Sunday (default)).

Solution

Use the following conversion rules to map Informix® date/time types to Microsoft™ SQL Server date/time types:
Table 237: Informix® data types and Microsoft™ SQL Server equivalents

<table>
<thead>
<tr>
<th>Informix® data type</th>
<th>Microsoft™ SQL Server data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME HOUR TO MINUTE</td>
<td>TIME(0)</td>
</tr>
<tr>
<td>DATETIME HOUR TO SECOND</td>
<td>TIME(0)</td>
</tr>
<tr>
<td>DATETIME HOUR TO FRACTION(n)</td>
<td>TIME(n)</td>
</tr>
<tr>
<td>DATETIME YEAR TO MONTH</td>
<td>DATETIME2(0)</td>
</tr>
<tr>
<td>DATETIME YEAR TO DAY</td>
<td>DATETIME2(0)</td>
</tr>
<tr>
<td>DATETIME YEAR TO HOUR</td>
<td>DATETIME2(0)</td>
</tr>
<tr>
<td>DATETIME YEAR TO MINUTE</td>
<td>DATETIME2(0)</td>
</tr>
<tr>
<td>DATETIME YEAR TO SECOND</td>
<td>DATETIME2(0)</td>
</tr>
<tr>
<td>DATETIME YEAR TO FRACTION(n)</td>
<td>DATETIME2(n)</td>
</tr>
</tbody>
</table>

The DATE and DATETIME types translation can be controlled with the following FGLPROFILE entries:

```ini
[dbi.database.
 dsname.ifxemul.datatype.date] = {true | false}
[dbi.database.
 dsname.ifxemul.datatype.datetime] = {true | false}
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

SQL Server and Informix® DATE data type are equivalent and store year, month, day values.

SQL Server TIME(n) data type can be used to store Informix® DATETIME HOUR TO FRACTION(n), DATETIME HOUR TO SECOND, DATETIME HOUR TO MINUTE values, and any other DATETIME type with qualifiers HOUR, MINUTE, SECOND and FRACTION(n). Missing time parts default to 00:00:00.0. For example, when using a DATETIME MINUTE TO FRACTION(3) with the value of "45:23.999", the SQL Server TIME(3) value will be "00:45:23.999".

Informix® DATETIME values with any qualifiers from YEAR to FRACTION(5) can be stored in SQL Server DATETIME2(n) columns. Missing date or time parts default to 1900-01-01 00:00:00.0. For example, when using a DATETIME DAY TO MINUTE with the value of "23 11:45", the SQL Server DATETIME(0) value will be "1900-01-23 11:45:00".

Important:
- When fetching a TIME or DATETIME2 with a precision that is greater than 5 (the DATETIME precision limit), the database interface will allocate a buffer of VARCHAR(16) for the TIME and VARCHAR(27) for the DATETIME2 column. As a result, you can fetch such data into a CHAR or VARCHAR variable.
- Review the program logic if you are using the Informix® WEEKDAY() function because SQL Server uses a different basis for the days numbers (Monday = 1).
- Use the SQL Server's GETDATE() function to get the system current date.

Related concepts
- Date/time literals in SQL statements on page 635
- Good practices for date and time handling in SQL.

INTERVAL data type

Informix®

Informix® provides the INTERVAL data type to store a value that represents a span of time.

INTERVAL types are divided into two classes:
• *year-month* intervals. For example: INTERVAL YEAR(5) TO MONTH
• *day-time* intervals. For example: INTERVAL DAY(9) TO SECOND

INTERVAL columns can be defined with various time units, by specifying a start and end qualifier. For example, you can define an interval to store a number of hours and minutes with INTERVAL HOUR(n) TO MINUTE, where n defines the maximum number of digits for the hours unit.

The values of Informix® INTERVAL can be represented with a character string literal, or as INTERVAL() literals:

```
'-9834 15:45:12.345' -- an INTERVAL DAY(6) TO FRACTION(3)
'7623-11'   -- an INTERVAL YEAR(9) TO MONTH
INTERVAL(18734:45) HOUR(5) TO MINUTE
INTERVAL(-7634-11) YEAR(5) TO MONTH
```

**Microsoft™ SQL Server**

Microsoft™ SQL Server does not provide a data type corresponding the Informix® INTERVAL data type.

**Solution**

The INTERVAL data type and values are converted CHAR(50) column with IBM® DB2®.

INTERVAL values can be stored and retrieved from the database. However, since IBM® DB2® does not support a native interval type, arithmetics cannot be performed on the database side in SQL statements.

The INTERVAL types translation can be controlled with the following FGLPROFILE entry:

```
dbi.database.dsnname.ifxemul.datatype.interval = \true \false
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

Using portable data types on page 614

Only a limited set of data types are really portable across several database engines.

**SERIAL and BIGSERIAL data types**

**Informix®**

Informix® supports the SERIAL, BIGSERIAL data types to produce automatic integer sequences:

• SERIAL can produce 32 bit integers (INTEGER)
• BIGSERIAL can produced 64 bit integers (BIGINT)
• SERIAL8 is a synonym for BIGSERIAL

Steps to use serials with Informix®:

1. Create the table with a column using SERIAL, or BIGSERIAL.
2. To generate a new serial, no value or a zero value is specified in the INSERT statement:

```
INSERT INTO tab1 ( c ) VALUES ( 'aa' )
INSERT INTO tab1 ( k, c ) VALUES ( 0, 'aa' )
```

3. After INSERT, the new value of a SERIAL column is provided in SQLCA.SQLERRD[2], while the new value of a BIGSERIAL value must be fetched with a SELECT dbinfo('bigserial') query.

Informix® allows you to insert rows with a value different from zero for a serial column. Using an explicit value will automatically increment the internal serial counter, to avoid conflicts with future INSERT statements that are using a zero value:

```
CREATE TABLE tab ( k SERIAL); -- internal counter = 0
INSERT INTO tab VALUES ( 0 ); -- internal counter = 1
```
Microsoft™ SQL Server

Microsoft™ SQL Server IDENTITY columns:

- When creating a table, the IDENTITY keyword must be specified after the column data type:

```
CREATE TABLE tab1 ( k INTEGER IDENTITY, c NVARCHAR(10) )
```

- You can specify a start value and an increment with "identity(start, incr)"

```
CREATE TABLE tab1 ( k INTEGER IDENTITY(100,2), ...)
```

- A new number is automatically created when inserting a new row:

```
INSERT INTO tab1 ( c ) VALUES ( 'aaa' )
```

- To get the last generated number, Microsoft™ SQL SERVER provides the `SCOPE_IDENTITY()` function. The `@@IDENTITY` global T-SQL variable is not recommended, as it is scope-less.

- To put a specific value into an IDENTITY column, the `SET` command must be used:

```
SET IDENTITY_INSERT tab1 ON
INSERT INTO tab1 ( k, c ) VALUES ( 100, 'aaa' )
SET IDENTITY_INSERT tab1 OFF
```

Informix® SERIALs and SQL Server IDENTITY columns are quite similar; the main difference is that SQL Server does not allow you to use the zero value for the identity column when inserting a new row.

Starting with version 2012, Microsoft™ SQL Server supports sequences:

```
-- To create a sequence object:
CREATE SEQUENCE myseq START WITH 100 INCREMENT BY 1;

-- To get a new sequence value:
SELECT NEXT VALUE FOR myseq;

-- To find the current sequence value (last generated)
SELECT convert(bigint, current_value) FROM sys.sequences WHERE name = 'myseq';

-- To reset the sequence with a new start number:
ALTER SEQUENCE myseq START WITH 100;
```

Solution

**Note:** For best SQL portability when using different types of databases, consider using sequences as described in Solution 3: Use native SEQUENCE database objects on page 628.

To emulation Informix® serials with SQL Server, you can use three different solutions:

1. Native SQL Server IDENTITY columns (default).
2. Insert triggers based on sequences (requires SQL Server 2012 and +).
3. Insert triggers based on the SERIALREG table (for SQL Server prior to 2012).
The method used to emulate SERIAL types is defined by the ifxemul.datatype.serial.emulation FGLPROFILE parameter:

```
  dbi.database.dbname.ifxemul.datatype.serial.emulation
    = \"native\" | \"trigseq\" | \"regtable\"
```

1. native: uses IDENTITY columns.
2. trigseq: uses insert triggers with sequences (CREATE SEQUENCE).
3. regtable: uses insert triggers with the SERIALREG table.

The default emulation technique is "native".

The serial types emulation can be enabled or disabled with the following FGLPROFILE entries:

```
  dbi.database.dbname.ifxemul.datatype.serial = {true | false}
  dbi.database.dbname.ifxemul.datatypeserial8 = {true | false}
  dbi.database.dbname.ifxemul.datatype.bigserial = {true | false}
```

The native IDENTITY-based solution is faster, but does not allow explicit serial value specification in insert statements; the other solutions are slower but allow explicit serial value specification in INSERT statements.

**Important:** The trigger-based solutions are provided to simplify the conversion from Informix, but are slower as the solution uses IDENTITY columns. To get best performances, we strongly recommend that you use native IDENTITY columns instead of triggers.

### Disabling automatic serial retrieval for SQLCA.SQLERRD[2]

SERIAL emulation can be totally disabled by setting the ifxemul.datatype.serial FGLPROFILE entry to false:

```
  dbi.database.dbname.ifxemul.datatype.serial = false
```

For Informix® compatibility, after an INSERT statement, the ODI drivers automatically execute another SQL query (or do a DB client API call when possible), to get the last generated serial, and fill the SQLCA.SQLERRD[2] register. This results in some overhead that can be avoided, if the SQLCA.SQLERRD[2] register is not used by the program.

When SERIAL emulation is required (to create temp tables with a serial column during program execution), and the SQLCA.SQLERRD[2] register does not need to be filled, (typically because you use your own method to retrieve the last generated serial), you can set the ifxemul.datatype.serial.sqlerrd2 FGLPROFILE entry to false. This will avoid the automatic retrieval of last serial value to fill SQLCA.SQLERRD[2]:

```
  dbi.database.dbname.ifxemul.datatype.serial.sqlerrd2 = false
```

See also db_get_last_serial() on page 2308.

### 1. Using the native serial emulation

In order to use the "native" emulation, make sure that the following FGLPROFILE entry is not defined (it is the default), or is set the entry to "native":

```
  dbi.database.dbname.ifxemul.datatype.serial.emulation = "native"
```

In database creation scripts, all SERIAL[(n)] data types must be converted by hand to INTEGER IDENTITY[(n,1)] data types, while BIGSERIAL[(n)] data types must be converted by hand to BIGINT IDENTITY[(n,1)] data types.

Tables created from the BDL programs can use the SERIAL data type. When a BDL program executes a CREATE [TEMP] TABLE with a SERIAL column, the database interface automatically converts the "SERIAL[(n)]" data type to "INTEGER IDENTITY[(n,1)]".
In BDL, the new generated SERIAL value is available from the SQLCA.SQLERRD[2] variable. This is supported by the database interface which performs a "SELECT SCOPE_IDENTITY()". However, SQLCA.SQLERRD[2] is defined as an INTEGER, it cannot hold values from BIGINT identity columns. If you are using BIGINT IDENTITY columns, you must retrieve the last generated serial with the SCOPE_IDENTITY() SQL function.

By default (see SET IDENTITY_INSERT), SQL Server does not allow you to specify the IDENTITY column in INSERT statements; You must convert all INSERT statements to remove the identity column from the list.

For example, the following statement:

```
INSERT INTO tab (col1,col2) VALUES (0, p_value)
```

must be converted to:

```
INSERT INTO tab (col2) VALUES (p_value)
```

Static SQL INSERT using records defined from the schema file (DEFINE rec LIKE tab.*) must also be reviewed:

```
INSERT INTO tab VALUES (rec.*) -- will use the serial column
```

must be converted to:

```
INSERT INTO tab VALUES rec.* -- without parentheses, serial column is removed
```

Since 2.10.06, SELECT * FROM table INTO TEMP with original table having an IDENTITY column is supported: The database driver converts the Informix® SELECT INTO TEMP to the following sequence of statements:

```
SELECT selection-items INTO #table FROM ... WHERE 1=2
SET IDENTITY_INSERT #table ON
INSERT INTO #table (column-list ) SELECT original select clauses
SET IDENTITY_INSERT #table OFF
```

See also temporary tables.

### 2. Using the trigseq serial emulation (SQL Server 2012 and +)

In order to use the serial emulation based on triggers and sequences, make sure that all database users creating tables in program have permissions to create/drop sequences and triggers.

Define the FGLPROFILE entry to enable "trigseq" serial emulation:

```
dbi.database.dbname.ifxemul.datatype.serial.emulation = "trigseq"
```

In database creation scripts, all SERIAL[(n)] data types must be converted to INTEGER data types, BIGSERIAL must be converted to BIGINT and you must create one trigger for each table. To know how to write those triggers, you can create a small Genero program that creates a table with a SERIAL column. Set the FGLSQLDEBUG environment variable and run the program. The debug output will show you the native trigger creation command using a sequence.

Tables created from the BDL programs can use the SERIAL data type. When a BDL program executes a CREATE [TEMP] TABLE with a SERIAL column, the database interface automatically converts the "SERIAL[(n)]" data type to "INTEGER" and creates the insert triggers. When using BIGSERIAL[(n)], the column is converted to a BIGINT.

**Important:**

- SQL Server does not allow you to create triggers on temporary tables. Therefore, you cannot create temp tables with a SERIAL column when using this solution.
• SELECT ... INTO TEMP statements using a table created with a SERIAL column do not automatically create the SERIAL triggers in the temporary table. The type of the column in the new table is INTEGER. Similarly, a BIGSERIAL column becomes BIGINT.
• When a table is dropped, all associated triggers are also dropped.
• INSERT statements using NULL for the SERIAL column will produce a new serial value, instead of using NULL:

```sql
INSERT INTO tab (coll, col2) VALUES (NULL, 'data')
```

This behavior is mandatory in order to support INSERT statements which do not use the serial column:

```sql
INSERT INTO tab (col2) VALUES ('data')
```

Check if your application uses tables with a SERIAL column that can contain a NULL value.

3. Using the `regtable` serial emulation (SQL Server versions prior to 2012)

**Note:** This solution is supported for SQL Server versions prior to 2012, if your server is a SQL Server 2012 or +, consider using the "trigseq" emulation instead.

In order to use the serial emulation based on triggers and the SERIALREG table, make sure that all database users creating tables in program have permissions to create/drop triggers.

Then, prepare the database and create the SERIALREG table as follows:

```sql
CREATE TABLE serialreg {
    tablename VARCHAR(50) NOT NULL,
    lastserial BIGINT NOT NULL,
    PRIMARY KEY (tablename)
}
```

The SERIALREG table and columns have to be created with lower case names, since the SQL SERVER database is created with case sensitive names, because triggers are using this table in lower case.

Define the FGLPROFILE entry to enable "regtable" serial emulation:

```sql
dbi.database.dbname.ifxemul.datatype.serial.emulation = "regtable"
```

In database creation scripts, all SERIAL[(n)] data types must be converted to INTEGER data types, BIGSERIAL must be converted to BIGINT and you must create one trigger for each table. To know how to write those triggers, you can create a small Genero program that creates a table with a SERIAL column. Set the FGLSQLDEBUG environment variable and run the program. The debug output will show you the native trigger creation command using the SERIALREG table.

Tables created from the BDL programs can use the SERIAL data type. When a BDL program executes a CREATE [TEMP] TABLE with a SERIAL column, the database interface automatically converts the "SERIAL[(n)]" data type to "INTEGER" and creates the insert triggers. When using BIGSERIAL[(n)], the column is converted to a BIGINT.

**Important:**

• The serial production is based on the SERIALREG table which registers the last generated number for each table. If you delete rows of this table, sequences will restart at 1 and you will get unexpected data.
• SQL Server does not allow you to create triggers on temporary tables. Therefore, you cannot create temp tables with a SERIAL column when using this solution.
• SELECT ... INTO TEMP statements using a table created with a SERIAL column do not automatically create the SERIAL triggers in the temporary table. The type of the column in the new table is INTEGER. Similarly, a BIGSERIAL column becomes BIGINT.
• When a table is dropped, all associated triggers are also dropped.
• INSERT statements using NULL for the SERIAL column will produce a new serial value, instead of using NULL:

```
INSERT INTO tab (col1,col2) VALUES ( NULL,'data')
```

This behavior is mandatory in order to support INSERT statements which do not use the serial column:

```
INSERT INTO tab (col2) VALUES ('data')
```

Check if your application uses tables with a SERIAL column that can contain a NULL value.

**Related concepts**

* Auto-incremented columns (serials) on page 626
  How to implement automatic record keys.

* The SQLCA diagnostic record on page 594
  The SQLCA variable is a predefined record containing SQL statement execution information.

* FGLPROFILE entries for core language on page 256
  This is a summary of FGLPROFILE entries supported by the core BDL.

**SQL Server UNIQUEIDENTIFIER data type**

SQL Server supports a special type named UNIQUEIDENTIFIER, which can be used to store “Globally Unique Identifiers” (GUIDs). UNIQUEIDENTIFIER values can be generated with the NEWID() function. When creating a table, you typically define a UNIQUEIDENTIFIER column with a DEFAULT clause where the value is produced from a NEWID() call:

```
CREATE TABLE mytab ( k INT, id UNIQUEIDENTIFIER DEFAULT NEWID(), c VARCHAR(10) )
```

The UNIQUEIDENTIFIER type is based on the BINARY(16) SQL Server type. The Genero language does not have an equivalent type for BINARY(16). However, BINARY values can be represented as hexadecimal strings in CHAR or VARCHAR variables.

A UNIQUEIDENTIFIER value is usually represented as a GUID identifier, with the following hexadecimal format:

```
XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX
```

where X can be 0-9, A-F.

You typically fetch UNIQUEIDENTIFIER data into a CHAR(36) Genero variable. The resulting value will be expressed in hexadecimal string using the format. You can then reuse that value in an SQL statement, but you have to convert the CHAR(36) hexadecimal string value back to a UNIQUEIDENTIFIER value with the CONVERT() SQL Server function, as shown in this example:

```
DEFINE pi CHAR(36)
CREATE TABLE mytab ( k INT, i UNIQUEIDENTIFIER DEFAULT NEWID(), c VARCHAR(10) )
INSERT INTO mytab ( k, c ) VALUES ( 1, 'aaa' )
SELECT i INTO pi FROM mytab WHERE k = 1
UPDATE mytab SET c = 'xxx' WHERE i = CONVERT(UNIQUEIDENTIFIER, pi)
```

When extracting a database schema, UNIQUEIDENTIFIER columns can be clearly distinguished from BINARY(N) columns. The fgldbsch tool will produce a CHAR(36) type code in the .sch file for UNIQUEIDENTIFIER columns.

You can also exclude the UNIQUEIDENTIFIER columns from the table definition in the schema file, by using the x character at the appropriate position of the string passed with the -cv data type conversion option of fgldbsch.

**SQL Server ROWVERSION data type**

SQL Server provides a special type named ROWVERSION, to stamp row modifications. The ROWVERSION data type replaces the old TIMESTAMP column definition. When you define a column with the ROWVERSION,
SQL Server will automatically increment the version column when the row is modified. ROWVERSION is just an incrementing number, it does not preserve date or time information. It can be used to control concurrent access to the same rows.

The ROWVERSION type is based on the BINARY(8) SQL Server type. The Genero language does not have an equivalent type for BINARY(8). Therefore, you must fetch ROWVERSION data into a CHAR(16) variable. The resulting value will be expressed in hexadecimal. You can then reuse that value in an UPDATE statement to check that the row was not modified by another process, but you have to convert the CHAR(16) hexadecimal value back to a BINARY(8) value with the CONVERT() SQL Server function, as shown in this example:

```sql
DEFINE pv CHAR(16)
CREATE TABLE mytab ( k INT, v ROWVERSION, c VARCHAR(10) )
INSERT INTO mytab VALUES ( 1, NULL, 'aaa' )
SELECT v INTO pv FROM mytab WHERE k = 1
UPDATE mytab SET c = 'xxx' WHERE k = 1 AND v = CONVERT(BINARY(8), pv, 2)
```

Since ROWVERSION is a synonym for BINARY(8), ROWVERSION columns cannot be clearly identified in ODBC. Therefore, the following conversion rule applies when fetching data from the server:

- If the column is defined as BINARY(N), with N<=128, the data will be fetched as a CHAR(N*2), as an hexadecimal string.
- If the column is defined as BINARY(N), with N>128, the data will be fetched as a BYTE, as a regular binary value.

When extracting a database schema, ROWVERSION columns are identified as TIMESTAMP columns and can be clearly distinguished from BINARY(N) columns. The `fgldbssch` tool will produce a CHAR(16) type code in the .sch file for ROWVERSION or TIMESTAMP columns.

**ROWID columns**

**Informix**

When creating a table, Informix® automatically adds a ROWID integer column (applies to non-fragmented tables only).

The ROWID column is auto-filled with a unique number and can be used like a primary key to access a given row.

**Note:** Informix® ROWID usage was a common practice in the early days of Informix® 4GL programming. Today it is recommended to define all your database tables with a PRIMARY KEY to uniquely identify rows.

With Informix®, the `SQLCA.SQLERRD[6]` register contains the ROWID of the last row affected by an `INSERT`, `UPDATE` or `DELETE` statement.

**Microsoft™ SQL Server**

Microsoft™ SQL Server does not support ROWIDs.

**Solution**

If your Genero BDL application uses rowid columns, review the program logic to use primary keys instead. If the database table does not define a primary key, it should be added. All references to `SQLCA.SQLERRD[6]` must be removed, because this variable will not hold the ROWID of the last modified row.

If your existing Informix® application depends on using ROWID values, you can use the `IDENTITY` property of the `DECIMAL`, `INT`, `NUMERIC`, `SMALLINT`, `BIGINT`, or `TINYINT` data types, to simulate this functionality.

For databases where the keyword of the rowid pseudo-column is different than "ROWID", the translation can be controlled with the following `FGLPROFILE` entry:

```sql
dbi.database.dsname.ifxemul.rowid = \true/ false\n```

For more details see [IBM Informix emulation parameters in FGLPROFILE](#) on page 668.
Related concepts
Using ROWID columns on page 644
Automatic ROWID columns is not a common database feature.

Case sensitivity
In Informix®, database object names like table and column names are not case sensitive:

```sql
CREATE TABLE Customer ( Custno INTEGER, ... )
SELECT CustNo FROM cuSTomer ... 
```

In Microsoft™ SQL Server, database object names and character data are case-insensitive by default:

```sql
CREATE TABLE Customer ( Custno INTEGER, CustName CHAR(20) )
INSERT INTO CUSTOMER VALUES ( 1, 'TECHNOSOFT' )
SELECT CustNo FROM cuSTomer WHERE custname = 'techNOSoft'
```

The installation program of SQL Server allows you to customize the sort order. The sort order specifies the rules used by SQL Server to collate, compare, and present character data. It also specifies whether SQL Server is case-sensitive.

Genero compilers convert table and column names to lower case. For example, when writing following static SQL statement:

```sql
SELECT COUNT(*) FROM CUSTOMER WHERE CUSTNAME LIKE 'S%'
```

The SQL text stored in the pcode module will be:

```sql
SELECT COUNT(*) FROM customer WHERE custname LIKE 'S%'
```

Solution
Select the case-sensitive sort order when installing SQL Server to make queries case-sensitive.

Define the database tables and columns in lower case only, because Genero compilers convert them to lower case.

See also Name resolution of SQL objects on page 856.

**TEXT and BYTE (LOB) types**

**Informix®**

Informix® provides the TEXT, BYTE, CLOB and BLOB data types to store very large texts or binary data.

Legacy Informix® 4GL applications typically use the TEXT and BYTE types.

Genero BDL does not support the Informix® CLOB and BLOB types.

**Microsoft™ SQL Server**

Microsoft™ SQL Server provides the VARCHAR (MAX), NVARCHAR (MAX) and VARBINARY (MAX) data types to store large object data.

The SQL Server TEXT, NTEXT and IMAGE data types still exist, but are considered as obsolete and will be removed in a future version.

In SQL Server, the VARCHAR (MAX), NVARCHAR (MAX) and VARBINARY (MAX) types have a limit of 2 gigabytes (2^31 -1 actually).
Solution

In Genero programs connecting to SQL Server, the TEXT and BYTE data types of DDL statements such as CREATE TABLE are respectively converted to VARCHAR(MAX) and VARBINARY(MAX) types.

SQL Server database drivers make the appropriate bindings to use TEXT and BYTE Genero types as SQL parameters and fetch buffers, and can be used for SQL SERVER text, image or VARCHAR(MAX), NVARCHAR(MAX) and VARBINARY(MAX) columns.

To store TEXT data, when using a multibyte character set (such as UTF-8), define database columns as NVARCHAR(MAX). When using a single-byte character set in your application (like ISO-8859-15), the column type can be VARCHAR(MAX).

By default, in DDL statements executed by programs (such as CREATE TABLE), the TEXT type name is left untouched. To force the ODI drivers to replace TEXT by the NVARCHAR(MAX) type name when the locals is multibyte, set the dbi.database.dsname.ifxemul.nationalchars FGLPROFILE entry to true (default is false):

```
dbi.database.dsname.ifxemul.nationalchars = true
```

Note:

1. With SQL Server ODI drivers, dbi.database.dsname.ifxemul.nationalchars will only take effect, if the current application locale is multibyte (typically, UTF-8)
2. The dbi.database.dsname.ifxemul.nationalchars parameter is ignored, if the switch corresponding to the character type name (dbi.database.dsname.datatype.{char|varchar|text}) is set to false.
3. With the SNC ODI driver, to have dbi.database.dsname.ifxemul.nationalchars take effect, the wide-char mode must be enabled (this is the default with multibyte character sets).

Genero TEXT/BYTE program variables and the SQL Server large object types have the same a limit of 2 gigabytes.

Note: When using a stored procedure that has SET/IF statements and produces a result set with LOBs, the LOB columns must appear at the end of the SELECT list. If LOB columns are followed by other columns with regular types, the fetching rows will fail. Using SET NOCOUNT ON in the stored procedure does not help, because the cursor type is changed from a server cursor to a default result set cursor.

The TEXT and BYTE types translation can be controlled with the following FGLPROFILE entries:

```
dbi.database.dsname.ifxemul.text = { true | false }
dbi.database.dsname.ifxemul.byte = { true | false }
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

Related concepts

Using portable data types on page 614

Only a limited set of data types are really portable across several database engines.

Table constraints

Informix®

Informix® supports primary key, unique, foreign key, default and check constraints.

The constraint naming syntax is different in Informix® and most other databases: Informix expects the constraint name after the constraint definition:

```
CREATE TABLE emp (  
  ...  
  emp_code CHAR(10) UNIQUE CONSTRAINT pk_emp,  
  ...)
```
While other SQL database brands require to specify the constraint name before the constraint definition:

```sql
CREATE TABLE emp (
    ...
    emp_code CHAR(10) CONSTRAINT pk_emp UNIQUE,
    ...
)
```

**Microsoft™ SQL Server**

Microsoft™ SQL Server supports primary key, unique, foreign key, default and check constraints.

**Constraint naming**

The constraint naming clause must be placed before the constraint specification.

**Important:** SQL Server does not produce an error when using the Informix® syntax of constraint naming.

**The NULL / NOT NULL constraint**

**Note:** Microsoft™ SQL Server creates columns as NOT NULL by default, when no NULL constraint is specified (colname datatype {NULL | NOT NULL}). A special option is provided to invert this behavior: ANSI_NULL_DFLT_ON. This option can be enabled with the SET command, or in the database options of SQL Server Management Studio.

**Solutions**

**Constraint naming syntax**

The database interface does not convert constraint naming expressions when creating tables from BDL programs. Review the database creation scripts to adapt the constraint naming clauses for Microsoft™ SQL Server.

**The NULL / NOT NULL constraint**

Before using a database, you must check the "ANSI NULL Default" option in the database properties if you want to have the same default NULL constraint as when using Informix®.

**Related concepts**

- Data definition statements on page 613

It is recommended to avoid use of DDL in programs.

**Name resolution of SQL objects**

**Informix®**

Informix® uses the following form to identify an SQL object:

```
database[@dbservername]:][{owner|"owner"}].identifier
```

The ANSI convention is to use double quotes for identifier delimiters (For example: "customer", "cust_name").

Informix® database object names are not case-sensitive in non-ANSI databases. When using double-quoted identifiers, Informix® becomes case sensitive.
With non-ANSI Informix® databases, you do not have to give a schema name before the tables when executing an SQL statement:

```
SELECT ... FROM customer WHERE ...
```

In Informix® ANSI compliant databases:

- The table name must include "owner", unless the connected user is the owner of the database object.
- The database server shifts the owner name to uppercase letters before the statement executes, unless the owner name is enclosed in double quotes.

**Microsoft™ SQL Server**

With Microsoft™ SQL Server, an object name takes the following form:

```
[[database.]owner.]identifier
```

Object names are limited to 128 characters in SQL Server and cannot start with one of the following characters: @ (local variable) # (temp object).

To support double quotes as string delimiters in SQL Server, you can switch OFF the database option "Use quoted identifiers" in the database properties panel. But quoted table and column names are not supported when this option is OFF.

**Solution**

To write portable SQL, regarding database object names:

1. Use simple database object names (without any owner/schema prefix)
2. Do not use double quotes to surround database object identifiers.
3. If needed, define public synonyms to reference database objects in others databases/schema.
4. Specify database object identifiers in lowercase.

See also Naming database objects on page 637.

Check for single or double quoted table or column names in your source and remove them.

If cross-database queries are required (using the dbname:tabname Informix notation), consider to create views in the main database in SQL Server, to allow program to access the distant table from the same database connection:

```
CREATE VIEW myotherdb_customers AS SELECT * FROM myotherdb.dbo.customers
```

**Related concepts**

Database concepts on page 831

**Data manipulation**

Microsoft™ SQL Server related data manipulation topics.

**Reserved words**

**Informix®**

With Informix®, it is possible to create database objects with reserved words.

For example:

```
CREATE TABLE table ( char CHAR(10) );
```

Indeed this is not good practice, but Informix® SQL allows this to be backward compatible when introducing a new keyword in the SQL syntax.
Most other database systems do not allow reserved words as database identifiers. If your legacy code is using SQL reserved words of the target database SQL syntax, an error will be thrown at CREATE TABLE execution.

**Microsoft™ SQL Server**

Microsoft™ Transact-SQL does not allow you to use reserved words as database object names (tables, columns, constraint, indexes, triggers, stored procedures, ...).

An example of a common word which is part of SQL Server grammar is 'go' (see the 'Reserved keywords' section in the SQL Server Documentation).

**Solution**

Database objects having a name which is a Transact-SQL reserved word must be renamed.

All BDL application sources must be verified. To check if a given keyword is used in a source, you can use UNIX™ 'grep' or 'awk' tools. Most modifications can be automatically done with UNIX™ tools like 'sed' or 'awk'.

You can use SET QUOTED_IDENTIFIERS ON with double-quotes to enforce the use of keywords in the database objects naming, but it is not recommended.

**Outer joins**

**Informix® OUTER() syntax**

In Informix® SQL, outer joins can be defined in the FROM clause with the OUTER keyword:

```sql
SELECT ... FROM a, OUTER (b)
WHERE a.key = b.akey

SELECT ... FROM a, OUTER(b,OUTER(c))
WHERE a.key = b.akey
AND b.akey1 = c.bkey1 AND b.key2 = c.bkey2
```

Informix® also supports the ANSI OUTER join syntax, which is the recommended way to specify outer joins with recent SQL database engines:

```sql
SELECT ... FROM cust LEFT OUTER JOIN order
ON cust.key = order.custno
WHERE ... 
```

**Microsoft™ SQL Server**

Microsoft™ SQL Server supports the ANSI outer join syntax:

```sql
SELECT ... FROM cust LEFT OUTER JOIN order
    LEFT OUTER JOIN item
    ON order.key = item.ordno
    ON cust.key = order.custno
WHERE order.cdate > current date
```

**Note:**

The legacy syntax to define outers in SQL Server uses the *:= notation:

```sql
SELECT ... FROM a, b WHERE a.key *= b.key
```

**Solution**

The Genero database drivers can convert Informix Informix® OUTER specifications to ANSI outer joins.
**Note:** For better SQL portability, use the ANSI outer join syntax instead of the old Informix® OUTER syntax.

The outer join translation can be controlled with the following FGLPROFILE entry:

```plaintext
dbi.database.dsnname.ifxemul.outers = \true \false
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Prerequisites:**

1. In the FROM clause, the main table must be the first item and the outer tables must be listed from left to right in the order of outer levels.

   Example which does not work:

   ```plaintext
   ... FROM OUTER(tab2), tab1
   ```

2. The outer join in the WHERE clause must use the table name as prefix:

   ```plaintext
   ... WHERE tab1.col1 = tab2.col2
   ```

**Restrictions:**

1. Statements composed by 2 or more SELECT instructions are not supported:

   ```plaintext
   SELECT ... UNION SELECT ...
   ```

   or:

   ```plaintext
   SELECT ... WHERE col IN (SELECT...)
   ```

2. Additional conditions on outer table columns cannot be detected and therefore are not supported:

   ```plaintext
   ... FROM tab1, OUTER(tab2)
   WHERE tab1.col1 = tab2.col2
   AND tab2.colx > 10
   ```

3. Using subscript in outer conditions:

   ```plaintext
   ... FROM tab1, OUTER(tab2)
   WHERE tab1.col1[1,3] = tab2.col2[1,3]
   ```

**Notes:**

1. Table aliases are detected in OUTER expressions.

   OUTER example with table alias:

   ```plaintext
   ... OUTER(tab1 alias1) ...
   ```

2. In the outer join, `outertab.col` can be placed on both right or left sides of the equal sign:

   ```plaintext
   ... WHERE outertab.col1 = maintab.col2
   ```

3. Table names detection is not case-sensitive:

   ```plaintext
   SELECT ... FROM tab1, TAB2
   WHERE tab1.col1 = tab2.col2
   ```

4. Temporary tables are supported in OUTER specifications:

   ```plaintext
   CREATE TEMP TABLE tt1 ( ... )
   SELECT ... FROM tab1, OUTER(tt1) ...
   ```
**Related concepts**

*Outer joins* on page 643

Use standard ISO outer join syntax instead of the old IBM® Informix® OUTER() syntax.

**Transactions handling**

**Informix®**

With the Informix® native mode (non ANSI):

- Transactions blocks start with `BEGIN WORK` and terminate with `COMMIT WORK` or `ROLLBACK WORK`.
- Statements executed outside a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.

```sql
UPDATE tab1 SET ... -- auto-committed
BEGIN WORK            -- start of TX block
UPDATE tab1 SET ...  -- start of TX block
UPDATE tab2 SET ...  -- start of TX block
...
COMMIT WORK           -- end of TX block
```

Informix® version 11.50 introduces savepoints:

```sql
SAVEPOINT name [UNIQUE]
ROLLBACK [WORK] TO SAVEPOINT [name]
RELEASE SAVEPOINT name
```

**Microsoft™ SQL Server**

Microsoft™ SQL Server supports named and nested transactions:

- Transactions are started with `BEGIN TRANSACTION [name].`
- Transactions are validated with `COMMIT TRANSACTION [name].`
- Transactions are canceled with `ROLLBACK TRANSACTION [name].`
- Savepoints can be placed with `SAVE TRANSACTION name.`
- Transactions can be rolled back to a savepoint with `ROLLBACK TRANSACTION TO name.`
- Savepoints can not be released.
- Statements executed outside of a transaction are automatically committed (autocommit mode). This behavior can be changed with "SET IMPLICIT_TRANSACTION ON".
- DDL statements are not supported in transactions blocks.

**Solution**

Informix® transaction handling commands are automatically converted to Microsoft™ SQL Server instructions to start, validate or cancel transactions.

Regarding the transaction control instructions, the BDL applications do not have to be modified to work with Microsoft™ SQL Server.

**Important:** If you want to use savepoints, do not use the UNIQUE keyword in the savepoint declaration, always specify the savepoint name in `ROLLBACK TO SAVEPOINT`, and do not drop savepoints with `RELEASE SAVEPOINT`.

**Related concepts**

*Database transactions* on page 600
Database transactions define a set of SQL instructions to be executed as a whole, or rolled back as a whole.

**Temporary tables**

**Informix**

Informix® temporary tables are created with the `CREATE TEMP TABLE` DDL instruction or with `SELECT ... INTO TEMP` statement:

```sql
CREATE TEMP TABLE tt1 ( pkey INT, name VARCHAR(50) )
CREATE TEMP TABLE tt2 ( pkey INT, name VARCHAR(50) ) WITH NO LOG
SELECT * FROM tab1 WHERE pkey > 100 INTO TEMP tt2
```

Temporary tables are automatically dropped when the SQL session ends, but they can also be dropped with the `DROP TABLE` command. There is no name conflict when several users create temporary tables with the same name.

BDL REPORTs can create a temporary table when the rows are not sorted externally (by the source SQL statement).

Informix® allows you to create indexes on temporary tables. No name conflict occurs when several users create an index on a temporary table by using the same index identifier.

When creating temporary tables in Informix®, the `WITH NO LOG` clause can be used to avoid the overhead of recording DML operations in transaction logs.

**Microsoft™ SQL Server**

Microsoft™ SQL Server provides local (SQL session wide) or global (database wide) temporary tables by using the '#' or '##' characters as table name prefix.

The following SQL Server statement creates a session temporary table with the `CREATE TABLE` statement:

```sql
CREATE TABLE #temp1 ( kcol INTEGER, ... )
```

To create and fill a session temporary table in one step, use the SQL Server `SELECT ... INTO` statement:

```sql
SELECT * INTO #temp2 FROM customers WHERE ...
```

**Important:** When not specifying the collation in the column definition, CHAR/VARCHAR columns of SQL Server temporary tables created with `CREATE TABLE #tabname` get by default the collation of the tempdb system database. If the collation of the temp table columns does not match the current database collation, SQL Server will return the error 468 "Cannot resolve the collation conflict between A and B in the ..." when comparing CHAR/VARCHAR columns from permanent and temporary tables, as in the next query:

```sql
SELECT COUNT(*) FROM #temptab1 WHERE name IN (SELECT cust_name FROM customers)
```

To create columns with the database collation and avoid the 468 collaction mismatch error, one must add the `COLLATE DATABASE_DEFAULT` clause after CHAR/VARCHAR and NCHAR/NVARCHAR types in the `CREATE TABLE #tabname` statement.

When using the `SELECT * INTO #tabname` SQL Server statement, the character type columns of the temp table get the current database collation automatically.

**Solution**

In BDL, Informix® temporary tables instructions are converted to generate native SQL Server temporary tables.

When Informix emulations are enabled, `CREATE TEMP TABLE tabname` statements executed by Genero programs are converted to `CREATE TABLE #tabname` for SQL Server, and `SELECT <select-list> <select-body> INTO TEMP tabname` statements are converted to SQL Server `SELECT <select-list> INTO #tabname <select-body>`. 
**Important:**

Simple Informix-style SQL statement creating temporary tables can be converted to a native SQL equivalent instruction. However, complex SQL statements such as `SELECT .. INTO TEMP` with subqueries may fail. In such cases, create a view from the complex query and then create the temp table from the view. Or, disable Informix emulation and use the native SQL syntax to create the temporary table (EXECUTE IMMEDIATE "/* fglhint_no_ifxemul */ ...")

With Informix SQL, if the source table has a column defined as `SERIAL` or `BIGSERIAL`, a `SELECT ... INTO TEMP` will produce a new temp table with an auto-incremented serial column. With the `SELECT ... INTO TEMP` emulation for non-Informix databases, not using the native sequence generators (such as `IDENTITY` columns in SQL Server), the resulting temporary table will get a simple `INTEGER` or `BIGINT` column, instead of an auto-incremented column.

To avoid 468 SQL errors when comparing character type columns of temp tables and permanent tables, the collation of the temporary table columns must match the collation of the permanent table.

When using a converted Informix `SELECT ... INTO TEMP tabname` statement, or a native SQL Server `SELECT ... INTO #tabname` statement, the character columns of the temporary table will automatically get the default collation of the current database, and the error 468 will not occur when comparing character columns.

When using a converted Informix `CREATE TEMP TABLE tabname` statement, or the native SQL Server `CREATE TABLE #tabname` statement, the temporary table columns get by default the collation of the tempdb system database, inherited from the SQL Server instance collation. To avoid 468 errors, only for `CREATE TEMP TABLE` statements, the ODI driver will add the `COLLATE DATABASE_DEFAULT` clause to SQL Server character types corresponding to Informix `CHAR`, `VARCHAR`, `LVARCHAR` and `TEXT` type names. For example, a `CREATE TEMP TABLE tt1 ( col1 VARCHAR(20) NOT NULL )` will be converted to `CREATE TABLE #tt1 ( col1 VARCHAR(20) COLLATE DATABASE_DEFAULT NOT NULL )`

**Tip:** To avoid having the ODI drivers add the `COLLATE DATABASE_DEFAULT` clause, disable Informix emulation switches for character types, and replace the original Informix types by the required SQL Server types:

```sql
dbi.database.dsname.datatype.char = false
dbi.database.dsname.datatype.varchar = false
dbi.database.dsname.datatype.text = false
```

Another option is to use dynamic SQL and add the `fglhint` comment to disable Informix emulations and use directly the native SQL Server syntax:

```sql
EXECUTE IMMEDIATE "/* fglhint_no_ifxemul */ CREATE TABLE #tt1 ... "
```

The general FGLPROFILE entry to control temporary table emulation is:

```sql
dbi.database.dsname.ifxemul.temptables = | true | false |
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Note:** Microsoft™ SQL Server does not support scroll cursors based on a temporary table.

**Related concepts**

*Temporary tables* on page 641

Syntax for temporary table creation is not unique across all database engines.

**Substrings in SQL**

**Informix®**

Informix® SQL statements can use subscripts on columns defined with the character data type:

```sql
SELECT ... FROM tab1 WHERE col1[2,3] = 'RO'
SELECT ... FROM tab1 WHERE col1[10] = 'R' -- Same as col1[10,10]
```
Important: With other database servers as Informix®, when the subscript notation is used to modify column values in UPDATE statement, or as ORDER BY element, you will get and SQL error:

```
UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...
SELECT ... FROM tab1 ORDER BY col1[1,3]
```

**Microsoft™ SQL Server**

Microsoft™ SQL Server provides the `SUBSTR()` function, to extract a substring from a string expression:

```
SELECT .... FROM tab1 WHERE SUBSTRING(col1,2,2) = 'RO'
SELECT SUBSTRING('Some text',6,3) FROM tab1 -- Gives 'tex'
```

**Solution**

Replace all Informix® `col[x,y]` right-value expressions with `SUBSTRING(col,x,y-x+1)`. Rewrite UPDATE and ORDER BY clauses using `col[x,y]` expressions.

The translation of `col[x,y]` expressions can be controlled with the following FGLPROFILE entry:

```
dbi.database.dsname.ifxemul.colsubs = true | false
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

*Substring expressions* on page 644

Handle substrings expressions with different database engines.

**String delimiters**

**Informix®**

The ANSI SQL string delimiter character is the single quote (`'string'`), while double quotes are used to delimit database object names:

```
SELECT ... WHERE "tabname"."colname" = 'a string value'
```

In Informix® databases created in native mode (non-ANSI), you can use double quotes as string delimiters:

```
SELECT ... WHERE tabname.colname = 'a string value'
```

This is important, since many BDL programs use that character to delimit the strings in SQL commands.

**Note:** This problem concerns only double quotes within SQL statements. Double quotes used in pure BDL string expressions are not subject to SQL compatibility problems.

**Microsoft™ SQL Server**

Microsoft™ SQL Server follows the ANSI SQL specification, using single quotes for string delimiters and double quotes for database object names.

**Important:**

With SQL Server, all UNICODE strings must be prefaced with an `N` character:

```
UPDATE cust SET cust_name = N'###' WHERE cust_id=123
```
Without the N prefix, SQL Server will convert the characters from the current system locale to the database locale. With the N prefix, the server can recognize a UNICODE string and use it as is to insert into NCHAR or NVARCHAR columns.

Solution

When using Static SQL statements, the fglcomp compiler converts string literals using double quotes to string literals with single quotes:

```sql
$ cat s.4gl
MAIN
  DEFINE n INT
  SELECT COUNT(*) INTO n FROM tab1 WHERE col1 = "abc"
END MAIN
$ fglcomp -S s.4gl
s.4gl^3^SELECT COUNT(*) FROM tab1 WHERE col1 = 'abc'
```

However, SQL statements created dynamically are not modified by the Genero compiler.

The Genero database interface can automatically replace all double quotes by single quotes in SQL statements. This applies to static and dynamic SQL statements.

The translation of double quoted expression to single quoted expressions can be controlled with the following FGLPROFILE entry:

```
  dbi.database.dbname.ifxemul.dblquotes = { true | false }
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

However, database object names must not be delimited by double quotes, because the database interface cannot determine the difference between a database object name and a quoted string! For example, if the program executes the SQL statement:

```sql
... WHERE "tabname"."colname" = "a string value"
```

replacing all double quotes by single quotes would produce:

```sql
... WHERE 'tabname'.'colname' = 'a string value'
```

This would produce an error since 'tabname'.'colname' is not allowed by ORACLE.

Escaped string delimiters can be used inside strings like the following:

```
'This is a single quote: '"
'This is a single quote: \\
"This is a double quote: '""
```

Although double quotes are replaced automatically in SQL statements, it is recommended that you use only single quotes to enforce portability.

National character strings

When using the SNC database driver, all string literals of an SQL statement are automatically changed to get the N prefix. Thus, you don't need to add the N prefix by hand in all of your programs. This solution makes by the way your Genero code portable to other databases.

With the SNC database driver, character string data is converted from the current Genero BDL locale to Wide Char (Unicode UCS-2), before it is used in an ODBC call such as SQLPrepareW or SQLBindParameter(SQL_C_WCHAR). When fetching character data, the SNC database driver converts from Wide
Char to the current Genero BDL locale. The current Genero BDL locale is defined by LANG, and if LANG is not defined, the default is the ANSI Code Page of the Windows™ operating system. See CHARACTER data types for more details.

When using the FTM (FreeTDS) or the ESM (Easysoft) database driver on UNIX™, string literals get the N prefix if the current locale is a multibyte encoding like BIG5, EUC-JP or UTF-8. If the current locale is a single-byte encoding like ISO-8859-1, no prefix will be added to the string literals.

**Related concepts**

String literals in SQL statements on page 634

Single quotes is the standard for delimiting string literals in SQL.

**Single row SELECT**

**Informix®**

With Informix®, you must use the system table with a condition on the table id:

```sql
SELECT user FROM systables WHERE tabid=1
```

**Microsoft™ SQL Server**

With SQL Server, omit the FROM clause to generate one row only:

```sql
SELECT user
```

**Solution**

Check the BDL sources for "FROM systables WHERE tabid=1" and use dynamic SQL to resolve this problem.

Consider writing a FUNCTION which produces the FROM and WHERE part, depending on the target database type.

**MATCHES and LIKE**

**Informix®**

Informix® supports MATCHES and LIKE operators in SQL statements.

MATCHES expects * and ? wild-card characters, while LIKE uses the % and _ wild-cards as equivalents.

```sql
( col MATCHES 'Smi*' AND col NOT MATCHES 'R?x' )
( col LIKE 'Smi%' AND col NOT LIKE 'R_x' )
```

MATCHES accepts also brackets notation, to specify a set of matching characters at a given position:

```sql
( col MATCHES '[Pp]aris' )
( col MATCHES '[0-9][a-z]*' )
```

**Microsoft™ SQL Server**

Microsoft™ SQL Server does not provide an equivalent of the Informix® MATCHES operator.

The LIKE operator is supported.

**Important:**

The LIKE operator of SQL Server does not evaluate to true with CHAR/NCHAR columns, if the LIKE pattern is provided as a UNICODE string literal (with the N prefix), and the search pattern matches the value in the column (without an ending % wildcard for example).
See the following test:

```
CREATE TABLE mytable ( k INT, nc NCHAR(20) )
INSERT INTO mytable VALUES ( 1, N'abc' )
SELECT * FROM mytable WHERE nc = 'abc' -- one row is returned
SELECT * FROM mytable WHERE nc = N'abc' -- one row is returned
SELECT * FROM mytable WHERE nc LIKE 'abc' -- one row is returned
SELECT * FROM mytable WHERE nc LIKE N'abc' -- no rows are found
SELECT * FROM mytable WHERE nc LIKE N'abc%' -- one row is returned
```

This can be an issue because the SQL Server driver will by default automatically add an N prefix before all string literals in SQL statements.

See Microsoft™ SQL Server documentation for more details about the LIKE semantics regarding blank padding and see also CHARACTER data types for the N prefix usage and single-char or wide-char mode usage.

**Solution**

The database driver is able to translate Informix® MATCHES expressions to LIKE expressions, when no [ ] bracket character ranges are used in the MATCHES operand.

The MATCHES to LIKE expression translation is controlled by the following FGLPROFILE entry:

```
dbi.database.dbname.ifxemul.matches = \{ true \ false \}
```

**Important:** Only [NOT] MATCHES followed by a search pattern provided as a string literal can be converted by ODI drivers. A [NOT] MATCHES followed by a ? question mark parameter place holder is not translated!

For maximum portability, consider replacing the MATCHES expressions with LIKE expressions in all SQL statements.

Avoid using CHAR(N) types for variable length character data (such as name, address).

**LIKE with UNICODE string literals on CHAR/NCHAR columns**

Pay attention to UNICODE string prefixes N'...' in the LIKE expressions when used with CHAR/NCHAR columns:

You might want to always add a % wildcard at the end of the LIKE expression, or use the equal operator when doing a query with exact values.

**Related concepts**

MATCHES and LIKE operators on page 645

Use the standard LIKE operator instead of the MATCHES operator.

**The LENGTH() function**

**Informix®**

Informix® provides the `LENGTH()` function to count the number of bytes of a character string expression:

```
SELECT LENGTH("aaa"), LENGTH(col1) FROM table
```

Informix® `LENGTH()` does not count the trailing blanks for CHAR or VARCHAR expressions, while Oracle counts the trailing blanks.

Informix® `LENGTH()` returns 0 when the given string is empty. That means, `LENGTH('')=0`.

**Microsoft™ SQL Server**

Microsoft™ SQL Server supports the `LEN()` function, but there are some differences with Informix® `LENGTH()`.

**Note:** Do not confuse `LEN()` with `DATALEN()`, which returns the data size used for storage (number of bytes).
Like Informix®, Microsoft™ SQL Server ignores trailing blanks when computing the length of a string. When passing NULL as parameter, the SQL Server LEN() function returns NULL.

**Solution**

The database driver is able to replace LENGTH() by the LEN() function name. The translation of LENGTH() expressions can be controlled with the following FGLPROFILE entry:

```
dbi.database.dsname.ifxemul.length = {true | false}
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

The LENGTH() function in SQL on page 647

The semantics of the LENGTH() SQL function differs according to the database engine.

**Row limiting clause**

**Informix®**

Informix® SQL supports the SKIP and FIRST/LIMIT keywords to limit the number of rows of a result set:

```
SELECT SKIP 10 FIRST 20 customer.* FROM customer ... ORDER BY cust_name
```

This Informix SQL syntax is not portable.

Recent database engines support the row limiting clause syntax defined by the SQL standard:

```
SELECT ... OFFSET n ROWS FETCH FIRST m ROWS ONLY
```

This should be the preferred syntax to be used, if all target database types support this SELECT clause.

The ODI database drivers can convert the Informix SQL SKIP/FIRST row limiting clause to a native SQL equivalent, if the row limiting clause parameters are simple integer literals (the clause is not translated when using SQL parameters / program variables).

**Important:** In addition to the SKIP/FIRST clause of the projection clause, Informix SQL supports also a LIMIT clause after the ORDER BY clause:

```
SELECT customer.* FROM customer ... ORDER BY cust_name LIMIT 10
```

This Informix SQL syntax construction is not converted by the ODI drivers. To benefit from the conversion, review the code to use the Informix SQL SKIP/FIRST clause instead.

**Microsoft™ SQL Server**

Microsoft™ SQL Server supports the following row limiting clause:

```
SELECT ... ORDER BY ... OFFSET n ROWS [ FETCH FIRST m ROWS ONLY ]
```

**Solution**

The Informix SQL row limiting clause can be converted by the Microsoft™ SQL Server driver to the native SQL equivalent clause, when the parameters are simple integer literals.

**Note:** The row limiting clause must not use SQL parameters. Only row limiting clauses using integer constants will be converted.
The translation of the Informix SQL row limiting clause can be controlled with the following FGLPROFILE entry:

```
| dbi.database.dsnname.ifxemul.rowlimiting = true | false |
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

- **Row limiting clause (SELECT)** on page 650
- How to use the right clause to limit the number of rows produced by a SELECT statement?

**String concatenation operator**

**Informix®**

The Informix® concatenation operator is the double pipe (||):

```
SELECT firstname || ' ' || lastname FROM employee
```

**Microsoft™ SQL Server**

The Microsoft™ SQL Server concatenation operator is the plus sign:

```
SELECT firstname + ' ' + lastname FROM employee
```

**Solution**

The database interface detects double-pipe operators in SQL statements and converts them to a plus sign automatically.

**BDL programming**

Microsoft™ SQL Server related programming topics.

**SQL errors on PREPARE**

**Informix®**

With Informix®, a PREPARE instruction returns an SQL error in case of problem:

```
TRY
  PREPARE stmt FROM "SELECT * FROM WHERE pk=1"  -- table is missing!
CATCH
  DISPLAY "SQL ERROR:", SQLCA.SQLCODE
END TRY
```

**Microsoft™ SQL Server**

The Genero database drivers for Microsoft™ SQL Server are based on ODBC drivers (SQL Native Client, MS ODBC SQL, FreeTDS or Easysoft drivers). These ODBC drivers use system stored procedures to prepare and execute SQL statements (you can check this with the SQL Server Profiler).

With SQL server, PREPARE or DECLARE instructions do not return an SQL error, because the statement preparation is deferred to the execution, improving overall performances.

When preparing an SQL statement with the BDL PREPARE or DECLARE instruction, no SQL error will be returned if the SQL statement is invalid. However, an SQL error will occur after the OPEN / FOREACH / EXECUTE instructions.
Solution

Make sure your BDL programs do not test the STATUS or SQLCA.SQLCODE variable just after PREPARE instructions.

Change the program logic in order to handle the SQL errors when opening the cursors (OPEN) or when executing SQL statements (EXECUTE).

INSERT cursors

Informix®

Informix® provides **insert cursors** to optimize row creation in a database. An insert cursor is declared as a cursor, and rows as added with the PUT instruction. The rows are buffered and sent to the database server when executing a FLUSH instruction, or when the cursor is closed with CLOSE. When using transactions in Informix®, the OPEN, PUT and FLUSH instructions must be executed within a transaction block.

```sql
DECLARE c1 CURSOR FOR INSERT INTO tab1 ...
BEGIN WORK
OPEN c1
WHILE ...
    PUT c1 USING var-list
END WHILE
CLOSE c1
COMMIT WORK
```

**Microsoft™ SQL Server**

Microsoft™ SQL Server does not support insert cursors.

Solution

Insert cursors are emulated by the database interface, using basic INSERT SQL instructions.

The performances might be not as good as with Informix®, but the feature is fully supported.

Related concepts

- [Insert cursors](#) on page 633
- Using insert cursors with non-Informix databases.

Cursors WITH HOLD

Informix®

Informix® closes opened cursors automatically when a transaction ends, unless the WITH HOLD option is used in the DECLARE instruction:

```sql
DECLARE c1 CURSOR WITH HOLD FOR SELECT ...
OPEN c1
BEGIN WORK
FETCH c1 ...
COMMIT WORK
FETCH c1 ...
CLOSE c1
```

**Microsoft™ SQL Server**

Microsoft™ SQL Server does not close cursors when a transaction ends.

This behavior is controlled with the **SET CURSOR_CLOSE_ON_COMMIT ON/OFF** instruction.
Solution

BDL cursors declared WITH HOLD remain open even after terminating a transaction with a COMMIT WORK or ROLLBACK WORK.

For consistency with other database brands, database cursors that are not declared WITH HOLD are automatically closed, when a COMMIT WORK or ROLLBACK WORK is performed.

Important: Opening a WITH HOLD cursor declared with a SELECT FOR UPDATE results in an SQL error; in the same conditions, this does not normally appear with Informix®. Review the program logic in order to find another way to set locks.

Related concepts
Cursors WITH HOLD on page 631
Programming WITH HOLD cursors using SELECT with and without FOR UPDATE clause.

SELECT ... FOR UPDATE

Informix®

Legacy BDL programs typically use a cursor with SELECT FOR UPDATE to implement pessimistic locking and avoid several users editing the same rows:

```
DECLARE cc CURSOR FOR
SELECT ... FROM tab WHERE ... FOR UPDATE
OPEN cc
FETCH cc  lock is acquired
...
CLOSE cc  lock is released
```

The row must be fetched in order to set the lock.

If the cursor is local to a transaction, the lock is released when the transaction ends. If the cursor is declared WITH HOLD, the lock is released when the cursor is closed.

Informix® provides the SET LOCK MODE instruction to define the lock wait timeout:

```
SET LOCK MODE TO { WAIT | NOT WAIT | WAIT seconds }
```

The default mode is NOT WAIT.

Microsoft™ SQL Server

Microsoft™ SQL Server allows individual and exclusive row locking by using server cursors with Scroll Locks option, combined with SQL hints such as UPDLOCK in the FROM clause:

```
SELECT ... FROM tab1 WITH(UPDLOCK) WHERE ...
```

Note: In Transact-SQL, the FOR UPDATE [OF col-list] clause is not part of the SELECT syntax; It is part of T-SQL DECLARE CURSOR syntax.

Individual locks are acquired when fetching the rows.

When the cursor is opened outside a transaction (BDL WITH HOLD cursor option), locks are released when the cursor is closed.

When the cursor is opened inside a transaction, locks are released when the transaction ends.

SQL Server's locking granularity is at the row level, page level or table level (the level is automatically selected by the engine for optimization).
Solution

When executing a SELECT ... FOR UPDATE in the program, the SQL Server database drivers remove the FOR UPDATE clause from the SQL text and set the ODBC cursor attribute SQL_ATTR_CONCURRENCY to SQL_CONCUR_LOCK. This enables Scroll Locks concurrency in the server cursor.

The Scroll Locks option implements pessimistic concurrency control, in which the application attempts to lock the underlying database rows at the time they are read into the cursor result set.

SQL Server Transact-SQL hints such as UPDLOCK can be used to fine-tune the locking semantics that will be used by SQL Server.

When using server cursors with Scroll Locks option, an update lock is placed on the row when it is read into the cursor. If the cursor is opened within a transaction, the transaction update lock is held until the transaction is either committed or rolled back; the cursor lock is dropped when the next row is fetched.

If the cursor has been opened outside a transaction, the lock is dropped when the next row is fetched.

Therefore, it is recommended that a cursor is opened in a transaction whenever the user wants full pessimistic concurrency control.

An update lock prevents any other task from acquiring an update or exclusive lock, which prevents any other task from updating the row.

An update lock, however, does not block a shared lock, so it does not prevent other tasks from reading the row unless the second task is also requesting a read with an update lock.

SELECT FOR UPDATE statements are well supported in BDL as long as they are used inside a transaction. Avoid cursors declared WITH HOLD.

Note: The database interface is based on an emulation of an Informix® engine using transaction logging. Therefore, opening a SELECT ... FOR UPDATE cursor declared outside a transaction will raise an SQL error -255 (not in transaction).

The SELECT FOR UPDATE statement cannot contain an ORDER BY clause if you want to perform positioned updates/deletes with the WHERE CURRENT OF clause.

Cursors declared with SELECT ... FOR UPDATE using the WITH HOLD clause cannot be supported with SQL Server.

Review the program logic if you use pessimistic locking, because it is based on the NOT WAIT mode which is not supported by SQL Server.

Related concepts

Cursors WITH HOLD on page 631
Programming WITH HOLD cursors using SELECT with and without FOR UPDATE clause.

UPDATE/DELETE ... WHERE CURRENT OF

Informix®

Informix® allows positioned UPDATEs and DELETEs with the "WHERE CURRENT OF cursor" clause, if the cursor has been DECLARED with a SELECT ... FOR UPDATE statement.

Microsoft™ SQL Server

UPDATE/DELETE ... WHERE CURRENT OF is supported with Microsoft™ SQL Server by using server side cursors.

Solution

With Microsoft™ SQL Server, UPDATE/DELETE ... WHERE CURRENT OF instructions are executed as is without any SQL translation: Since BDL cursors for the SELECT FOR UPDATE statements are implemented with ODBC server cursors, native positioned update/delete can take place in SQL Server.
As a replacement of WHERE CURRENT OF, if the database table is defined with a primary key column, use the value fetched from the SELECT FOR UPDATE cursor in the WHERE clause of the UPDATE/DELETE statement.

**Related concepts**

**Positioned UPDATE/DELETE** on page 631
Using positioned updates/deletes with named database cursors.

**LOAD and UNLOAD**

**Informix®**

Informix® provides two SQL instructions to export / import data from / into a database table:

The UNLOAD instruction copies rows from a database table into a text file:

```
UNLOAD TO "filename.unl" SELECT * FROM tab1 WHERE ..
```

The LOAD instructions insert rows from a text file into a database table:

```
LOAD FROM "filename.unl" INSERT INTO tab1
```

**Microsoft™ SQL Server**

Microsoft™ SQL Server has LOAD and UNLOAD instructions, but those commands are related to database backup and recovery. Do not confuse with Informix® commands.

**Solution**

LOAD and UNLOAD instruction are implemented in the Genero BDL runtime system with basic INSERT (for LOAD) or SELECT (for UNLOAD) SQL commands. The LOAD and UNLOAD instruction can be supported with various database servers.

However, LOAD and UNLOAD require the description of the column types in order to work, that can lead to some differences in the data formatting.

**Note:** If no transaction is started, the LOAD instruction will automatically execute a BEGIN WORK and COMMIT WORK when finished, or ROLLBACK WORK if a row insertion failed while loading. Terminating a transaction will automatically close cursors not defined WITH HOLD option. To workaround this situation, see more details in the LOAD on page 736 reference topic.

The LOAD and UNLOAD BDL instructions are supported with Microsoft™ SQL Server with some limitations:

- The LOAD instruction does not work with tables using emulated SERIAL columns because the generated INSERT statement holds the SERIAL column which is actually a IDENTITY column in SQL Server. See the limitations of INSERT statements when using SERIAL types.
- Starting with Microsoft™ SQL Server 2008, Informix® DATETIME data is stored in SQL Server DATETIME2 (n<=5) or TIME (n<=5) columns, depending on the precision of the original DATETIME type:
  - With DATETIME2 (n<=5) columns, the result of LOAD and UNLOAD is equivalent to Informix® DATETIME YEAR TO SECOND or DATETIME YEAR TO FRACTION (n) columns. The data format will be "YYYY-MM-DD hh:mm:ss[.fff...]", where fff... depends on the precision (n) of the DATETIME2 (n) column.
  - With TIME (n) columns, the result of LOAD and UNLOAD is equivalent to Informix® DATETIME HOUR TO SECOND or DATETIME HOUR TO FRACTION (n) columns. The data format will be "hh:mm:ss[.fff...]", where fff... depends on the precision (n) of the TIME (n) column.
- With an Informix® database, simple dates are unloaded with the DBDATE format (ex:"23/12/1998"). Therefore, unloading from an Informix® database for loading into a Microsoft™ SQL Server database is not supported.

**Related concepts**

**LOAD and UNLOAD instructions** on page 639
The LOAD and UNLOAD instructions can produce different data formats depending on the database server type.

**SQL Interruption**

**Informix®**
With Informix®, it is possible to interrupt a long running query if the SQL INTERRUPT ON option.

**Microsoft™ SQL Server**
Microsoft™ SQL Server supports SQL Interruption: The db client must issue an SQLCancel() ODBC call to interrupt a query.

**Solution**
The SQL Server database drivers support SQL interruption and return the Informix® error code -213, when the statement is interrupted.

**Related concepts**
Using SQL interruption on page 601
Interrupt long running SQL queries, or interrupt queries waiting for locked data.

**Scrollable cursors**

**Informix®**
Informix® SQL and Genero BDL support scrollable cursors when you specify the SCROLL clause in the DECLARE cursor instruction:

```sql
DECLARE c1 SCROLL CURSOR FOR SELECT ...
```

**Important:** Informix does not allow to fetch TEXT/BYTE columns with scrollable cursors. If you declare a scroll cursor with a SELECT containing TEXT/BYTE columns, Informix will produce the SQL error -611 when executing the OPEN instruction.

**Microsoft™ SQL Server**
Microsoft™ SQL Server supports native scrollable cursors with the ODBC SQL_SCROLLABLE statement attribute.

**Solution**
All the SQL Server database drivers use the native SQL Server scrollable cursors, by setting the ODBC statement attribute SQL_ATTR_CURSOR_SCROLLABLE to SQL.Scrollable.

**Related concepts**
Scrollable cursors on page 623
How scrollable cursors can be supported on different databases.

**Stored procedure calls**
SQL Server supports stored procedures, as a saved collection of Transact-SQL statements that can take and return user-supplied parameters.
SQL Server stored procedures can also produce one or more result sets.
To create a stored procedure in an SQL Server database, use the CREATE PROCEDURE statement. See reference documentation for more details.
**Stored procedures with output parameters**

SQL Server stored procedures must be called with the input and output parameters specification in the `USING` clause of the `EXECUTE`, `OPEN` or `FOREACH` instruction. As in normal dynamic SQL, parameters must correspond by position and the `IN/OUT/INOUT` options must match the parameter definition of the stored procedure.

To execute the stored procedure, you must use an ODBC call escape sequence:

```sql
PREPARE stmt FROM "({ call proc1(?,?,?) })"
```

Here is a complete example creating and calling a stored procedure with output parameters:

```sql
MAIN
DEFINE n INTEGER
DEFINE d DECIMAL(6,2)
DEFINE c VARCHAR(200)
DATABASE test1
EXECUTE IMMEDIATE
"create procedure proc1"
  @v1 integer,"n"
  @v2 decimal(6,2) output,"n"
  @v3 varchar(20) output"
  as begin
  set @v2 = @v1 + 0.23
  set @v3 = 'Value = ' || cast(@v1 as varchar)"
  end"
PREPARE stmt FROM "({ call proc1(?,?,?) })"
LET n = 111
EXECUTE stmt USING n IN, d OUT, c OUT
DISPLAY d
DISPLAY c
END MAIN
```

**Stored procedures with result set**

With SQL Server, you can execute stored procedures returning a result set. To do so, you must declare a cursor and fetch the rows.

The next example uses a stored procedure with a simple `SELECT` statement. If the stored procedure contains additional Transact-SQL statements such as `SET` or `IF` (which is the case in complex stored procedures), SQL Server generates multiple result sets. By default the Genero SQL Server driver uses "server cursors" to support multiple active SQL statements. But SQL Server stored procedures generating multiple result sets cannot be used with server cursors: The server cursor is silently converted to a "default result set" cursor by the ODBC driver. Since Default result set cursors do not support multiple active statements, you cannot use another SQL statement while processing the results of such stored procedure. You must `CLOSE` the cursor created for the stored procedure before continuing with other SQL statements.

```sql
MAIN
DEFINE i, n INTEGER
DEFINE d DECIMAL(6,2)
DEFINE c VARCHAR(200)
DATABASE test1
CREATE TABLE tab1 ( c1 INTEGER, c2 DECIMAL(6,2), c3 VARCHAR(200) )
INSERT INTO tab1 VALUES ( 1, 123.45, 'aaaaaa' )
INSERT INTO tab1 VALUES ( 2, 123.66, 'bbbbbbbb' )
INSERT INTO tab1 VALUES ( 3, 444.77, 'cccccc' )
EXECUTE IMMEDIATE "create procedure proc2 @key integer"
  as select * from tab1 where c1 > @key"
DECLARE curs CURSOR FROM "({ call proc2(?) })"
LET i = 1
FOREACH curs USING i INTO n, d, c
  DISPLAY n, d, c
```
It is possible to fetch large objects (text/image) from stored procedure generating a result set. However, if the stored procedure executes other statements as the SELECT (like SET/IF commands), the SQL Server ODBC driver will convert the server cursor to a regular default result set cursor, requiring the LOB columns to appear at the end of the select list. Thus, in most cases (stored procedures typically use SET / IF statements), you will have to move the LOB columns and the end of the column list.

*Stored procedures returning a cursor as output parameter*

SQL Server supports "cursor output parameters": A stored procedure can declare/open a cursor and return a reference of the cursor to the caller.

SQL Server stored procedures returning a cursor as output parameter are not supported. There are two reasons for this: The language does not have a data type to store a server cursor reference, and the underlying ODBC driver does not support this anyway.

*Stored procedures with return code*

SQL Server stored procedures can return integer values. To get the return value of a stored procedure, you must use an assignment expression in the ODBC call escape sequence:

```
PREPARE stmt FROM "( ? = call proc3(?,?,?) )"
```

Then the statement can be executed with the EXECUTE instruction, by specifying the output parameter receiving the returned value as first element in the USING list, with the OUT modifier:

```
MAIN
  DEFINE p1, p2, r INTEGER
  CONNECT TO ...
  WHENEVER ERROR CONTINUE
  EXECUTE IMMEDIATE "DROP PROCEDURE p_mul"
  WHENEVER ERROR STOP
  EXECUTE IMMEDIATE "CREATE PROCEDURE p_mul ( @p1 INT, @p2 INT )
  AS BEGIN
  DECLARE @r INT
  SET @r = @p1 * @p2;
  RETURN @r;
  END"
  LET p1 = 15
  LET p2 = 3
  PREPARE s1 FROM "( ? = call p_mul(?,?) )"
  EXECUTE s1 USING r OUT, p1 IN, p2 IN
  DISPLAY "r = ", r
END MAIN
```

*Stored procedures with output parameters, return code and result set*

With SQL Server you can call stored procedures with a return code, output parameters and producing a result set.

Return codes and output parameters are the last items returned to the application by SQL Server; they are not returned until the last row of the result set has been fetched, after the SQLMoreResults() ODBC function is called. If output parameters are used, the SQL Server driver executes a SQLMoreResults() call when closing the cursor instead of SQLCloseCursor(), to get the return code and output parameter values from SQL Server.

```
MAIN
  DEFINE r, i, n INTEGER
  DEFINE d DECIMAL(6,2)
  DEFINE c VARCHAR(200)
  DATABASE test1
  CREATE TABLE tab1 ( c1 INTEGER, c2 DECIMAL(6,2), c3 VARCHAR(200) )
  INSERT INTO tab1 VALUES ( 1, 123.45, 'aaaaaa' )
```
The return code and output parameter variables must be defined as INOUT in the OPEN instruction.

**Oracle® MySQL / MariaDB**

**Supported versions**

Genero BDL supports the following Oracle® MySQL versions:

- Oracle® MySQL 5.6 with dbmmys_5_6
- Oracle® MySQL 5.7 with dbmmys_5_7
- Oracle® MySQL 8.0 with dbmmys_8_0

Genero BDL supports the following MariaDB versions:

- MariaDB 10.2+ with dbmmdb_10_2

**Installation (Runtime Configuration)**

Oracle® MySQL related installation topics.

**Install Oracle® MySQL/MariaDB and create a database - database configuration/design tasks**

If you are tasked with installing and configuring the database, here is a list of steps to be taken:

1. Install the Oracle® MySQL Server (or MariaDB) on your computer.

2. Configure the server with the appropriate storage engine.

   In order to have transaction support by default, you must use a storage engine that supports transactional tables, such as INNODB. In recent versions of MySQL, this is the default storage engine.

3. Consider setting the sql_mode configuration parameter to get the appropriate behavior of the MySQL server:

   a) When the STRICT_TRANS_TABLES option is set in the sql_mode parameter, numeric data truncation/overflow will produce SQL error -1264, and character strings too large for the target column will produce SQL error -1406.

   Without the STRICT_TRANS_TABLES option, MySQL will only produce the SQL warning -1265 (data truncated) when a character string is too large. However, numeric data truncation/overflow is also relaxed and produces warning -1264, when you might expect to get an SQL error and avoid invalid numeric values in your database. Therefore, it is recommended to use the STRICT_TRANS_TABLES option.

   b) Blank padding of fetched CHAR data can be controlled with the PAD_CHAR_TO_FULL_LENGTH option of the sql_mode parameter.

   You can use this parameter to get CHAR values padded with blanks to their full length, but the result of the SQL LENGTH() function will be different since trailing blanks are significant for that function in MySQL.
4. The mysqld process must be started to listen to database client connections. See MySQL documentation for more details about starting the database server process.

5. Create a database user dedicated to your application, the application administrator.
   Connect as the MySQL root user, create the application administrator user and grant all privileges to this user:

   ```bash
   $ mysql -u root -p
   mysql> create user 'mysuser'@'localhost' identified by 'password';
   mysql> grant all privileges on *.* to 'mysuser'@'localhost';
   ```

6. Connect as the application administrator and create a MySQL database with the CREATE DATABASE statement, and specify the character set to be used for this database:

   ```bash
   $ mysql -u mysuser -p
   ...
   mysql> create database mydatabase
       default character set utf8mb4
collate utf8mb4_0900_as_cs;
   ```

   **Note:** MySQL collation names include modifiers such as _ai/_as and _ci/_cs, to indicate if you want accent and character case sensitivity in your database. Consider to use the right modifiers, to have for example WHERE 'é'='É' evaluate to true or false. Use the _bin modifier (as in utf8mb4_bin), to distinguish any character, and when the sort order can be based on the binary value of characters. To check the client charset, database charset and collation:

   ```bash
   mysql> SELECT @@character_set_client, @@character_set_database, @@collation_database;
   ```

7. Create the application tables.

   Convert Informix® data types to MySQL data types. See Data type conversion table: Informix to MySQL on page 882 for more details.

Prepare the runtime environment - connecting to the database

1. Check that the required ODI database driver is available in $FGLDIR/dbdrivers.
   - For Oracle® MySQL 5.5 and 5.6, you must have the MySQL database driver "dbmmys_5_5".
   - For Oracle® MySQL 5.7, you must have the MySQL database driver "dbmmys_5_7".
   - For Oracle® MySQL 8.0, you must have the MySQL database driver "dbmmys"("dbmmys_8_0").
   - For MariaDB 10.0/10.1, you must have the MySQL database driver "dbmmys_5_5".
   - For MariaDB 10.2+, you must have the MariaDB database driver "dbmmdb"("dbmmdb_10_2").

2. Check that the MySQL/MariaDB database client software is available on the system.
   - For MySQL, check that the libmysqlclient.* library is installed on the system. The shared library version of the MySQL client library must match the libmysqlclient library version linked to the dbmmys.so ODI driver.
   - For MariaDB 10.0 and 10.1, check that the libmysqlclient.so.18 library is installed on the system: This is the shared library linked to the dbmmys_5_5.so ODI driver.
   - For MariaDB 10.2 and +, check that the libmariadb.* library is installed on the system. The shared library version of the MariaDB client library must match the libmariadb library version linked to the dbmmdb.so ODI driver.

3. Make sure that the MySQL/MariaDB client environment variables are properly set.
   Check for example MYSQL_HOME (the path to the installation directory), DATADIR (the path to the data files directory), etc. See MySQL documentation for more details about client environment variables to be set.

4. Check the MySQL/MariaDB client configuration options in the my.cnf file on your MySQL client environment.
The driver will read the options defined in the [client] configuration group of the my.cnf file defined for the client application. Note that you can specify a particular configuration file with the following FGLPROFILE parameter:

```
dbi.database.dbname.mys.config = "/opt/var/myapp/my.cnf"
```

or, for MariaDB:

```
dbi.database.dbname.mdb.config = "/opt/var/myapp/my.cnf"
```

5. Check the database client locale settings (default-character-set option in the my.cnf configuration file).

The database client locale must match the locale used by the runtime system (LC_ALL, LANG). For example, in order to define (4-bytes) UTF-8 as MySQL client character set:

```
[client]
default-character-set="utf8mb4"
```

6. Verify the environment variable defining the search path for the database client shared library (libmysqlclient.so on UNIX™, LIBMYSQL.dll on Windows®).

**Table 238: Shared library environment setting for MySQL**

<table>
<thead>
<tr>
<th>MySQL version</th>
<th>Shared library environment setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL 5.5 and higher and MariaDB 10.x</td>
<td>UNIX™: Add $MYSQL_HOME/lib to LD_LIBRARY_PATH (or its equivalent).</td>
</tr>
<tr>
<td></td>
<td>Windows®: Add %MYSQL_HOME%/bin to PATH.</td>
</tr>
</tbody>
</table>

7. To verify if the MySQL/MariaDB client environment is correct, start the SQL command interpreter:

```
$ mysql dbname -u appadmin -p
```

8. Set up the FGLPROFILE entries for database connections.

a) Define the MySQL/MariaDB database driver:

For MySQL:

```
dbi.database.dbname.driver = "dbmmys"
```

For MariaDB:

```
dbi.database.dbname.driver = "dbmmdb"
```

b) The “source” parameter defines the name of the MySQL/MariaDB database.

```
dbi.database.dbname.source = "test1"
```

**Database concepts**

*Oracle® MySQL related database concepts topics.*

**Database concepts**

Like Informix® servers, Oracle® MySQL can handle multiple database entities. Tables created by a user can be accessed without the owner prefix by other users as long as they have access privileges to these tables.

**Tip:** If you have several Informix database entities, migrating from the Informix database to another database it is a good opportunity to centralize all tables in a single database. To avoid conflicts with table names, use a prefix when needed.
Solution

Create a MySQL database for each Informix® database.

Data storage concepts

When converting from Informix® to Oracle® MySQL the aim is to try to preserve as much of the data storage information as possible in the process. The most important storage decisions made for Informix® database objects (like initial sizes and physical placement) can be applied to the MySQL database.

Storage concepts are quite similar in Informix® and in MySQL, but the names are different.

Concurrency management

Data consistency and concurrency concepts

- **Data Consistency** applies to situations when readers want to access data currently being modified by writers.
- **Concurrent Data Access** applies to situations when several writers are accessing the same data for modification.
- **Locking Granularity** defines the amount of data concerned when a lock is set (for example, row, page, table).

Informix®

Informix® uses a locking mechanism to handle data consistency and concurrency. When a process changes database information with UPDATE, INSERT or DELETE, an exclusive lock is set on the touched rows. The lock remains active until the end of the transaction. Statements performed outside a transaction are treated as a transaction containing a single operation and therefore release the locks immediately after execution. SELECT statements can set shared locks, depending on isolation level. In case of locking conflicts (for example, when two processes want to acquire an exclusive lock on the same row for modification, or when a writer is trying to modify data protected by a shared lock), the behavior of a process can be changed by setting the lock wait mode.

Control:

- Lock wait mode: SET LOCK MODE TO ...
- Isolation level: SET ISOLATION TO ...
- Locking granularity: CREATE TABLE ... LOCK MODE {PAGE|ROW}
- Explicit exclusive lock: SELECT ... FOR UPDATE

Defaults:

- The default isolation level is READ COMMITTED.
- The default lock wait mode is NOT WAIT.
- The default locking granularity is PAGE.

Oracle® MySQL

When data is modified, exclusive locks are set and held until the end of the transaction. For data consistency, MySQL uses a locking mechanism. Readers must wait for writers as in Informix®.

Control:

- No lock wait mode control is provided.
- Isolation level: SET TRANSACTION ISOLATION LEVEL ...
- Explicit exclusive lock: SELECT ... FOR UPDATE

Defaults:

- The default isolation level is Read Committed.
- The default locking granularity is per table (per page when using BDB tables).
Solution

The `SET ISOLATION TO ...` Informix® syntax is replaced by `SET SESSION TRANSACTION ISOLATION LEVEL ...` in MySQL. The table shows the isolation level mappings applied by the MySQL database driver:

<table>
<thead>
<tr>
<th>SET ISOLATION instruction in program</th>
<th>Native SQL command</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET ISOLATION TO DIRTY READ</td>
<td><code>SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED</code></td>
</tr>
<tr>
<td>SET ISOLATION TO COMMITTED READ [READ COMMITTED] [RETAIN UPDATE LOCKS]</td>
<td><code>SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED</code></td>
</tr>
<tr>
<td>SET ISOLATION TO CURSOR STABILITY</td>
<td><code>SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED</code></td>
</tr>
<tr>
<td>SET ISOLATION TO REPEATABLE READ</td>
<td><code>SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ</code></td>
</tr>
</tbody>
</table>

For portability, it is recommended that you work with Informix® in the read committed isolation level, make processes wait for each other (lock mode wait), and create tables with the "lock mode row" option.

See Informix® and MySQL documentation for more details about data consistency, concurrency and locking mechanisms.

Related concepts

- **Concurrent data access** on page 599
  Understanding concurrent data access and data consistency.

- **Optimistic locking** on page 624
  Implementing optimistic locking to handle access concurrently to the same database records.

- **Cursors WITH HOLD** on page 631
  Programming WITH HOLD cursors using `SELECT` with and without `FOR UPDATE` clause.

Transactions handling

**Informix®**

With the Informix® native mode (non ANSI):

- Transactions blocks start with `BEGIN WORK` and terminate with `COMMIT WORK` or `ROLLBACK WORK`.
- Statements executed outside a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.

```
UPDATE tab1 SET ...  -- auto-committed
BEGIN WORK            -- start of TX block
UPDATE tab1 SET ...
UPDATE tab2 SET ...
...
COMMIT WORK           -- end of TX block
```

Informix® version 11.50 introduces savepoints:

```
SAVEPOINT name [UNIQUE]
ROLLBACK [WORK] TO SAVEPOINT _name_
RELEASE SAVEPOINT name
```
Oracle® MySQL and MariaDB

- Transactions are started with `START TRANSACTION`.
- Transactions are validated with `COMMIT [WORK]`.
- Transactions are canceled with `ROLLBACK [WORK]`.
- Savepoints can be placed with `SAVEPOINT name`.
- Transactions can be rolled back to a savepoint with `ROLLBACK [WORK] TO [SAVEPOINT] name`.
- Savepoints can be released with `RELEASE SAVEPOINT name`.
- Statements executed outside of a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.

Solution

Informix® transaction handling commands are automatically converted to MySQL instructions to start, validate or cancel transactions.

MySQL does not support transactions by default. You must set the server system parameter `table_type=InnoDB`.

Regarding the transaction control instructions, the BDL applications do not have to be modified in order to work with MySQL, as long as you have a transaction manager installed with MySQL.

If you want to use savepoints, do not use the `UNIQUE` keyword in the savepoint declaration, always specify the savepoint name in `ROLLBACK TO SAVEPOINT`, and do not drop savepoints with `RELEASE SAVEPOINT`.

Related concepts

Database transactions on page 600
Database transactions define a set of SQL instructions to be executed as a whole, or rolled back as a whole.

Database users

Informix®

Until version 11.70.xC2, Informix® database users must be created at the operating system level and must be members of the 'informix' group.

Starting with 11.70.xC2, Informix® supports database-only users with the `CREATE USER` instruction, as in most other db servers.

Any database user must have sufficient privileges to connect and use resources of the database; user rights are defined with the `GRANT` command.

Oracle® MySQL and MariaDB

MySQL users must be registered in the database:

```
$ mysql --host orion -u root -p
mysql> create user 'mysuser'@'localhost' identified by 'password';
mysql> grant all privileges on *.* to 'mysuser'@'localhost';
```

Solution

The solution depends on the application logic, for example, if it is a multi-user application, you have to create one or more MySQL users.

Related concepts

Database users and security on page 608
Properly identifying database users allows to use database security and audit features.

**Data dictionary**

Oracle® MySQL related data dictionary topics.

**Data type conversion table: Informix to MySQL**

**Table 240: Data type conversion table (Informix® to Oracle® MySQL)**

<table>
<thead>
<tr>
<th>Informix® data types</th>
<th>Oracle® MySQL data types</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR(n)</td>
<td>CHAR(n) or TEXT (see note 1)</td>
</tr>
<tr>
<td>VARCHAR(n[,m])</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>LVARCHAR(n)</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>NCHAR(n)</td>
<td>NCHAR(n)</td>
</tr>
<tr>
<td>NVARCHAR(n[,m])</td>
<td>NVARCHAR(n)</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>BIGINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>INT8</td>
<td>BIGINT</td>
</tr>
<tr>
<td>SERIAL{(start)}</td>
<td>INTEGER (see note 2)</td>
</tr>
<tr>
<td>BIGSERIAL{(start)}</td>
<td>BIGINT (see note 2)</td>
</tr>
<tr>
<td>SERIAL8{(start)}</td>
<td>BIGINT (see note 2)</td>
</tr>
<tr>
<td>DOUBLE PRECISION / FLOAT{(n)}</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>REAL / SMALLFLOAT</td>
<td>FLOAT</td>
</tr>
<tr>
<td>DECIMAL(p,s)</td>
<td>DECIMAL(p, s)</td>
</tr>
<tr>
<td>DECIMAL(p) with p&lt;=15</td>
<td>DECIMAL(p*2, p)</td>
</tr>
<tr>
<td>DECIMAL(p) with &gt;15</td>
<td>N/A</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL(32,16) (see note 3)</td>
</tr>
<tr>
<td>MONEY(p,s)</td>
<td>DECIMAL(p, s)</td>
</tr>
<tr>
<td>MONEY(p)</td>
<td>DECIMAL(p, 2)</td>
</tr>
<tr>
<td>MONEY</td>
<td>DECIMAL(16, 2)</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME HOUR TO MINUTE</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME HOUR TO SECOND</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME HOUR TO FRACTION(p)</td>
<td>TIME(p)</td>
</tr>
<tr>
<td>DATETIME YEAR TO MONTH</td>
<td>DATETIME</td>
</tr>
<tr>
<td>DATETIME YEAR TO DAY</td>
<td>DATETIME</td>
</tr>
<tr>
<td>DATETIME YEAR TO HOUR</td>
<td>DATETIME</td>
</tr>
<tr>
<td>DATETIME YEAR TO MINUTE</td>
<td>DATETIME</td>
</tr>
<tr>
<td>DATETIME YEAR TO SECOND</td>
<td>DATETIME</td>
</tr>
<tr>
<td>Informix® data types</td>
<td>Oracle® MySQL data types</td>
</tr>
<tr>
<td>----------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>DATETIME YEAR TO FRACTION (p)</td>
<td>DATETIME (p)</td>
</tr>
<tr>
<td>INTERVAL q1 TO q2</td>
<td>CHAR (50)</td>
</tr>
<tr>
<td>TEXT</td>
<td>MEDIUMTEXT / LONGTEXT (max is 2Gb)</td>
</tr>
<tr>
<td>BYTE</td>
<td>MEDIUMBLOB / LONGBLOB (max is 2Gb)</td>
</tr>
</tbody>
</table>

Notes:

1. The CHAR types with a size > 255 are converted TEXT types. For more details, see CHAR and VARCHAR data types on page 883.
2. For more details about serial emulation, see SERIAL and BIGSERIAL data type on page 891.
3. The DECIMAL (32, 16) type is not supported by MySQL and MariaDB and will produce an SQL error.

Related tasks
Install Oracle MySQL/MariaDB and create a database - database configuration/design tasks on page 876

BOOLEAN data type

Informix®
Informix® supports the BOOLEAN data type, which can store 't' or 'f' values.

Genero BDL implements the BOOLEAN data type in a different way: A BOOLEAN variable stores integer values 1 or 0 (for TRUE or FALSE). This type is designed to hold the result of a boolean expression.

Oracle® MySQL and MariaDB
MySQL supports the BOOLEAN type name as an alias for the built-in TINYINT (1) data type.

Columns defined as BOOLEAN or TINYINT (1) can store 1 or 0 integer values for TRUE and FALSE.

Solution
The MySQL and MariaDB database interfaces support the BOOLEAN data type and stores 1 or 0 values in the column.

The BOOLEAN type translation can be controlled with the following FGLPROFILE entry:

```
  dbi.database.dsnname.ifxemul.datatype.boolean = \true \false
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

Related concepts
Using portable data types on page 614
Only a limited set of data types are really portable across several database engines.

CHAR and VARCHAR data types

Informix®
Informix® supports the following character data types:

Table 241: Informix® character data types

<table>
<thead>
<tr>
<th>Informix® data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR (n)</td>
<td>SBCS and MBCS character data (max is 32767 bytes)</td>
</tr>
</tbody>
</table>
Informix® data type | Description
---|---
VARCHAR\((n, m)\) | SBCS and MBCS character data (max is 255 bytes)
NCHAR\((n)\) | Same as CHAR, with specific collation order
NVARCHAR\((n, m)\) | Same as VARCHAR, with specific collation order
LVARCHAR\((n)\) | Max size varies depending on the IDS version

With Informix®, both CHAR/VARCHAR and NCHAR/NVARCHAR data types can be used to store single-byte or multibyte encoded character strings. The only difference between CHAR/VARCHAR and NCHAR/NVARCHAR is in how they use sorting: N[VAR]CHAR types use the collation order, while [VAR]CHAR types use the byte order.

The character set used to store strings in CHAR/VARCHAR/NCHAR/NVARCHAR columns is defined by the DB_LOCALE environment variable.

The character set used by applications is defined by the CLIENT_LOCALE environment variable.

Informix® uses Byte Length Semantics (the size N that you specify in [VAR]CHAR\(N\) is expressed in bytes, not characters as in some other databases)

**Oracle® MySQL and MariaDB**

MySQL and MariaDB support the following data types to store character data:

**Table 242: MySQL character data types**

<table>
<thead>
<tr>
<th>MySQL data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR((n))</td>
<td>SBCS or MBCS character data using the database character set, where n is specified in characters (max is 255 characters)</td>
</tr>
<tr>
<td>VARCHAR((n))</td>
<td>SBCS or MBCS character data using the database character set, where n is specified in characters (max is 65535 characters)</td>
</tr>
<tr>
<td>NCHAR((n))</td>
<td>SBCS or MBCS character data using the national character set, where n is specified in characters (max is 255 characters)</td>
</tr>
<tr>
<td>NVARCHAR((n))</td>
<td>SBCS or MBCS character data using the national character set, where n is specified in characters (max is 65535 characters)</td>
</tr>
<tr>
<td>TEXT</td>
<td>The Large Object type to store SBCS or MBCS character data using the database character set</td>
</tr>
</tbody>
</table>

MySQL uses character length semantics to define the size of CHAR/VARCHAR columns, while Informix® and Genero use Byte Length Semantics.

MySQL can support multiple character sets, you can run the `SHOW CHARACTER SET` statement to list supported encoding. There are different configuration levels to define the character set used by MySQL to store data. The server `character_set` defines the default for database character sets if not specified in the `CREATE DATABASE` command. You can even define a specific character set at the table and column level, but this is not recommended with Genero applications. The database character set is used to store CHAR and VARCHAR columns. The NCHAR and NATIONAL VARCHAR types use a predefined character set which can be different from the database character set. In MySQL the national character set is UTF-8.

MySQL can automatically convert from/to the client and server characters sets. In the client applications, the character set can be defined with the `default-character-set` configuration option, or with the `SET NAMES` SQL instruction (but that instruction is not supported with Genero see below).

**Important:** If the `STRICT_TRANS_TABLES` option is not defined in the `sql_mode` parameter, MySQL truncates character strings, when the value is too large for the target column. However, the `STRICT_TRANS_TABLES` option controls also numeric data truncation/overflow. This option should be used, to avoid numeric data truncation/overflows.
overflow being ignored (with only an SQL warning), and to produce an SQL error instead when the numeric value does not fit into the target column type.

Note that by default, when fetching CHAR columns from MySQL, trailing blanks are trimmed. This does not matter as long as you fetch CHAR columns into CHAR variables, but this non-standard behavior will impact CHAR fetch into VARCHAR, or other SQL areas such as string concatenation for example. You can control the behavior of CHAR trailing blanks trimming with the PAD_CHAR_TO_FULL_LENGTH option of the sql_mode parameter. But when this mode is used, the result of the SQL LENGTH() function will be different since trailing blanks are significant for that function in MySQL.

Solution

Informix® CHAR (N) types must be mapped to MySQL CHAR (N) types. Informix® VARCHAR (N) or LVARCHAR (N) types must be mapped to MySQL VARCHAR (N).

You can store single-byte or multibyte character strings in MySQL CHAR, VARCHAR and TEXT columns.

Define the DB client application character set, to make it correspond to the runtime system character set.

Important: MySQL/MariaDB support the SET NAMES SQL instruction, to change the character set after connecting to the database. However, this is not supported with Genero: The driver needs to know the character set at connection initialization. Use the default-character-set configuration option.

With MySQL/MariaDB, the client application character set must be defined with the default-character-set option, under the [client] section of the configuration file, for example:

```
[client]
default-character-set="utf8mb4"
```

Note: A specific MySQL/MariaDB configuration file can be defined with an FGLPROFILE entry, see Oracle MySQL specific FGLPROFILE parameters on page 675, MariaDB specific FGLPROFILE parameters on page 675.

MySQL uses character length semantics: When you define a CHAR(20) and the database character set is multibyte, the column can hold more bytes/characters than the Informix® CHAR(20) type, when using byte length semantics. When using a multibyte character set (such as UTF-8), define database columns with the size in character units, and use character length semantics in BDL programs with FGL_LENGTH_SEMANTICS=CHAR.

When extracting a database schema from a MySQL database, the fgldbsch schema extractor uses the size of the column in characters, not the octet length. If you have created a CHAR(10 (characters) ) column in a MySQL database using the UTF8 character set, the .sch file will get a size of 10, which will be interpreted depending on FGL_LENGTH_SEMANTICS as a number of bytes or characters.

Review your database schema when using CHAR columns with a size exceeding the MySQL limits: If you need to store CHAR character strings larger as the MySQL CHAR limit, you can use the MySQL TEXT type. However, as of MySQL version 5.0.3 (supporting large VARCHAR sizes), as long as you use short sizes for CHAR (<100c), the character types can be used as is in MySQL.

The CHAR (N>255) types are converted by the SQL Translator to a MySQL TEXT type, because MySQL CHAR type has a limit of 255 characters. When designing a database, consider using CHAR only for short character string data storage (<50c), and use VARCHAR for larger character string data storage (name, address, comments).

Note: For each TEXT column fetched from MySQL, the MySQL database driver needs to allocate a temporary string buffer of 65535 bytes. The memory used by this temporary buffer is freed when freeing the cursor.

When using VARCHAR types, the SQL Translator leaves the type definition as is, even for N > 255, assuming that the target MySQL server version is at least 5.0.3 (supporting VARCHAR (N) up to 65535 characters).

See also the section about Localization.

The CHAR/VARCHAR type translation can be controlled with the following FGLPROFILE entries:

```
dbi.database.dsnname.ifxemul.datatype.char = { true | false }
```
For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

**CHAR and VARCHAR types** on page 616
Using the CHAR and VARCHAR data types with different databases.

**Numeric data types**

**Informix**

Informix® supports several data types to store numbers:

**Table 243: Informix® numeric data types**

<table>
<thead>
<tr>
<th>Informix® data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>16 bit signed integer</td>
</tr>
<tr>
<td>INTEGER</td>
<td>32 bit signed integer</td>
</tr>
<tr>
<td>BIGINT</td>
<td>64 bit signed integer</td>
</tr>
<tr>
<td>INT8</td>
<td>64 bit signed integer (replaced by BIGINT)</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>Equivalent to DECIMAL(16)</td>
</tr>
<tr>
<td>DECIMAL(p)</td>
<td>Floating-point decimal number (max precision is 32)</td>
</tr>
<tr>
<td>DECIMAL(p, s)</td>
<td>Fixed-point decimal number (max precision is 32)</td>
</tr>
<tr>
<td>MONEY</td>
<td>Equivalent to DECIMAL(16,2)</td>
</tr>
<tr>
<td>MONEY(p)</td>
<td>Equivalent to DECIMAL(p,2) (max precision is 32)</td>
</tr>
<tr>
<td>MONEY(p, s)</td>
<td>Equivalent to DECIMAL(p,s) (max precision is 32)</td>
</tr>
<tr>
<td>REAL / SMALLFLOAT</td>
<td>32-bit floating point decimal (C float)</td>
</tr>
<tr>
<td>DOUBLE PRECISION / FLOAT[n]</td>
<td>64-bit floating point decimal (C double)</td>
</tr>
</tbody>
</table>

**Oracle® MySQL and MariaDB**

MySQL and MariaDB support the following data types to store numbers:

**Table 244: MySQL numeric data types**

<table>
<thead>
<tr>
<th>MySQL data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>16 bit signed integer</td>
</tr>
<tr>
<td>INTEGER</td>
<td>32 bit signed integer</td>
</tr>
<tr>
<td>BIGINT</td>
<td>64 bit signed integer</td>
</tr>
<tr>
<td>DECIMAL(p, s)</td>
<td>Fixed point decimal. Maximum precision depends on MySQL Version, see documentation</td>
</tr>
<tr>
<td>DECIMAL(p)</td>
<td>Stores whole numeric numbers up to p digits</td>
</tr>
<tr>
<td>FLOAT(M,D)</td>
<td>32 bit floating point number</td>
</tr>
<tr>
<td>DOUBLE(M,D)</td>
<td>64 bit floating point number</td>
</tr>
</tbody>
</table>
The `STRICT_TRANS_TABLES` option in the `sql_mode` parameter defines if numeric data truncation/overflow should produce an SQL error -1264, or just an SQL warning -1265. To avoid problems you want to use the `STRICT_TRANS_TABLES` option.

**Note:** Before MySQL 5.0.3, the maximum range of `DECIMAL` values is the same as for `DOUBLE`. Since MySQL 5.0.3, `DECIMAL` can store real precision numbers as in Informix®. However, the maximum number of digits depends on the version of MySQL, see documentation for more details. We strongly recommend that you make tests (INSERT + SELECT) to check whether large decimals are properly inserted and fetched back.

**Solution**

Use the following conversion rules to map Informix numeric types to MySQL numeric types:

**Table 245: Informix® numeric data types and MySQL equivalents**

<table>
<thead>
<tr>
<th>Informix® data type</th>
<th>MySQL equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>INT8 / BIGINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>DECIMAL(p&lt;=16)</td>
<td>DECFLOAT(16)</td>
</tr>
<tr>
<td>DECIMAL(p&gt;16)</td>
<td>DECFLOAT(34)</td>
</tr>
<tr>
<td>DECIMAL(p&lt;=31,s)</td>
<td>DECIMAL(p,s)</td>
</tr>
<tr>
<td>DECIMAL(32,s)</td>
<td>No equivalent</td>
</tr>
<tr>
<td>MONEY</td>
<td>DECIMAL(16,2)</td>
</tr>
<tr>
<td>MONEY(p)</td>
<td>DECIMAL(p,2)</td>
</tr>
<tr>
<td>MONEY(p,s)</td>
<td>DECIMAL(p,s)</td>
</tr>
<tr>
<td>SMALLFLOAT</td>
<td>REAL</td>
</tr>
<tr>
<td>FLOAT[(n)]</td>
<td>FLOAT[(n)] (DOUBLE)</td>
</tr>
</tbody>
</table>

**Important:** If the `STRICT_TRANS_TABLES` option is not defined in the `sql_mode` parameter, MySQL truncates character strings, when the value is too large for the target column. However, the `STRICT_TRANS_TABLES` option controls also numeric data truncation/overflow. This option should be used, to avoid numeric data truncation/overflow being ignored (with only an SQL warning), and to produce an SQL error instead when the numeric value does not fit into the target column type.

The numeric types translation can be controlled with the following FGLPROFILE entries:

```
| dbi.database.dsname.ifxemul.datatype.smallint = { true | false }
| dbi.database.dsname.ifxemul.datatype.integer = { true | false }
| dbi.database.dsname.ifxemul.datatype.bigint = { true | false }
| dbi.database.dsname.ifxemul.datatype.int8 = { true | false }
| dbi.database.dsname.ifxemul.datatype.decimal = { true | false }
| dbi.database.dsname.ifxemul.datatype.money = { true | false }
| dbi.database.dsname.ifxemul.datatype.float = { true | false }
| dbi.database.dsname.ifxemul.datatype.smallfloat = { true | false }
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

Using portable data types on page 614
Only a limited set of data types are really portable across several database engines.

**DATE and DATETIME data types**

**Informix®**
Informix® provides two data types to store date and time information:

- **DATE** = for year, month and day storage.
- **DATETIME** = for year to fraction (1-5) storage.

The **DATE** type is stored as an **INTEGER** with the number of days since 1899/12/31.

The **DATETIME** type can be defined with various time units, by specifying a start and end qualifier. For example, you can define a datetime to store an hour-to-second time value with **DATETIME HOUR TO SECOND**.

The values of Informix® **DATETIME** can be represented with a character string literal, or as **DATETIME()** literals:

```
'2017-12-24 15:45:12.345'  -- a DATETIME YEAR TO FRACTION(3)
'15:45'   -- a DATETIME HOUR TO MINUTE
DATETIME(2017-12-24 12:45) YEAR TO MINUTE
DATETIME(12:45:56.333) HOUR TO FRACTION(3)
```

Informix® is able to convert quoted strings to **DATE**/**DATETIME** data, if the string contains matching environment parameters. The string to date conversion rules for **DATE** is defined by the **DBDATE** environment variable. The string to datetime format for **DATETIME** is defined by the **GL_DATETIME** environment variable.

**Note:** Within Genero programs, the string representation for **DATETIME** values is always ISO (YYYY-MM-DD hh:mm:ss.fffffff)

Informix® supports date arithmetic on **DATE** and **DATETIME** values. The result of an arithmetic expression involving dates/times is an **INTEGER** number of days when only **DATE** values are used, and an **INTERVAL** value if a **DATETIME** is used in the expression.

Informix® automatically converts an **INTEGER** to a **DATE** when the integer is used to set a value of a date column.

Informix® provides the **CURRENT [ q1 TO q2 ]** operator, to get the system date/time on the server where the current database is located. When no qualifiers are specified, **CURRENT** returns a **DATETIME YEAR TO FRACTION(3)**. Informix also supports the **SYSDATE** operator, which returns the current system time as a **DATETIME YEAR TO FRACTION(5)**.

**Note:** The **USEOSTIME** configuration parameter must be set to 1 in order to get the subsecond precision in **CURRENT** and **SYSDATE** operators. See Informix documentation for more details.

**Oracle® MySQL and MariaDB**

MySQL and MariaDB provide the following data type to store date and time data:

**Table 246: MySQL date/time data types**

<table>
<thead>
<tr>
<th>MySQL data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>for year, month, day storage.</td>
</tr>
<tr>
<td>TIME(n)</td>
<td>for hour, minute, second and fraction of second storage.</td>
</tr>
<tr>
<td>DATETIME(n)</td>
<td>for year, month, day, hour, minute, second and fraction of second storage.</td>
</tr>
</tbody>
</table>

**Note:** The range for MySQL **DATETIME** values is 1000-01-01 00:00:00 to 9999-12-31 23:59:59.999999.
**MySQL data type** | **Description**
---|---
TIMESTAMP | Similar to DATETIME, but with smaller value range.

Important: The range for MySQL TIMESTAMP values is 1970-01-01 00:00:01.000000 to 2038-01-19 03:14:07.999999.

Like Informix®, MySQL can convert quoted strings to datetime data based on the ISO datetime format (YYYY-MM-DD hh:mm:ss).

In MySQL, the result of an arithmetic expression involving DATE values is an INTEGER representing a number of days.

Important: When the ALLOW_INVALID_DATES option is set in the sql_mode parameter, the database engine will allow invalid date values to be inserted, resulting as 0000-00-00 values in the table. It is not recommended to use this option.

**Solution**

Use the following conversion rules to map Informix® date/time types to MySQL date/time types:

<table>
<thead>
<tr>
<th>Informix® data type</th>
<th>MySQL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME HOUR TO MINUTE</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME HOUR TO SECOND</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME HOUR TO FRACTION(p)</td>
<td>TIME(p)</td>
</tr>
<tr>
<td>DATETIME YEAR TO MONTH</td>
<td>DATETIME</td>
</tr>
<tr>
<td>DATETIME YEAR TO DAY</td>
<td>DATETIME</td>
</tr>
<tr>
<td>DATETIME YEAR TO HOUR</td>
<td>DATETIME</td>
</tr>
<tr>
<td>DATETIME YEAR TO MINUTE</td>
<td>DATETIME</td>
</tr>
<tr>
<td>DATETIME YEAR TO SECOND</td>
<td>DATETIME</td>
</tr>
<tr>
<td>DATETIME YEAR TO FRACTION(p)</td>
<td>DATETIME(p)</td>
</tr>
</tbody>
</table>

The DATE and DATETIME types translation can be controlled with the following FGLPROFILE entries:

```sql
    dbi.database.dsnname.ifxemul.datatype.date = 1 true 1 false 1
    dbi.database.dsnname.ifxemul.datatype.datetime = 1 true 1 false 1
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

MySQL and Informix® DATE data types both store year, month, day values.

**Note:** With the ALLOW_INVALID_DATES option in sql_mode parameter, invalid date values can be inserted. Fetching an invalid date value (0000-00-00) will result as NULL in the target DATE BDL variable.

MySQL TIME[(n)] data type can be used to store Informix® DATETIME HOUR TO FRACTION(n), DATETIME HOUR TO SECOND, DATETIME HOUR TO MINUTE values, and any other DATETIME type with qualifiers HOUR, MINUTE, SECOND and FRACTION(n). Missing time parts default to 00:00:00.0. For example, when using a DATETIME MINUTE TO FRACTION(3) with the value of "45:23.999", the MySQL TIME(3) value will be "00:45:23.999".

Informix® DATETIME values with any qualifiers from YEAR to FRACTION(5) can be stored in MySQL DATETIME[(n)] columns. Missing date or time parts default to 1900-01-01 00:00:00.0. For example, when using
a DATETIME DAY TO MINUTE with the value of "23 11:45", the MySQL DATETIME value will be "1900-01-23 11:45:00".

**Important:** MySQL version older than 5.6.4 do not support fractional part in TIME and DATETIME. If you try to store a DATETIME q1 TO FRACTION (p) with such an old server version, the fractional part is lost.

**Related concepts**

- Date/time literals in SQL statements on page 635
- Good practices for date and time handling in SQL.

**INTERVAL data type**

**Informix®**

Informix® provides the INTERVAL data type to store a value that represents a span of time.

INTERVAL types are divided into two classes:

- **year-month** intervals. For example: INTERVAL YEAR(5) TO MONTH
- **day-time** intervals. For example: INTERVAL DAY(9) TO SECOND

INTERVAL columns can be defined with various time units, by specifying a start and end qualifier. For example, you can define an interval to store a number of hours and minutes with INTERVAL HOUR (n) TO MINUTE, where n defines the maximum number of digits for the hours unit.

The values of Informix® INTERVAL can be represented with a character string literal, or as INTERVAL() literals:

```
'1-9834 15:45:12.345' -- an INTERVAL DAY(6) TO FRACTION(3)
'7623-11'   -- an INTERVAL YEAR(9) TO MONTH
INTERVAL(18734:45) HOUR(5) TO MINUTE
INTERVAL(-7634-11) YEAR(5) TO MONTH
```

**Oracle® MySQL and MariaDB**

MySQL and Mariadb support an INTERVAL data type. However, the interval values are used mainly for date/time calculations like:

```
SELECT NOW() + INTERVAL 5 MINUTE
```

It is not possible to create a MySQL/Mariadb table column with an interval type.

**Solution**

The INTERVAL data type and values are converted CHAR (50) column with MySQL.

INTERVAL values can be stored and retrieved from the database. However, interval arithmetics cannot be performed on the database side in SQL statements.

The INTERVAL types translation can be controlled with the following FGLPROFILE entry:

```
dbi.database.dsname.ifxemul.datatype.interval = ↓ true ↓ false ↓
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

- Using portable data types on page 614
Only a limited set of data types are really portable across several database engines.

**SERIAL and BIGSERIAL data type**

**Informix®**
Informix® supports the SERIAL, BIGSERIAL data types to produce automatic integer sequences:

- SERIAL can produce 32 bit integers (INTEGER)
- BIGSERIAL can produced 64 bit integers (BIGINT)
- SERIAL8 is a synonym for BIGSERIAL

Steps to use serials with Informix®:
1. Create the table with a column using SERIAL, or BIGSERIAL.
2. To generate a new serial, no value or a zero value is specified in the INSERT statement:

   ```sql
   INSERT INTO tab1 ( c ) VALUES ( 'aa' )
   INSERT INTO tab1 ( k, c ) VALUES ( 0, 'aa' )
   ```
3. After INSERT, the new value of a SERIAL column is provided in SQLCA.SQLERRD[2], while the new value of a BIGSERIAL value must be fetched with a SELECT dbinfo('bigserial') query.

Informix® allows you to insert rows with a value different from zero for a serial column. Using an explicit value will automatically increment the internal serial counter, to avoid conflicts with future INSERT statements that are using a zero value:

```sql
CREATE TABLE tab ( k SERIAL); -- internal counter = 0
INSERT INTO tab VALUES ( 0 ); -- internal counter = 1
INSERT INTO tab VALUES ( 10 ); -- internal counter = 10
INSERT INTO tab VALUES ( 0 ); -- internal counter = 11
DELETE FROM tab; -- internal counter = 11
INSERT INTO tab VALUES ( 0 ); -- internal counter = 12
```

**Oracle® MySQL and MariaDB**
MySQL supports the AUTO_INCREMENT column definition option as well as the SERIAL keyword:

- In CREATE TABLE, you specify a auto-incremented column with the AUTO_INCREMENT attribute
- Auto-incremented columns have the same behavior as Informix® SERIAL columns
- A start value can be defined with ALTER TABLE tabname AUTO_INCREMENT = value
- The column must be the primary key, or the first column of an index.
- When using the InnoDB engine, with MySQL 5.7 and earlier, auto-incremented columns might reuse unused sequences after a server restart. For example, if you insert rows that generate the numbers 101, 102 and 103, then you delete rows 102 and 103; When the server is restarted, next generated number will be 101 + 1 = 102. Starting with MySQL 8.0, the last auto-incremented value is written to the disk and persists across server restarts.
- SERIAL is a synonym for BIGINT UNSIGNED NOT NULL AUTO_INCREMENT UNIQUE.

**Solution**

*Note:* For best SQL portability when using different types of databases, consider using sequences as described in [Solution 3: Use native SEQUENCE database objects](#) on page 628.

The Informix® SERIAL data type is emulated with MySQL AUTO_INCREMENT option.

The serial types emulation can be enabled or disabled with the following FGLPROFILE entries:

```sql
$dbi.database.dbname.ifxemul.datatype.serial = [true|false]
$dbi.database.dbname.ifxemul.datatype.serial8 = [true|false]
$dbi.database.dbname.ifxemul.datatype.bigserial = [true|false]
```
Disabling automatic serial retrieval for SQLCA.SQLERRD[2]

SERIAL emulation can be totally disabled by setting the `ifxemul.datatype.serial` FGLPROFILE entry to false:

```
dbi.database.dbname.ifxemul.datatype.serial = false
```

For Informix® compatibility, after an INSERT statement, the ODI drivers automatically execute another SQL query (or do a DB client API call when possible), to get the last generated serial, and fill the `SQLCA.SQLERRD[2]` register. This results in some overhead that can be avoided, if the `SQLCA.SQLERRD[2]` register is not used by the program.

When SERIAL emulation is required (to create temp tables with a serial column during program execution), and the `SQLCA.SQLERRD[2]` register does not need to be filled, (typically because you use your own method to retrieve the last generated serial), you can set the `ifxemul.datatype.serial.sqlerrd2` FGLPROFILE entry to false. This will avoid the automatic retrieval of last serial value to fill `SQLCA.SQLERRD[2]`:

```
dbi.database.dbname.ifxemul.datatype.serial.sqlerrd2 = false
```

See also `db_get_last_serial()` on page 2308.

Using the native serial emulation (only option)

After an insert, `SQLCA.SQLERRD[2]` holds the last generated serial value. However, `SQLCA.SQLERRD[2]` is defined as an INTEGER, it cannot hold values from BIGINT auto incremented columns. If you are using BIGINT auto incremented columns, you must use the `LAST_INSERT_ID()` SQL function.

**AUTO_INCREMENT** columns must be primary keys. This is handled automatically when you create a table in a BDL program.

Like Informix®, MySQL allows you to specify a zero for auto-incremented columns. However, for SQL portability, it is recommended to review INSERT statements to remove the SERIAL column from the list.

For example, the following statement:

```
INSERT INTO tab (col1,col2) VALUES ( 0, p_value)
```

can be converted to:

```
INSERT INTO tab (col2) VALUES (p_value)
```

Static SQL INSERT using records defined from the schema file must also be reviewed:

```
DEFINE rec LIKE tab.*
INSERT INTO tab VALUES ( rec.* ) -- will use the serial column
```

can be converted to:

```
INSERT INTO tab VALUES rec.* -- without parentheses, serial column is removed
```

Related concepts

- Auto-incremented columns (serials) on page 626
How to implement automatic record keys.

**ROWID columns**

**Informix®**

When creating a table, Informix® automatically adds a **ROWID** integer column (applies to non-fragmented tables only).

The **ROWID** column is auto-filled with a unique number and can be used like a primary key to access a given row.

**Note:** Informix® **ROWID** usage was a common practice in the early days of Informix® 4GL programming. Today it is recommended to define all your database tables with a **PRIMARY KEY** to uniquely identify rows.

With Informix®, the SQLCA.SQLERRD[6] register contains the **ROWID** of the last row affected by an **INSERT**, **UPDATE** or **DELETE** statement.

**Oracle® MySQL and MariaDB**

MySQL and Mariadb do not support **ROWIDs**.

**Solution**

If your Genero BDL application uses rowid columns, review the program logic to use primary keys instead. If the database table does not define a primary key, it should be added. All references to SQLCA.SQLERRD[6] must be removed, because this variable will not hold the **ROWID** of the last modified row.

For databases where the keyword of the rowid pseudo-column is different than "**ROWID**", the translation can be controlled with the following FGLPROFILE entry:

```plaintext
dbi.database.dsnname.ifxemul.rowid = \true \false
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

- **Using ROWID columns** on page 644
- Automatic ROWID columns is not a common database feature.

**TEXT and BYTE (LOB) types**

**Informix®**

Informix® provides the **TEXT**, **BYTE**, **CLOB** and **BLOB** data types to store very large texts or binary data.

Legacy Informix® 4GL applications typically use the **TEXT** and **BYTE** types.

Genero BDL does not support the Informix® **CLOB** and **BLOB** types.

**Oracle® MySQL and MariaDB**

MySQL provides the following types to store large objects in the database:

**Table 248: MySQL and MariaDB LOB data types**

<table>
<thead>
<tr>
<th>MySQL data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TINYTEXT</td>
<td>To store text data with a length &lt; 2^8 bytes</td>
</tr>
<tr>
<td>TEXT</td>
<td>To store text data with a length &lt; 2^16 bytes</td>
</tr>
<tr>
<td>MEDIUMTEXT</td>
<td>To store text data with a length &lt; 2^24 bytes</td>
</tr>
<tr>
<td>LONGTEXT</td>
<td>To store text data with a length &lt; 2^32 bytes</td>
</tr>
<tr>
<td>MySQL data type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>TINYBLOB</td>
<td>To store binary data with a length &lt; $2^8$ bytes</td>
</tr>
<tr>
<td>BLOB</td>
<td>To store binary data with a length &lt; $2^{16}$ bytes</td>
</tr>
<tr>
<td>MEIDUMBLOB</td>
<td>To store binary data with a length &lt; $2^{24}$ bytes</td>
</tr>
<tr>
<td>LONGBLOB</td>
<td>To store binary data with a length &lt; $2^{32}$ bytes</td>
</tr>
</tbody>
</table>

**Solution**

The MySQL and MariaDB database interface can convert BDL TEXT data to LONGTEXT and BYTE data to LONGBLOB.

The TEXT and BYTE types translation can be controlled with the following FGLPROFILE entries:

```plaintext
dbi.database.dsname.ifxemul.text = [true | false]
dbi.database.dsname.ifxemul.byte = [true | false]
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

Genero TEXT/BYTE program variables have a limit of 2 gigabytes, make sure that the large object data does not exceed this limit.

**Note:** Because MySQL/MariaDB CHAR and VARCHAR cannot exceed 255 bytes, we recommend that you use the MySQL TEXT type to store CHAR/VARCHAR values with a size larger than 255 bytes. When fetching TEXT columns from a MySQL database, these will be treated as CHAR/VARCHAR types by the MySQL database driver. See CHAR/VARCHAR types for more details.

**Related concepts**

Using portable data types on page 614

Only a limited set of data types are really portable across several database engines.

**Table constraints**

**Informix®**

Informix® supports primary key, unique, foreign key, default and check constraints.

The constraint naming syntax is different in Informix® and most other databases: Informix expects the constraint name after the constraint definition:

```sql
CREATE TABLE emp (  
  ...  
  emp_code CHAR(10) UNIQUE CONSTRAINT pk_emp,  
  ...  
)
```

While other SQL database brands require to specify the constraint name before the constraint definition:

```sql
CREATE TABLE emp (  
  ...  
  emp_code CHAR(10) CONSTRAINT pk_emp UNIQUE,  
  ...  
)
```

**Oracle® MySQL and MariaDB**

MySQL and Mariadb support primary key, unique, foreign key and default constraints.
**Important:** MySQL and MariaDB do not support CHECK constraints. In fact, the syntax is allowed but the constraint is ignored.

**Constraint naming syntax**

The constraint naming clause must be placed before the constraint specification.

**Primary keys**

MySQL creates an index to enforce PRIMARY KEY constraints (some RDBMS do not create indexes for constraints). Using CREATE UNIQUE INDEX to define unique constraints is obsolete (use primary keys or a secondary key instead).

In MySQL, the name of a PRIMARY KEY is PRIMARY.

**Unique constraints**

Like Informix®, MySQL creates an index to enforce UNIQUE constraints (some RDBMS do not create indexes for constraints).

When using a unique constraint, Informix® allows only one row with a NULL value, while MySQL allows several rows with NULL! Using CREATE UNIQUE INDEX is obsolete.

**Foreign keys**

Both Informix® and MySQL support the ON DELETE CASCADE option. In MySQL, foreign key constraints are checked immediately, so NO ACTION and RESTRICT are the same.

**Check constraints**

Check constraints are not yet supported in MySQL.

**Solution**

**Constraint naming syntax**

The database interface does not convert constraint naming expressions when creating tables from BDL programs. Review the database creation scripts to adapt the constraint naming clauses for MySQL.

If your application tables use CHECK constraints, you need to implement these constraints with triggers.

**Related concepts**

- [Data definition statements](#) on page 613
- It is recommended to avoid use of DDL in programs.

**Name resolution of SQL objects**

**Informix®**

Informix® uses the following form to identify an SQL object:

```
database[@dbservername]:][{owner|"owner"}].identifier
```

The ANSI convention is to use double quotes for identifier delimiters (For example: "customer"."cust_name").

Informix® database object names are not case-sensitive in non-ANSI databases. When using double-quoted identifiers, Informix® becomes case sensitive.
With non-ANSI Informix® databases, you do not have to give a schema name before the tables when executing an SQL statement:

```
SELECT ... FROM customer WHERE ...
```

In Informix® ANSI compliant databases:

- The table name must include "owner", unless the connected user is the owner of the database object.
- The database server shifts the owner name to uppercase letters before the statement executes, unless the owner name is enclosed in double quotes.

Oracle® MySQL and MariaDB

With MySQL and MariaDB, an object name takes the following form:

```
[database.]identifier
```

**Solution**

To write portable SQL, regarding database object names:

1. Use simple database object names (without any owner/schema prefix)
2. Do not use double quotes to surround database object identifiers.
3. If needed, define public synonyms to reference database objects in others databases/schema.
4. Specify database object identifiers in lowercase.

See also Naming database objects on page 637.

**Data manipulation**

Oracle® MySQL related data manipulation topics.

**Reserved words**

**Informix®**

With Informix®, it is possible to create database objects with reserved words.

For example:

```
CREATE TABLE table ( char CHAR(10) );
```

Indeed this is not good practice, but Informix® SQL allows this to be backward compatible when introducing a new keyword in the SQL syntax.

Most other database systems do not allow reserved words as database identifiers. If your legacy code is using SQL reserved words of the target database SQL syntax, an error will be thrown at CREATE TABLE execution.

**Oracle® MySQL and MariaDB**

SQL object names like table and column names cannot be SQL reserved words in MySQL.

**Solution**

Table or column names which are MySQL reserved words must be renamed.
Outer joins

Informix® OUTER() syntax

In Informix® SQL, outer joins can be defined in the FROM clause with the OUTER keyword:

```
SELECT ... FROM a, OUTER (b)
    WHERE a.key = b.akey

SELECT ... FROM a, OUTER(b,OUTER(c))
    WHERE a.key = b.akey
    AND b.key1 = c.bkey1 AND b.key2 = c.bkey2
```

Informix® also supports the ANSI OUTER join syntax, which is the recommended way to specify outer joins with recent SQL database engines:

```
SELECT ... FROM cust LEFT OUTER JOIN order
    ON cust.key = order.custno
    WHERE ...
```

Oracle® MySQL and MariaDB

MySQL and MariaDB support the ANSI outer join syntax:

```
SELECT ...
    FROM cust LEFT OUTER JOIN order
        LEFT OUTER JOIN item
            ON order.key = item.ordno
            ON cust.key = order.custno
    WHERE order.cdate > current date
```

Solution

The Genero database drivers can convert Informix Informix® OUTER specifications to ANSI outer joins.

**Note:** For better SQL portability, use the ANSI outer join syntax instead of the old Informix® OUTER syntax.

The outer join translation can be controlled with the following FGLPROFILE entry:

```
    dbi.database.dsnname.ifxemulouters = \t true \t false \n```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

- **Prerequisites:**
  
  1. In the FROM clause, the main table must be the first item and the outer tables must be listed from left to right in the order of outer levels.

    Example which does not work:

    ```
    ... FROM OUTER(tab2), tab1
    ```

  2. The outer join in the WHERE clause must use the table name as prefix:

    ```
    ... WHERE tab1.col1 = tab2.col2
    ```

- **Restrictions:**

  1. Statements composed by 2 or more SELECT instructions are not supported:

    ```
    SELECT ... UNION SELECT ...
    ```
or:

```
SELECT ... WHERE col IN (SELECT...)
```

2. Additional conditions on outer table columns cannot be detected and therefore are not supported:

```
... FROM tab1, OUTER(tab2)
    WHERE tab1.col1 = tab2.col2
    AND tab2.colx > 10
```

3. Using subscript in outer conditions:

```
... FROM tab1, OUTER(tab2)
    WHERE tab1.col1[1,3] = tab2.col2[1,3]
```

**Notes:**

1. Table aliases are detected in OUTER expressions.

   OUTER example with table alias:

   ```
   ... OUTER(tab alias1) ...
   ```

2. In the outer join, `outertab.col` can be placed on both right or left sides of the equal sign:

   ```
   ... WHERE outertab.col1 = maintab.col2
   ```

3. Table names detection is not case-sensitive:

   ```
   SELECT ... FROM tab1, TAB2
   WHERE tab1.col1 = tab2.col2
   ```

4. Temporary tables are supported in OUTER specifications:

   ```
   CREATE TEMP TABLE tt1 ( ... )
   SELECT ... FROM tab1, OUTER(tt1) ...
   ```

**Related concepts**

*Outer joins* on page 643

Use standard ISO outer join syntax instead of the old IBM® Informix® `OUTER()` syntax.

**Transactions handling**

**Informix®**

With the Informix® native mode (non ANSI):

- Transactions blocks start with `BEGIN WORK` and terminate with `COMMIT WORK` or `ROLLBACK WORK`.
- Statements executed outside a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.

```
UPDATE tab1 SET ...   -- auto-committed
BEGIN WORK            -- start of TX block
UPDATE tab1 SET ...
UPDATE tab2 SET ...
...
COMMIT WORK           -- end of TX block
```

Informix® version 11.50 introduces savepoints:

```
SAVEPOINT name [UNIQUE]
ROLLBACK WORK1 TO SAVEPOINT _name_ 1
```
Oracle® MySQL and MariaDB

- Transactions are started with `START TRANSACTION`.
- Transactions are validated with `COMMIT [WORK]`.
- Transactions are canceled with `ROLLBACK [WORK]`.
- Savepoints can be placed with `SAVEPOINT name`.
- Transactions can be rolled back to a savepoint with `ROLLBACK [WORK] TO [SAVEPOINT] name`.
- Savepoints can be released with `RELEASE SAVEPOINT name`.
- Statements executed outside of a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.

Solution

Informix® transaction handling commands are automatically converted to MySQL instructions to start, validate or cancel transactions.

MySQL does not support transactions by default. You must set the server system parameter `table_type=InnoDB`.

Regarding the transaction control instructions, the BDL applications do not have to be modified in order to work with MySQL, as long as you have a transaction manager installed with MySQL.

If you want to use savepoints, do not use the `UNIQUE` keyword in the savepoint declaration, always specify the savepoint name in `ROLLBACK TO SAVEPOINT`, and do not drop savepoints with `RELEASE SAVEPOINT`.

Related concepts

Database transactions on page 600

Database transactions define a set of SQL instructions to be executed as a whole, or rolled back as a whole.

Temporary tables

Informix®

Informix® temporary tables are created with the `CREATE TEMP TABLE DDL` instruction or with `SELECT ... INTO TEMP` statement:

```
CREATE TEMP TABLE tt1 ( pkey INT, name VARCHAR(50) )
CREATE TEMP TABLE tt2 ( pkey INT, name VARCHAR(50) ) WITH NO LOG
SELECT * FROM tab1 WHERE pkey > 100 INTO TEMP tt2
```

Temporary tables are automatically dropped when the SQL session ends, but they can also be dropped with the `DROP TABLE` command. There is no name conflict when several users create temporary tables with the same name.

BDL REPORTs can create a temporary table when the rows are not sorted externally (by the source SQL statement).

Informix® allows you to create indexes on temporary tables. No name conflict occurs when several users create an index on a temporary table by using the same index identifier.

When creating temporary tables in Informix®, the `WITH NO LOG` clause can be used to avoid the overhead of recording DML operations in transaction logs.

Oracle® MySQL and MariaDB

MySQL support temporary tables with the following syntax:

```
CREATE TEMPORARY TABLE tablename ( coldef [, . . . ] )
```
and:

```
CREATE TEMPORARY TABLE tablename LIKE other-table
```

**Solution**

In BDL, Informix® temporary tables instructions are converted to generate native SQL Server temporary tables.

**Important:**

Simple Informix-style SQL statement creating temporary tables can be converted to a native SQL equivalent instruction. However, complex SQL statements such as `SELECT .. INTO TEMP` with subqueries may fail. In such cases, create a view from the complex query and then create the temp table from the view. Or, disable Informix emulation and use the native SQL syntax to create the temporary table (`EXECUTE IMMEDIATE "/* fglhint_no_ifxemul */ ..."`)  

With Informix SQL, if the source table has a column defined as `SERIAL` or `BIGSERIAL`, a `SELECT ... INTO TEMP` will produce a new temp table with an auto-incremented serial column. With the `SELECT ... INTO TEMP` emulation for non-Informix databases, not using the native sequence generators (such as `IDENTITY` columns in SQL Server), the resulting temporary table will get a simple `INTEGER` or `BIGINT` column, instead of an auto-incremented column. 

The general FGLPROFILE entry to control temporary table emulation is:

```
dbi.database.dsnname.ifxemul.temptables = 1 true 1 false 1
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

- Temporary tables on page 641
- Syntax for temporary table creation is not unique across all database engines.

**Substrings in SQL**

**Informix®**

Informix® SQL statements can use subscripts on columns defined with the character data type:

```
SELECT ... FROM tab1 WHERE col1[2,3] = 'RO'
SELECT ... FROM tab1 WHERE col1[10] = 'R' -- Same as col1[10,10]
UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...
SELECT ... FROM tab1 ORDER BY col1[1,3]
```

**Important:** With other database servers as Informix®, when the subscript notation is used to modify column values in UPDATE statement, or as ORDER BY element, you will get an SQL error:

```
UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...
SELECT ... FROM tab1 ORDER BY col1[1,3]
```

**Oracle® MySQL and MariaDB**

MySQL and MariaDB provide the function, to extract a substring from a string expression:

```
SELECT .... FROM tab1 WHERE SUBSTRING(col1,2,3) = 'RO'
SELECT SUBSTRING('Some text',6,3) ... -- Gives 'tex'
```

**Solution**

Replace all Informix® `col[x,y]` right-value expressions by `SUBSTRING(col,x,y-x+1)`. 
Rewrite UPDATE and ORDER BY clauses using \texttt{col[x,y]} expressions.

The translation of \texttt{col[x,y]} expressions can be controlled with the following FGLPROFILE entry:

\begin{verbatim}
dbi.database.dsnname.ifxemul.colsubs = \{ true \ false \}
\end{verbatim}

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

\textbf{Related concepts}

Substring expressions on page 644

Handle substrings expressions with different database engines.

\section*{The \texttt{LENGTH}() function}

\textbf{Informix}\textsuperscript{®}

Informix\textsuperscript{®} provides the \texttt{LENGTH()} function to count the number of bytes of a character string expression:

\begin{verbatim}
SELECT LENGTH("aaa"), LENGTH(col1) FROM table
\end{verbatim}

Informix\textsuperscript{®} \texttt{LENGTH()} does not count the trailing blanks for \texttt{CHAR} or \texttt{VARCHAR} expressions, while Oracle counts the trailing blanks.

Informix\textsuperscript{®} \texttt{LENGTH()} returns 0 when the given string is empty. That means, \texttt{LENGTH('')}=0.

\textbf{Oracle\textsuperscript{®} MySQL}

MySQL supports the \texttt{LENGTH()} function, which is similar to Informix\textsuperscript{®} \texttt{LENGTH()}.

The MySQL \texttt{LENGTH()} function ignores trailing blanks.

When passing \texttt{NULL} as parameter, the MySQL \texttt{LENGTH()} function returns \texttt{NULL}.

\section*{Solution}

The SQL \texttt{LENGTH()} function name can be used with MySQL.

\textbf{Related concepts}

The \texttt{LENGTH()} function in SQL on page 647

The semantics of the \texttt{LENGTH()} SQL function differs according to the database engine.

\section*{MATCHES and \texttt{LIKE}}

\textbf{Informix}\textsuperscript{®}

Informix\textsuperscript{®} supports MATCHES and LIKE operators in SQL statements.

MATCHES expects * and ? wild-card characters, while LIKE uses the % and _ wild-cards as equivalents.

\begin{verbatim}
( col MATCHES 'Smi*' AND col NOT MATCHES 'R?x' )
( col LIKE 'Smi%' AND col NOT LIKE 'R_x' )
\end{verbatim}

MATCHES accepts also brackets notation, to specify a set of matching characters at a given position:

\begin{verbatim}
( col MATCHES '[Pp]aris' )
( col MATCHES '[0-9][a-z]*' )
\end{verbatim}

\textbf{Oracle\textsuperscript{®} MySQL and MariaDB}

MySQL and MariaDB do not provide an equivalent of the Informix\textsuperscript{®} MATCHES operator.

The LIKE operator is supported.
Solution

The database driver is able to translate Informix® MATCHES expressions to LIKE expressions, when no [ ] bracket character ranges are used in the MATCHES operand.

The MATCHES to LIKE expression translation is controlled by the following FGLPROFILE entry:

```
  dbi.database.dbname.ifxemul.matches = [ true ] false
```

**Important:** Only [NOT] MATCHES followed by a search pattern provided as a string literal can be converted by ODI drivers. A [NOT] MATCHES followed by a ? question mark parameter place holder is not translated!

For maximum portability, consider replacing the MATCHES expressions with LIKE expressions in all SQL statements.

Avoid using CHAR(N) types for variable length character data (such as name, address).

Related concepts

MATCHES and LIKE operators on page 645
Use the standard LIKE operator instead of the MATCHES operator.

Row limiting clause

Informix®

Informix® SQL supports the SKIP and FIRST/LIMIT keywords to limit the number of rows of a result set:

```
SELECT SKIP 10 FIRST 20 customer.* FROM customer ... ORDER BY cust_name
```

This Informix SQL syntax is not portable.

Recent database engines support the row limiting clause syntax defined by the SQL standard:

```
SELECT ... OFFSET n ROWS FETCH FIRST m ROWS ONLY
```

This should be the preferred syntax to be used, if all target database types support this SELECT clause.

The ODI database drivers can convert the Informix SQL SKIP/FIRST row limiting clause to a native SQL equivalent, if the row limiting clause parameters are simple integer literals (the clause is not translated when using SQL parameters / program variables).

**Important:** In addition to the SKIP/FIRST clause of the projection clause, Informix SQL supports also a LIMIT clause after the ORDER BY clause:

```
SELECT customer.* FROM customer ... ORDER BY cust_name LIMIT 10
```

This Informix SQL syntax construction is not converted by the ODI drivers. To benefit from the conversion, review the code to use the Informix SQL SKIP/FIRST clause instead.

Oracle® MySQL and MariaDB

Oracle® MySQL and MariaDB support the following row limiting clause:

```
SELECT ... ORDER BY ... LIMIT m OFFSET n
```

Solution

The Informix SQL row limiting clause can be converted by the Oracle® MySQL and MariaDB drivers to the native SQL equivalent clause, when the parameters are simple integer literals.
Note: The row limiting clause must not use SQL parameters. Only row limiting clauses using integer constants will be converted.

The translation of the Informix SQL row limiting clause can be controlled with the following FGLPROFILE entry:

```
dbi.database.dsname.ifxemul.rowlimiting = {true | false}
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

Related concepts
Row limiting clause (SELECT) on page 650
How to use the right clause to limit the number of rows produced by a SELECT statement?

BDL programming
Oracle® MySQL related programming topics.

INSERT cursors

Informix®
Informix® provides insert cursors to optimize row creation in a database. An insert cursor is declared as a cursor, and rows as added with the PUT instruction. The rows are buffered and sent to the database server when executing a FLUSH instruction, or when the cursor is closed with CLOSE. When using transactions in Informix®, the OPEN, PUT and FLUSH instructions must be executed within a transaction block.

```
DECLARE c1 CURSOR FOR INSERT INTO tab1 ... 
BEGIN WORK
OPEN c1
WHILE ...
  PUT c1 USING var-list
END WHILE
CLOSE c1
COMMIT WORK
```

Oracle® MySQL and MariaDB
MySQL and MariaDB do not support insert cursors.

Solution
Insert cursors are emulated by the database interface, using basic INSERT SQL instructions. The performances might be not as good as with Informix®, but the feature is fully supported.

Related concepts
Insert cursors on page 633
Using insert cursors with non-Informix databases.

Cursors WITH HOLD

Informix®
Informix® closes opened cursors automatically when a transaction ends, unless the WITH HOLD option is used in the DECLARE instruction:

```
DECLARE c1 CURSOR WITH HOLD FOR SELECT ...
OPEN c1
BEGIN WORK
FETCH c1 ...
COMMIT WORK
FETCH c1 ...
```
Oracle® MySQL and MariaDB

With MySQL and MariaDB, opened cursors using SELECT statements without a FOR UPDATE clause are not closed when a transaction ends. All database cursors are WITH HOLD cursors, unless the FOR UPDATE clause is used in the SELECT statement.

Cursors declared FOR UPDATE and using the WITH HOLD option cannot be supported with MySQL because FOR UPDATE cursors are automatically closed by the database server when the transaction ends.

Solution

BDL cursors declared WITH HOLD remain open even after terminating a transaction with a COMMIT WORK or ROLLBACK WORK.

For consistency with other database brands, database cursors that are not declared WITH HOLD are automatically closed, when a COMMIT WORK or ROLLBACK WORK is performed.

Important: Opening a WITH HOLD cursor declared with a SELECT FOR UPDATE results in an SQL error; in the same conditions, this does not normally appear with Informix®. Review the program logic in order to find another way to set locks.

Related concepts

Cursors WITH HOLD on page 631
Programming WITH HOLD cursors using SELECT with and without FOR UPDATE clause.

SELECT ... FOR UPDATE

Informix®

Legacy BDL programs typically use a cursor with SELECT FOR UPDATE to implement pessimistic locking and avoid several users editing the same rows:

```sql
DECLARE cc CURSOR FOR SELECT ... FROM tab WHERE ... FOR UPDATE
OPEN cc
FETCH cc <-- lock is acquired
...
CLOSE cc <-- lock is released
```

The row must be fetched in order to set the lock.

If the cursor is local to a transaction, the lock is released when the transaction ends. If the cursor is declared WITH HOLD, the lock is released when the cursor is closed.

Informix® provides the SET LOCK MODE instruction to define the lock wait timeout:

```sql
SET LOCK MODE TO { WAIT | NOT WAIT | WAIT seconds }
```

The default mode is NOT WAIT.

Oracle® MySQL and MariaDB

MySQL and MariaDB support the FOR UPDATE clause in SELECT.

MySQL and MariaDB locking mechanism depends upon the transaction manager.

The default locking granularity is per table when you use the default non-transactional configuration.

Use the InnoDB Storage Engine to get transactions and locking mechanisms.
Locks are released at the end of the transaction.

Solution
Check if the MySQL storage engine supports `SELECT FOR UPDATE`, otherwise review the program logic.

Related concepts
Cursors WITH HOLD on page 631
Programming WITH HOLD cursors using `SELECT` with and without `FOR UPDATE` clause.

**UPDATE/DELETE ... WHERE CURRENT OF**

**Informix**
Informix allows positioned UPDATEs and DELETEs with the "WHERE CURRENT OF cursor" clause, if the cursor has been DECLARED with a SELECT ... FOR UPDATE statement.

**Oracle®, MySQL and MariaDB**
MySQL and MariaDB do not support UPDATE/DELETE with the WHERE CURRENT OF cursor clause.

Solution
UPDATE/DELETE ... WHERE CURRENT OF is not supported with MySQL/MariaDB.
As a replacement of WHERE CURRENT OF, if the database table is defined with a primary key column, use the value fetched from the SELECT [FOR UPDATE] cursor in the WHERE clause of the UPDATE/DELETE statement.

Related concepts
Positioned UPDATE/DELETE on page 631
Using positioned updates/deletes with named database cursors.

**LOAD and UNLOAD**

**Informix**
Informix provides two SQL instructions to export / import data from / into a database table:
The UNLOAD instruction copies rows from a database table into a text file:

```
UNLOAD TO "filename.unl" SELECT * FROM tab1 WHERE ..
```

The LOAD instructions insert rows from a text file into a database table:

```
LOAD FROM "filename.unl" INSERT INTO tab1
```

**Oracle®, MySQL and MariaDB**
MySQL does not provide LOAD and UNLOAD instructions.

Solution
LOAD and UNLOAD instruction are implemented in the Genero BDL runtime system with basic INSERT (for LOAD) or SELECT (for UNLOAD) SQL commands. The LOAD and UNLOAD instruction can be supported with various database servers.

However, LOAD and UNLOAD require the description of the column types in order to work, that can lead to some differences in the data formatting.

Note: If no transaction is started, the LOAD instruction will automatically execute a BEGIN WORK and COMMIT WORK when finished, or ROLLBACK WORK if a row insertion failed while loading. Terminating a transaction will
automatically close cursors not defined with `HOLD` option. To workaround this situation, see more details in the `LOAD` on page 736 reference topic.

The `LOAD` and `UNLOAD` BDL instructions are supported with MySQL and MariaDB.

**Related concepts**

*LOAD and UNLOAD instructions* on page 639

The `LOAD` and `UNLOAD` instructions can produce different data formats depending on the database server type.

**SQL Interruption**

**Informix®**

With Informix®, it is possible to interrupt a long running query if the `SQL INTERRUPT ON` option.

**Oracle® MySQL and MariaDB**

MySQL and MariaDB provides the `KILL QUERY` command to interrupt a running query on the server.

**Important:** The database client program must open a second connection to execute the `KILL QUERY` statement.

**Solution**

SQL interruption is supported with MySQL.

The database driver opens a second connection to the server and sends a `KILL QUERY` command, with the MySQL process id of the current connection.

**Important:** Opening a second connection does not work when using Unix sockets, connect to MySQL with a host name and TCP port.

**Related concepts**

*Using SQL interruption* on page 601

Interrupt long running SQL queries, or interrupt queries waiting for locked data.

**Scrollable cursors**

**Informix®**

Informix® SQL and Genero BDL support scrollable cursors when you specify the `SCROLL` clause in the `DECLARE` cursor instruction:

```
DECLARE c1 SCROLL CURSOR FOR SELECT ...
```

**Important:** Informix does not allow to fetch TEXT/BYTE columns with scrollable cursors. If you declare a scroll cursor with a `SELECT` containing TEXT/BYTE columns, Informix will produce the SQL error -611 when executing the `OPEN` instruction.

**Oracle® MySQL and MariaDB**

MySQL and MariaDB do not support scrollable cursors.

**Solution**

The MySQL and MariaDB database drivers emulate scrollable cursors by fetching rows in a temporary file.

**Important:** With MySQL and MariaDB, it is NOT possible to use LOB columns in a scrollable cursor. If TEXT/ BYTE columns are used with a scrollable cursor, the `OPEN` instruction will produce the SQL error -611 (as with Informix).

**Related concepts**

*Scrollable cursors* on page 623
How scrollable cursors can be supported on different databases.

**Stored procedure calls**

Oracle® MySQL and MariaDB implement stored procedures and stored functions as a collection of SQL statements that can take and return user-supplied parameters. Functions are very similar to procedures, except that they return a scalar value and can be used in SQL expressions.

To create a stored procedure in an Oracle® MySQL or MariaDB database, use the `CREATE PROCEDURE` statement. See reference documentation for more details.

**Stored procedures with output parameters**

Since Oracle® MySQL C API (version 5.0) does not support an output parameter specification, the `IN/OUT/INOUT` technique cannot be used.

In order to return values from a MySQL stored procedure or stored function, you must use SQL variables. There are three steps to execute the procedure or function:

1. With the `SET` SQL statement, create and assign an SQL variables for each parameter.
2. CALL the stored procedure or stored function with the created SQL variables.
3. Perform a `SELECT` statement to return the SQL variables to the application.

In order to retrieve returning values into program variables, you must use an `INTO` clause in the `EXECUTE` instruction.

The following example shows how to call a stored procedure with output parameters:

MySQL version 5.0 does not allow you to prepare the `CREATE PROCEDURE` statement; you may need to execute this statement from the mysql command line tool.

MySQL version 5.0 cannot execute "SELECT @variable" with server-side cursors. Since the MySQL driver uses server-side cursors to support multiple active result sets, it is not possible to execute the `SELECT` statement to return output parameter values.

MySQL version >=5.0 evaluates "@variable" user variables assigned with a string as large text (CLOB) expressions. That type of values must normally be fetched into `TEXT` variable. To workaround this behavior, you can use the `substring(@var,1,255)` function to return a `VARCHAR()` expression from MySQL and fetch into a `VARCHAR()` variable.

```sql
MAIN
DEFINE n INTEGER
DEFINE d DECIMAL(6,2)
DEFINE c VARCHAR(200)
DATABASE test1
EXECUTE IMMEDIATE "create procedure proc1(
    p1 integer,
    out p2 numeric(6,2),
    out p3 varchar(200)
)"
    no sql begin
    set p2 = p1 + 0.23;
    set p3 = concat( 'Value = ', p1 );
  end;
LET n = 111
EXECUTE IMMEDIATE "set @p1 = ", n
EXECUTE IMMEDIATE "set @p2 = NULL"
EXECUTE IMMEDIATE "set @p3 = NULL"
EXECUTE IMMEDIATE "call proc1(@p1, @p2, @p3)"
PREPARE stmt FROM "select @p2, substring(@p3,1,200)"
EXECUTE stmt INTO d, c
DISPLAY d
DISPLAY c
```
Stored functions returning values

The following example shows how to retrieve the return value of a stored function with Oracle® MySQL:

MySQL version 5.0 does not allow you to prepare the `CREATE FUNCTION` statement; you may need to execute this statement from the `mysql` command line tool.

```sql
DEFINE n INTEGER
DEFINE c VARCHAR(200)
DATABASE test1
EXECUTE IMMEDIATE "create function func1(p1 integer)"
|| " no sql begin"
|| " return concat( 'Value = ', p1 );"
|| " end;"
PREPARE stmt FROM "select func1(?)"
LET n = 111
EXECUTE stmt USING n INTO c
DISPLAY c
END MAIN
```

Stored procedures with result set

Note that Oracle® MySQL version 5.0 stored procedures and stored functions cannot return a result set.

Oracle® Database

Supported versions

Genero BDL supports the following Oracle Database versions:

- Oracle Database 11.x
- Oracle Database 12.x
- Oracle Database 18.x
- Oracle Database 19.x

Installation (Runtime Configuration)

Oracle Database related installation topics.

Install Oracle and create a database - database configuration/design tasks

If you are tasked with installing and configuring the database, here is a list of steps to be taken:

1. Install the ORACLE database software on your computer.
2. Create and setup the Oracle instance and database. Consider creating a multitenant database when using Oracle 12c and higher, to create several pluggable databases (PDB) in the same Oracle instance.
   Specify the database character set when creating the database instance. If you plan to use UTF-8, use also character length semantics (See NLS_LENGTH_SEMANTICS session parameter).
3. Create a database context dedicated to your application.
   According the Oracle version, define a db user / schema to hold application tables, or create a pluggable database (starting with Oracle 12c).
   a) With Oracle version 11g and lower, group application tables in a schema by creating a dedicated database user.
      Connect as system user with:

      ```
      $ sqlplus / AS SYSDBA
      ```
and execute the following SQL command to create the db user:

```
CREATE USER appadmin IDENTIFIED BY password;
```

Grant privileges to the application administrator user:

```
GRANT DBA, UNLIMITED TABLESPACE TO appadmin;
```

b) With Oracle version 12c and higher, group application tables in a pluggable database (PDB).

Connect as system user with:

```
$ sqlplus / AS SYSDBA
```

and create a pluggable database and its PDB administrator user. This is a basic PDB creation example using Oracle Managed Files, consider planning the PDB creation with the person in charge of Oracle database administration:

```
CREATE PLUGGABLE DATABASE mypdb
    ADMIN USER pdbadmin IDENTIFIED BY password
    ROLES = (DBA)
    DEFAULT TABLESPACE mypdb_01
    DATAFILE 'path_01' SIZE 250M AUTOEXTEND ON;
```

For now the PDB is only mounted, it must be opened for regular usage:

```
ALTER PLUGGABLE DATABASE mypdb OPEN;
```

PDBs must be identified as separate database services (i.e. different from the CDB service). By default Oracle creates a database service with the same name as the PDB. To access the PDB through TNS, create the mypdb record in TNSNAMES.ORA file in addition to the default database service (ORC*):

```
tnsname =
    (DESCRIPTION =
        (ADDRESS = (PROTOCOL = TCP)(HOST = localhost)(PORT = 1521))
        (CONNECT_DATA =
            (SERVER = DEDICATED)
            (SERVICE_NAME = mypdb)
        )
    )
```

By default when Oracle starts, the PDBs are mounted but are not open for regular usage. With Oracle 12c, you can create a database trigger to open all PDBs automatically:

```
CREATE OR REPLACE TRIGGER open_pdbs
    AFTER STARTUP ON DATABASE
BEGIN
    EXECUTE IMMEDIATE 'ALTER PLUGGABLE DATABASE ALL OPEN';
END open_pdbs;
/
```

Or, save the state of a specific PDB when it is open:

```
ALTER PLUGGABLE DATABASE test1 OPEN;
ALTER PLUGGABLE DATABASE test1 STAND_STATE;
```

Re-connect as PDB administrator and create a user dedicated to application tables administration:

```
CONNECT pdbadmin/password@mypdb
CREATE USER appadmin IDENTIFIED BY password;
```
Grant privileges to the application administrator user:

```
GRANT DBA, UNLIMITED TABLESPACE TO appadmin;
```

4. If programs create temporary tables, you must create a dedicated tablespace and schema depending on the type of temporary table emulation used.

For more details about temporary table emulations, see Temporary tables on page 943.

5. Create the application tables by connecting to the database context as the application administrator:

```
$ sqlplus appadmin/password@tnsname
```

Convert Informix® data types to Oracle data types. See issue data type Conversion Tables for more details.

6. If you plan to use SERIAL emulation, you must choose a serial emulation method.

Select the best emulation technique that matches your needs. You need to prepare the database depending on the emulation type. For more details, see SERIAL and BIGSERIAL data types on page 931.

Prepare the runtime environment - connecting to the database

1. In order to connect to Oracle®, you must have a database driver "dbmora" in $FGLDIR/dbdrivers.

2. If you want to connect to a remote Oracle® server from an application server, you must install the Oracle® Client Software on your application server and configure this.

3. Make sure that the ORACLE client environment variables are properly set.

Check variables such as ORACLE_HOME (the path to the installation directory), ORACLE_SID (the server identifier when connecting locally), etc. See the Oracle® documentation for more details.

4. Verify the environment variable defining the search path for database client shared libraries (libclntsh.so on UNIX™, OCI.DLL on Windows®)

<table>
<thead>
<tr>
<th>Oracle® version</th>
<th>Shared library environment setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle® 11g and higher</td>
<td><em>UNIX™</em>: Add $ORACLE_HOME/lib to LD_LIBRARY_PATH (or its equivalent).</td>
</tr>
<tr>
<td></td>
<td><em>Windows®</em>: Add %ORACLE_HOME%\bin to PATH.</td>
</tr>
</tbody>
</table>

5. Check the database client locale settings (NLS_LANG, NLS_DATE_FORMAT, etc).

The database client locale must match the locale used by the runtime system (LC_ALL, LANG).

6. If you are using the TNS protocol, verify if the Oracle® listener is started on the server.

7. To test the client environment settings, you can try to connect to the Oracle® server with the SQL*Plus tool:

```
$ sqlplus username/password@service
```

8. Disable installation of signal handlers for the OCI diagnostic framework.

By default, the OCI client library installs signal handlers for the OCI diagnostic framework. These signal handlers are registered when connecting to the database, and can conflict with the signal handlers installed by the fglrun runtime system.

Unless you are explicitly asked to use OCI diagnostic framework, there is no need to let OCI use these signals.

To disable OCI client library signal handlers, define the following parameters in the sqlnet.ora file:

```
DIAG_ADR_ENABLED=FALSE
DIAG_DDE_ENABLED=FALSE
DIAG_SIGHANDLER_ENABLED=FALSE
```

For more details, search Oracle® OCI documentation about "Fault Diagnosability in OCI".

9. Set up the FGLPROFILE entries for database connections.
a) Set up FGLPROFILE for the SERIAL emulation method.
The following entry defines the SERIAL emulation method. You can use the SEQUENCE based trigger or the SERIALREG based trigger method:

```
dbi.database.dbname.ifxemul.datatype.serial.emulation = "(native|regtable)"
```

The value 'native' selects the SEQUENCE based method, and the value 'regtable' selects the SERIALREG based method. This entry has no effect if `dbi.database.dbname.ifxemul.datatype.serial` is set to 'false'.

The default is SERIAL emulation enabled with native method (SEQUENCE-based). See issue SERIAL and BIGSERIAL data types on page 931 for more details.

b) The "source" parameter defines the TNS name of the Oracle database.

```
dbi.database.dbname.source = "stock"
```

c) Define the database schema selection if needed.

The following entry defines the database schema to be used by the application. The database interface automatically executes an "ALTER SESSION SET CURRENT_SCHEMA owner" instruction to switch to a specific schema:

```
dbi.database.dbname.ora.schema = "name"
```

Here `dbname` identifies the database name used in the BDL program (DATABASE `dbname`) and `name` is the schema name to be used in the ALTER SESSION instruction. If this entry is not defined, no ALTER SESSION instruction is executed and the current schema defaults to the user's name.

d) Define pre-fetch parameters.

Oracle® offers high performance by pre-fetching rows in memory. The pre-fetching parameters can be tuned with the following entries:

```
dbi.database.dbname.ora.prefetch.rows = integer
dbi.database.dbname.ora.prefetch.memory = integer # in bytes
```

These values will be applied to all application cursors.

The interface pre-fetches rows up to the `prefetch.rows` limit unless the `prefetch.memory` limit is reached, in which case the interface returns as many rows as will fit in a buffer of size `prefetch.memory`. By default, pre-fetching is on and defaults to 10 rows; the memory parameter is set to zero, so the memory size is not included in computing the number of rows to pre-fetch.

e) If needed, define a specific command to generate session identifiers with this FGLPROFILE setting:

```
dbi.database.dbname.ora.sid.command = "SELECT ...
```

This unique session identifier will be used to create table names for temporary table emulation. By default, the database driver will use "SELECT USERENV('SESSIONID') FROM DUAL".

f) If needed, define a specific command to generate session identifiers with this FGLPROFILE setting:

```
dbi.database.dbname.ora.sid.command = "SELECT ...
```

This unique session identifier will be used to create table names for temporary table emulation. By default, the database driver will use "SELECT USERENV('SESSIONID') FROM DUAL".

The default temporary table emulation uses regular permanent tables.
If this does not fit your needs, you can use GLOBAL TEMPORARY TABLES with this FGLPROFILE setting:

```
dbi.database.dbname.ifxemul.temptables.emulation = "global"
```

h) By default, the tablespace for the default temporary table emulation is TEMPTABS. For global temporary table emulation, there is no tablespace used by default.

If required, define the tablespace to be used for temporary table emulations (note that this parameter applies to all temporary table emulation methods):

```
dbi.database.dbname.ora.temptables.tablespace = "mytemptabs"
```

i) By default, no schema is used for the default temporary table emulation. For global temporary table emulation, the schema TEMPTABS is used by default.

If required, define the schema to be used for temporary table emulations (note that these parameters applies to all temporary table emulation methods). Define the "source" parameter to use the current user login, or a SELECT command to produce the schema name:

```
# Get the schema from the current user name specified in connection command:
dbi.database.dbname.ora.temptables.schema.source = "login"
# or, get the schema from a SELECT statement:
dbi.database.dbname.ora.temptables.schema.source = "command"
dbi.database.dbname.ora.temptables.schema.command = "SELECT SYS_CONTEXT('USERENV','SESSION_USER') FROM DUAL"
```

### Database concepts

**Oracle Database related database concepts topics.**

**Database concepts**

Informix® servers can handle multiple database entities. By default an ORACLE instance can only handle one database entity. Starting with Oracle 12c, you can use a multi-tenant database to define several pluggable databases.

**Tip:** If you have several Informix database entities, migrating from the Informix database to another database it is a good opportunity to centralize all tables in a single database. To avoid conflicts with table names, use a prefix when needed.

ORACLE can manage multiple schemas, but by default other users must give the owner name as prefix to the table name:

```
SELECT * FROM stores.customer
```

**Solution 1: With Oracle 12c and higher**

Oracle 12c introduced the multi-tenant database concept, where you can create several pluggable databases in a root container. Consider using this feature, if you need to create several copies of the same database entity, that can be accessed/seen as individual data sources.

**Solution 2: With Oracle 10g and 11g**

In an Oracle database, each user can manage his own database schema. You can dedicate a database user to administer each occurrence of the application database.

Any user can select the current database schema with the following SQL command:

```
ALTER SESSION SET CURRENT_SCHEMA = "schema"
```

Using this instruction, any user can access the tables without giving the owner prefix as long as the table owner has granted the privileges to access the tables.
You can make the database interface select the current schema automatically with the following fgplprofile entry:

dbi.database.dbname.schema = "schema"

When using multiple database schemas, it is recommended that you create them in separated tablespaces to enable independent backups and keep logical sets of tables together. The simplest way is to define a default tablespace when creating the schema owner:

```
CREATE USER user IDENTIFIED BY password
DEFAULT TABLESPACE deftablespace
TEMPORARY TABLESPACE tmptablespace
```

**Data storage concepts**

When converting from Informix® to Oracle® the aim is to try to preserve as much of the data storage information as possible in the process. The most important storage decisions made for Informix® database objects (like initial sizes and physical placement) can be applied to the Oracle® database.

Storage concepts are quite similar in Informix® and in Oracle®, but the names are different.

This table compares Informix® storage concepts to Oracle® storage concepts:

**Table 249: Physical units of storage**

<table>
<thead>
<tr>
<th><strong>Informix®</strong></th>
<th><strong>ORACLE</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>The largest unit of physical disk space is a &quot;chunk&quot;, which can be allocated either as a cooked file (I/O is controlled by the OS) or as raw device (=UNIX® partition, I/O is controlled by the database engine). A &quot;dbspace&quot; uses at least one &quot;chunk&quot; for storage. You must add &quot;chunks&quot; to &quot;dbspaces&quot; in order to increase the size of the logical unit of storage. A &quot;page&quot; is the smallest physical unit of disk storage that the engine uses to read from and write to databases. A &quot;chunk&quot; contains a certain number of &quot;pages&quot;. The size of a &quot;page&quot; must be equal to the operating system's block size. An &quot;extent&quot; consists of a collection of contiguous &quot;pages&quot; that the engine uses to allocate both initial and subsequent storage space for database tables. When creating a table, you can specify the first extent size and the size of future extents with the EXTENT SIZE and NEXT EXTENT options. For a single table, &quot;extents&quot; can be located in different &quot;chunks&quot; of the same &quot;dbspace&quot;.</td>
<td>One or more &quot;data files&quot; are created for each &quot;tablespace&quot; to physically store the data of all logical structures. Like Informix® &quot;chunks&quot;, a &quot;data file&quot; can be an OS file or a raw device. You can add &quot;data files&quot; to a &quot;tablespace&quot; in order to increase the size of the logical unit of storage or you can use the AUTOEXTEND option when using OS files. At the finest level of granularity, Oracle® stores data in &quot;data blocks&quot; which size corresponds to a multiple of the operating system's block size. You set the &quot;data block&quot; size when creating the database. An &quot;extent&quot; is a specific number of contiguous &quot;data blocks&quot;, obtained in a single allocation. When creating a table, you can specify the first extent size and the size of future extents with the STORAGE() option. For a single table, &quot;extents&quot; can be located in different &quot;data files&quot; of the same &quot;tablespace&quot;.</td>
</tr>
</tbody>
</table>

**Table 250: Logical units of storage**

<table>
<thead>
<tr>
<th><strong>Informix®</strong></th>
<th><strong>ORACLE</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>A &quot;table&quot; is a logical unit of storage that contains rows of data values.</td>
<td>Same concept as Informix®.</td>
</tr>
</tbody>
</table>
Informix® ORACLE

A "database" is a logical unit of storage that contains table and index data. Each database also contains a system catalog that tracks information about database elements like tables, indexes, stored procedures, integrity constraints and user privileges.

Same concept as Informix®, but one Oracle® instance can manage only one database, in the same way that Informix® does.

Database tables are created in a specific "dbspace", which defines a logical place to store data.

Database tables are created in a specific "tablespace", which defines a logical place to store data.

If no dbspace is given when creating the table, Informix® defaults to the current database dbspace.

If no tablespace is given when creating the table, Oracle® defaults to the user's default tablespace.

The total disk space allocated for a table is the "tblspace", which includes "pages" allocated for data, indexes, blobs, tracking page usage within table extents.

A "segment" is a set of "extents" allocated for a certain logical structure. There are four different types of segments, including data segments, index segments, rollback segments and temporary segments.

Do not confuse the Informix® "tblspace" concept and Oracle® "tablespaces".

Informix® ORACLE

Table 251: Other concepts related to storage

Informix® ORACLE

When initializing an Informix® engine, a "root dbspace" is created to store information about all databases, including storages information (chunks used, other dbspaces, etc.)

Each Oracle® database has a "control file" that records the physical structure of the database, like the database name, location and names of "data files" and "redo log" files, and time stamp of database creation.

The "physical log" is a set of continuous disk pages where the engine stores "before-images" of data that has been modified during processing.

A "rollback segment" records the actions of SQL transactions that can be rolled back, and it records the data as it existed before an operation in a transaction.

The "logical log" is a set of "logical-log files" used to record logical operations during online processing. All transaction information is stored in the logical log files if a database has been created with transaction log.

The "redo log files" hold all changes made to the database, in case the database experiences an instance failure.

Informix® combines "physical log" and "logical log" information when doing fast recovery. Saved "logical logs" can be used to restore a database from tape.

Redo entries record data that can be used to reconstruct all changes made to the database, including the rollback segments stored in the database buffers of the SGA. Therefore, the online redo log also protects rollback data.

Concurrency management

Data consistency and concurrency concepts

- Data Consistency applies to situations when readers want to access data currently being modified by writers.
- Concurrent Data Access applies to situations when several writers are accessing the same data for modification.
- Locking Granularity defines the amount of data concerned when a lock is set (for example, row, page, table).

Informix® uses a locking mechanism to handle data consistency and concurrency. When a process changes database information with UPDATE, INSERT or DELETE, an exclusive lock is set on the touched rows. The lock remains active until the end of the transaction. Statements performed outside a transaction are treated as a transaction containing a single operation and therefore release the locks immediately after execution. SELECT statements can
set shared locks, depending on isolation level. In case of locking conflicts (for example, when two processes want to acquire an exclusive lock on the same row for modification, or when a writer is trying to modify data protected by a shared lock), the behavior of a process can be changed by setting the lock wait mode.

Control:

- Lock wait mode: SET LOCK MODE TO ...
- Isolation level: SET ISOLATION TO ...
- Locking granularity: CREATE TABLE ... LOCK MODE {PAGE|ROW}
- Explicit exclusive lock: SELECT ... FOR UPDATE

Defaults:

- The default isolation level is READ COMMITTED.
- The default lock wait mode is NOT WAIT.
- The default locking granularity is PAGE.

**ORACLE**

When data is modified, exclusive locks are set and held until the end of the transaction. For data consistency, ORACLE uses a multi-version consistency model: a copy of the original row is kept for readers before performing writer modifications. Readers do not have to wait for writers as in Informix®. The simplest way to think of Oracle's implementation of read consistency is to imagine each user accessing a private copy of the database, hence the multi-version consistency model. The lock wait mode cannot be changed session wide as in Informix®, the waiting behavior can be controlled with a SELECT FOR UPDATE NOWAIT only. Locks are set at the row level in ORACLE, and this cannot be changed.

Control:

- Lock wait mode (on SELECT only): SELECT ... FOR UPDATE NOWAIT
- Isolation level: SET TRANSACTION ISOLATION LEVEL TO ...
- Explicit exclusive lock: SELECT ... FOR UPDATE [NOWAIT]

Defaults:

- The default isolation level is Read Committed (readers cannot see uncommitted data, no shared lock is set when reading data).

The main difference between Informix® and ORACLE is that readers do not have to wait for writers in ORACLE.

**Solution**

The SET ISOLATION TO ... Informix® syntax is replaced by ALTER SESSION SET ISOLATION_LEVEL ... in Oracle. The next table shows the isolation level mappings done by the database driver:

<table>
<thead>
<tr>
<th>SET ISOLATION instruction in program</th>
<th>Native SQL command</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET ISOLATION TO DIRTY READ</td>
<td>ALTER SESSION SET ISOLATION_LEVEL = READ COMMITTED</td>
</tr>
<tr>
<td>SET ISOLATION TO COMMITTED READ [READ COMMITTED] [RETAIN UPDATE LOCKS]</td>
<td>ALTER SESSION SET ISOLATION_LEVEL = READ COMMITTED</td>
</tr>
<tr>
<td>SET ISOLATION TO CURSOR STABILITY</td>
<td>ALTER SESSION SET ISOLATION_LEVEL = READ COMMITTED</td>
</tr>
<tr>
<td>SET ISOLATION TO REPEATABLE READ</td>
<td>ALTER SESSION SET ISOLATION_LEVEL = SERIALIZABLE</td>
</tr>
</tbody>
</table>
ORACLE does not provide a dirty read mode, the (session wide) lock wait mode cannot be changed and the locking precision is always at the row level. Based on this, it is recommended that you work with Informix® in the read committed isolation level (default), make processes wait for each other (lock mode wait), and use the default page-level locking granularity.

See the Informix® and ORACLE documentation for more details about data consistency, concurrency and locking mechanisms.

Related concepts

- **Concurrent data access** on page 599
  Understanding concurrent data access and data consistency.

- **Optimistic locking** on page 624
  Implementing optimistic locking to handle access concurrently to the same database records.

- **Cursors WITH HOLD** on page 631
  Programming WITH HOLD cursors using SELECT with and without FOR UPDATE clause.

Transactions handling

**Informix®**

With the Informix® native mode (non ANSI):

- Transactions blocks start with BEGIN WORK and terminate with COMMIT WORK or ROLLBACK WORK.
- Statements executed outside a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.

```plaintext
UPDATE tab1 SET ...   -- auto-committed
BEGIN WORK            -- start of TX block
UPDATE tab1 SET ...
UPDATE tab2 SET ...
...
COMMIT WORK           -- end of TX block
```

Informix® version 11.50 introduces savepoints:

```plaintext
SAVEPOINT name [UNIQUE]
ROLLBACK [WORK] TO SAVEPOINT [name] 
RELEASE SAVEPOINT name
```

**ORACLE**

With ORACLE transactions:

- Beginnings of transactions are implicit.
- A transaction ends with a COMMIT or ROLLBACK statement.
- The current transaction is automatically committed when a DDL statement is executed.

```plaintext
UPDATE tab1 SET ...   -- start of TX block
UPDATE tab2 SET ...
...
COMMIT                -- end of TX block
```

ORACLE supports savepoints too. However, there are differences:

- Savepoints cannot be declared as UNIQUE
- Rollback must always specify the savepoint name
- You cannot release savepoints (RELEASE SAVEPOINT)
Solution

Regarding transaction control instructions, BDL applications do not have to be modified in order to work with ORACLE. The Informix® behavior is simulated with an autocommit mode in the ORACLE interface. A switch to the explicit commit mode is done when a BEGIN WORK is performed by the BDL program.

When executing a DDL statement inside a transaction, ORACLE automatically commits the transaction. Therefore, you must extract the DDL statements from transaction blocks.

If you want to use savepoints, do not use the UNIQUE keyword in the savepoint declaration, always specify the savepoint name in ROLLBACK TO SAVEPOINT, and do not drop savepoints with RELEASE SAVEPOINT.

See also SELECT FOR UPDATE

Related concepts

Database transactions on page 600

Database transactions define a set of SQL instructions to be executed as a whole, or rolled back as a whole.

Database users

Informix®

Until version 11.70.xC2, Informix® database users must be created at the operating system level and must be members of the ‘informix’ group.

Starting with 11.70.xC2, Informix® supports database-only users with the CREATE USER instruction, as in most other db servers.

Any database user must have sufficient privileges to connect and use resources of the database; user rights are defined with the GRANT command.

ORACLE

Oracle® users can be authenticated in different ways: as database users, as operating system users or by delegating authentication to another service, like Kerberos or LDAP.

Users must be created in the database with a CREATE USER command, to create a user authenticated by the database server:

```
CREATE USER username IDENTIFIED BY password
```

Oracle® users can also be created with the IDENTIFIED EXTERNALLY clause:

```
CREATE USER username IDENTIFIED EXTERNALLY
```

In this case, Oracle® trusts the operating system to authenticate the user. See the Oracle® documentation for OS user authentication configuration, especially the OS_AUTHENT_PREFIX (empty string) and REMOTE_OS_AUTHENT (true) server parameters. Note also that the Oracle® user name needs to be specified in uppercase in the CREATE USER instruction, and gets an additional prefix, depending on the operating system (domain name on Windows® platforms)

In Oracle®, users can also be defined in a central LDAP directory, with the IDENTIFIED GLOBALLY clause:

```
CREATE USER username IDENTIFIED GLOBALLY AS 'distinguished_name'
```

Global users are registered and managed by an external LDAP service, and are identified by the distinguished name (DN).

Oracle® supports also proxy authentication, by granting connection privileges through another "proxy user":

```
ALTER USER username GRANT CONNECT THROUGH proxy_user
```
Solution

Based on the application logic, you must create one or several Oracle® users. Use database or external authentication.

Connecting as an external user

If you want to keep the same Informix® OS users, you can configure Oracle® for OS authentication, and create users with the IDENTIFIED EXTERNALLY option.

In order to match operating system user names, the prefix for database user names must be set to an empty string '' and remote:

```
SQL> ALTER SYSTEM SET os_authent_prefix = ''; SCOPE = spfile;
System altered.
SQL> ALTER SYSTEM SET remote_os_authent = TRUE SCOPE = spfile;
SQL> quit
```

After changing the OS_AUTHENT_PREFIX parameter, restart the Oracle server instance and check the settings:

```
SQL> SHOW PARAMETER os_authent_prefix;
NAME                TYPE            VALUE
----------------------------------------------
os_authent_prefix   string

SQL> SHOW PARAMETER remote_os_authent;
NAME                TYPE            VALUE
----------------------------------------------
remote_os_authent   boolean         TRUE
```

Then create the user (note that the user name must be in uppercase letters), and grant permissions:

```
SQL> CREATE USER "SF" IDENTIFIED EXTERNALLY;
User created.
SQL> GRANT CONNECT, RESOURCE TO "SF";
Grant succeeded.
```

To connect your Genero program to Oracle® as an external user, perform the CONNECT TO instruction without specifying the USER/USING clauses.

For example, omit the USER/USING clause in the CONNECT TO instruction:

```
CONNECT TO "orc1fox+driver='dbmora'"
```

If no db login is specified, the Oracle® driver will open a database session with the OCI_CRED_EXT credentials.

Note: You can check if external or RDBMS authentication takes place with the FGLSQLDEBUG output (check the line containing "Credential flag").

Switching to the application schema

If several DB users are defined for the application, you might want to switch to a common schema with the following FGLPROFILE entry:

```
dbi.database.mydb.ora.schema = "app_owner"
```
Connecting as SYSDBA or SYSOPER

An Oracle® connection can also be established as SYSDBA or SYSOPER users. This is possible by specifying the /SYSDBA or /SYSOPER strings after the user name in the USER clause of the CONNECT TO instruction.

For example:

```
CONNECT TO "orc1fox+driver='dbmora'"
    USER "orauser/SYSDBA" USING "fourjs"
```

Using proxy authentication

If Oracle® proxy authentication is required, specify the /PROXY_CLIENT:username string after the user name in the USER clause of the CONNECT TO instruction.

For example:

```
CONNECT TO "orc1fox+driver='dbmora'"
    USER "orauser/PROXY_CLIENT:appuser" USING "fourjs"
```

In the above example, the credentials of the orauser login will be used to establish the connection, and then Oracle® will automatically switch to the user "appuser", assuming that the proxy connection has been granted with:

```
ALTER USER appuser GRANT CONNECT THROUGH orauser
```

Related concepts

Database users and security on page 608
Properly identifying database users allows to use database security and audit features.

Oracle DB specific FGLPROFILE parameters on page 673

Setting privileges

Informix®

Informix® users must have at least the CONNECT privilege to access the database:

```
GRANT CONNECT TO username
```

Application administration users need the RESOURCE privilege to create tables:

```
GRANT RESOURCE TO username
```

Since version 7.20, Informix® supports database roles:

```
GRANT rolename TO username
```

ORACLE

ORACLE supports the concept of roles to group privileges which then can be assigned to users.

ORACLE users do not have to explicitly set a role, they are assigned to a default privilege domain (set of roles). More than one role can be enabled at a time with ORACLE.

Informix® database privileges do NOT correspond exactly to ORACLE CONNECT, RESOURCE and DBA roles. However, roles can be created with equivalent privileges.
ORACLE users must have at least the CREATE SESSION privilege to access the database. This privilege is part of the CONNECT role:

```
GRANT CONNECT TO (PUBLIC | username)
```

**Solution**

Create a role which groups Informix® CONNECT privileges, and assign this role to the application users:

```
CREATE ROLE ifx_connect IDENTIFIED BY oracle;
GRANT CREATE SESSION, ALTER SESSION, CREATE ANY VIEW, ... TO ifx_connect;
GRANT ifx_connect TO user1;
```

**Data dictionary**

Oracle Database related data dictionary topics.

**Data type conversion table: Informix to Oracle**

**Table 253: Data type conversion table (Informix to Oracle)**

<table>
<thead>
<tr>
<th>Informix® data types</th>
<th>ORACLE data types</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR(n)</td>
<td>CHAR(n) (max is 2000 bytes)</td>
</tr>
<tr>
<td>VARCHAR(n[,m])</td>
<td>VARCHAR2(n) (max is 4000 bytes)</td>
</tr>
<tr>
<td>LVARCHAR(n)</td>
<td>VARCHAR2(n) (max is 4000 bytes)</td>
</tr>
<tr>
<td>NCHAR(n)</td>
<td>NCHAR(n) (max is 2000 bytes)</td>
</tr>
<tr>
<td>NVARCHAR(n[,m])</td>
<td>NVARCHAR2(n) (max is 4000 bytes)</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>CHAR(1)</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>NUMBER(5,0)</td>
</tr>
<tr>
<td>INTEGER</td>
<td>NUMBER(10,0)</td>
</tr>
<tr>
<td>BIGINT</td>
<td>NUMBER(20,0)</td>
</tr>
<tr>
<td>INT8</td>
<td>NUMBER(20,0)</td>
</tr>
<tr>
<td>SERIAL[(start)]</td>
<td>NUMBER(10,0) (see note 1)</td>
</tr>
<tr>
<td>BIGSERIAL[(start)]</td>
<td>NUMBER(20,0) (see note 1)</td>
</tr>
<tr>
<td>SERIAL8[(start)]</td>
<td>NUMBER(20,0) (see note 1)</td>
</tr>
<tr>
<td>DOUBLE PRECISION / FLOAT[(n)]</td>
<td>BINARY_DOUBLE</td>
</tr>
<tr>
<td>REAL / SMALLFLOAT</td>
<td>BINARY_FLOAT</td>
</tr>
<tr>
<td>DECIMAL(p,s)</td>
<td>NUMBER(p,s)</td>
</tr>
<tr>
<td>DECIMAL(p)</td>
<td>FLOAT(p*3.32193)</td>
</tr>
<tr>
<td>DECIMAL (not recommended)</td>
<td>FLOAT</td>
</tr>
<tr>
<td>MONEY (p,s)</td>
<td>NUMBER(p,s)</td>
</tr>
<tr>
<td>MONEY (p)</td>
<td>NUMBER(p,2)</td>
</tr>
<tr>
<td>MONEY</td>
<td>NUMBER(16,2)</td>
</tr>
<tr>
<td>TEXT</td>
<td>CLOB (max is 2Gb)</td>
</tr>
<tr>
<td>Informix® data types</td>
<td>ORACLE data types</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>BYTE</td>
<td>BLOB (max is 2Gb)</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME YEAR TO YEAR</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME YEAR TO MONTH</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME YEAR TO DAY</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME YEAR TO HOUR</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME YEAR TO SECOND</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME YEAR TO FRACTION(n)</td>
<td>TIMESTAMP(n)</td>
</tr>
<tr>
<td>DATETIME MONTH TO MONTH</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME MONTH TO DAY</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME MONTH TO HOUR</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME MONTH TO MINUTE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME MONTH TO SECOND</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME MONTH TO FRACTION(n)</td>
<td>TIMESTAMP(n)</td>
</tr>
<tr>
<td>DATETIME DAY TO DAY</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME DAY TO HOUR</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME DAY TO MINUTE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME DAY TO SECOND</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME DAY TO FRACTION(n)</td>
<td>TIMESTAMP(n)</td>
</tr>
<tr>
<td>DATETIME HOUR TO HOUR</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME HOUR TO MINUTE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME HOUR TO SECOND</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME HOUR TO FRACTION(n)</td>
<td>TIMESTAMP(n)</td>
</tr>
<tr>
<td>DATETIME MINUTE TO MINUTE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME MINUTE TO SECOND</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME MINUTE TO FRACTION(n)</td>
<td>TIMESTAMP(n)</td>
</tr>
<tr>
<td>DATETIME SECOND TO SECOND</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME SECOND TO FRACTION(n)</td>
<td>TIMESTAMP(n)</td>
</tr>
<tr>
<td>DATETIME FRACTION TO FRACTION(n)</td>
<td>TIMESTAMP(n)</td>
</tr>
<tr>
<td>INTERVAL YEAR[(p)] TO MONTH</td>
<td>INTERVAL YEAR[(p)] TO MONTH</td>
</tr>
<tr>
<td>INTERVAL MONTH[(p)] TO MONTH</td>
<td>CHAR(50)</td>
</tr>
<tr>
<td>INTERVAL DAY[(p)] TO FRACTION(n)</td>
<td>INTERVAL DAY[(p)] TO SECOND(n)</td>
</tr>
<tr>
<td>INTERVAL HOUR[(p)] TO HOUR</td>
<td>CHAR(50)</td>
</tr>
<tr>
<td>INTERVAL HOUR[(p)] TO MINUTE</td>
<td>CHAR(50)</td>
</tr>
<tr>
<td>INTERVAL HOUR[(p)] TO SECOND</td>
<td>CHAR(50)</td>
</tr>
<tr>
<td>Informix® data types</td>
<td>ORACLE data types</td>
</tr>
<tr>
<td>----------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>INTERVAL HOUR[(p)] TO FRACTION(n)</td>
<td>CHAR(50)</td>
</tr>
<tr>
<td>INTERVAL MINUTE[(p)] TO MINUTE</td>
<td>CHAR(50)</td>
</tr>
<tr>
<td>INTERVAL MINUTE[(p)] TO SECOND</td>
<td>CHAR(50)</td>
</tr>
<tr>
<td>INTERVAL MINUTE[(p)] TO SECOND TO FRACTION(n)</td>
<td>CHAR(50)</td>
</tr>
<tr>
<td>INTERVAL SECOND[(p)] TO SECOND</td>
<td>CHAR(50)</td>
</tr>
<tr>
<td>INTERVAL SECOND[(p)] TO FRACTION(n)</td>
<td>CHAR(50)</td>
</tr>
<tr>
<td>INTERVAL FRACTION[(p)] TO FRACTION</td>
<td>CHAR(50)</td>
</tr>
</tbody>
</table>

Notes:

1. For more details about serial emulation, see SERIAL and BIGSERIAL data types on page 931.

**BOOLEAN data type**

**Informix®**

Informix® supports the BOOLEAN data type, which can store 't' or 'f' values.

Genero BDL implements the BOOLEAN data type in a different way: A BOOLEAN variable stores integer values 1 or 0 (for TRUE or FALSE). This type is designed to hold the result of a boolean expression.

**ORACLE**

Oracle does not implement a native BOOLEAN type in SQL types.

However, a BOOLEAN type exists in the PL/SQL language.

**Solution**

The Oracle database interface converts the BOOLEAN type to CHAR(1) columns and stores '1' or '0' values in the column.

The BOOLEAN type translation can be controlled with the following FGLPROFILE entry:

```
dbi.database.dsnname.ifxemul.datatype.boolean = { true | false }
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

Using portable data types on page 614

Only a limited set of data types are really portable across several database engines.

**CHAR and VARCHAR data types**

**Informix®**

Informix® supports the following character data types:

**Table 254: Informix® character data types**

<table>
<thead>
<tr>
<th>Informix® data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR(n)</td>
<td>SBCS and MBCS character data (max is 32767 bytes)</td>
</tr>
<tr>
<td>VARCHAR(n[,m])</td>
<td>SBCS and MBCS character data (max is 255 bytes)</td>
</tr>
</tbody>
</table>
Informix®, both CHAR/VARCHAR and NCHAR/NVARCHAR data types can be used to store single-byte or multibyte encoded character strings. The only difference between CHAR/VARCHAR and NCHAR/NVARCHAR is in how they use sorting: N[VAR]CHAR types use the collation order, while [VAR]CHAR types use the byte order.

The character set used to store strings in CHAR/VARCHAR/NCHAR/NVARCHAR columns is defined by the DB_LOCALE environment variable.

The character set used by applications is defined by the CLIENT_LOCALE environment variable.

Informix® uses Byte Length Semantics (the size N that you specify in [VAR]CHAR(N) is expressed in bytes, not characters as in some other databases)

**ORACLE**

Oracle® supports the following data types to store character data:

**Table 255: Oracle® character data types**

<table>
<thead>
<tr>
<th>Oracle® data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR(n)</td>
<td>SBCS or MBCS character data using the database character set, where n is specified in bytes or characters, based on the length semantics (max is 2000 bytes)</td>
</tr>
<tr>
<td>VARCHAR2(n)</td>
<td>SBCS or MBCS character data using the database character set, where n is specified in bytes or characters, based on the length semantics (max is 4000 bytes)</td>
</tr>
<tr>
<td>NCHAR(n)</td>
<td>SBCS or MBCS character data using the national character set, where n is specified in bytes or characters, based on the length semantics (max is 2000 bytes)</td>
</tr>
<tr>
<td>NVARCHAR2(n)</td>
<td>SBCS or MBCS character data using the national character set, where n is specified in bytes or characters, based on the length semantics (max is 4000 bytes)</td>
</tr>
</tbody>
</table>

**Note:** Oracle® 12c introduced extended character types with the MAX_STRING_SIZE=EXTENDED server parameter. Use VARCHAR2 type can get a size up to 32Kb when MAX_STRING_SIZE=EXTENDED is set. However, the storage technique used by Oracle® 12c for such a large string type is different from the native/standard VARCHAR2 (4000) type. Large character strings will be stored as LOBs. Extended character types are not supported by Genero's Oracle® database driver.

In Oracle® CHAR(N) / VARCHAR2(N) types, the size N can be specified in character or byte units, depending on length semantics settings. See Length semantics settings on page 521 for more details.

When comparing CHAR and VARCHAR2 values in Oracle, the trailing blanks are significant; this is not the case when using Informix® VARCHAR columns. However, before comparing string values, Oracle® blank-pads CHAR(N) data to the maximum length of both operands. As result, it looks like trailing blanks are not significant in CHAR(N) comparison. For example, a column defined as CHAR(5) with the value 'abc ' (with 2 trailing blanks) will not be equal to 'abc', but when comparing (col = 'abc'), Oracle® will add 2 blanks to the right operand and values will match. Blank padding does not occur for VARCHAR2() data, as result, the expression (col = 'abc') will be false, if col VARCHAR2 does not exactly contain the value 'abc'. For more details, see blank-padded and non-padded comparison semantics in Oracle® documentation.
Oracle® treats empty strings like NULL values; Informix® doesn’t. See issue Empty Character Strings for more details.

With Oracle, you can define a Database Character Set and a National Character Set. Oracle® uses the Database Character Set to store string data in the CHAR/VARCHAR2 columns, and uses the National Character Set for NCHAR/NVARCHAR2 columns.

**Solution**

Informix® CHAR (N) types must be mapped to Oracle® CHAR (N) types, and Informix® VARCHAR (N) or LVARCHAR (N) columns must be mapped to Oracle® VARCHAR2 (N).

Check that your database tables do not use CHAR, VARCHAR or LVARCHAR types with a length exceeding the Oracle® limits of CHAR/VARCHAR2.

When using a multibyte character set (such as UTF-8), configure Oracle® to use character length semantics, define CHAR/VARCHAR2 database columns with a size in character units, and use character length semantics in BDL programs with FGL_LENGTH_SEMANTICS=CHAR. See Length semantics settings on page 521 for more details.

When extracting a database schema from an Oracle® database, the fgl dbsch schema extractor uses the size of the column in characters, not the octet length. If you have created a CHAR(10 (characters) ) column a in the database, the .sch file will get a size of 10, that will be interpreted according to FGL_LENGTH_SEMANTICS as a number of bytes or characters.

The Oracle® client character set must correspond to the Genero runtime system locale (LANG/LC_ALL). You can define the Oracle® client character set with the NLS_LANG environment variable. See also the section about Localization.

The CHAR/VARCHAR type translation can be controlled with the following FGLPROFILE entries:

```
  dbi.database.dsname.ifxemul.datatype.char = ↓ true ↓ false ↓
  dbi.database.dsname.ifxemul.datatype.varchar = ↓ true ↓ false ↓
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

CHAR and VARCHAR types on page 616
Using the CHAR and VARCHAR data types with different databases.

**Numeric data types**

**Informix®**

Informix® supports several data types to store numbers:

**Table 256: Informix® numeric data types**

<table>
<thead>
<tr>
<th>Informix® data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>16 bit signed integer</td>
</tr>
<tr>
<td>INTEGER</td>
<td>32 bit signed integer</td>
</tr>
<tr>
<td>BIGINT</td>
<td>64 bit signed integer</td>
</tr>
<tr>
<td>INT8</td>
<td>64 bit signed integer (replaced by BIGINT)</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>Equivalent to DECIMAL(16)</td>
</tr>
<tr>
<td>DECIMAL(p)</td>
<td>Floating-point decimal number (max precision is 32)</td>
</tr>
<tr>
<td>DECIMAL(p, s)</td>
<td>Fixed-point decimal number (max precision is 32)</td>
</tr>
</tbody>
</table>
### Informix® data type

<table>
<thead>
<tr>
<th>Informix® data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MONEY</td>
<td>Equivalent to DECIMAL(16,2)</td>
</tr>
<tr>
<td>MONEY(p)</td>
<td>Equivalent to DECIMAL(p,2) (max precision is 32)</td>
</tr>
<tr>
<td>MONEY(p,s)</td>
<td>Equivalent to DECIMAL(p,s) (max precision is 32)</td>
</tr>
<tr>
<td>REAL / SMALLFLOAT</td>
<td>32-bit floating point decimal (C float)</td>
</tr>
<tr>
<td>DOUBLE PRECISION / FLOT[(n)]</td>
<td>64-bit floating point decimal (C double)</td>
</tr>
</tbody>
</table>

### ORACLE

Oracle® supports following data types to store numbers:

#### Table 257: Oracle® numeric data types

<table>
<thead>
<tr>
<th>Oracle® data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMBER(p,s) (1&lt;=p&lt;=38, -84&lt;=s&lt;=127)</td>
<td>Fixed point decimal numbers.</td>
</tr>
<tr>
<td>NUMBER(p) (1&lt;=p&lt;=38)</td>
<td>Integer numbers with a precision of p digits.</td>
</tr>
<tr>
<td>NUMBER(*,s)</td>
<td>Fixed point decimal numbers with a precision of 38 digits.</td>
</tr>
<tr>
<td>NUMBER</td>
<td>Floating point decimals with a precision of 38 digits.</td>
</tr>
<tr>
<td>FLOAT(b) (1&lt;=b&lt;=126)</td>
<td>Floating point numbers with a binary precision b. This is a sub-type of NUMBER.</td>
</tr>
<tr>
<td>BINARY_FLOAT (since Oracle® 10g)</td>
<td>32-bit floating point number.</td>
</tr>
<tr>
<td>BINARY_DOUBLE (since Oracle® 10g)</td>
<td>64-bit floating point number.</td>
</tr>
</tbody>
</table>

The type names SMALLINT, INTEGER are supported by Oracle®. However, these will be converted to the native NUMBER(*,0) type. When dividing INTEGER or SMALLINT types, Informix® rounds the result (7 / 2 = 3), while Oracle® doesn't, because it does not have a native integer data type (7 / 2 = 3.5)

The DECIMAL type name is also supported by Oracle®, and is mapped to the native NUMBER type. When using a precision and scale (DECIMAL(p,s)), the resulting NUMBER(p,s) type is equivalent. However, DECIMAL(p) becomes a NUMBER(p,0), and DECIMAL without precision / scale becomes a NUMBER(*,0) both storing whole numbers. This is different from Informix SQL, where a DECIMAL (with or without precision) can store real numbers.

**Important:** Oracle NUMBER without precision / scale can store real numbers, while Oracle DECIMAL will store whole numbers. For example:

```sql
CREATE TABLE t1 ( num NUMBER, dec DECIMAL );
INSERT INTO t1 VALUES ( 123.456, 123.456 );
SELECT * FROM t1;
NUM    DEC
------- --------
123.456 123
```

### Solution

Use the following conversion rules to map Informix® numeric types to Oracle® numeric types:
Table 258: Informix® numeric types and Oracle® equivalents

<table>
<thead>
<tr>
<th>Informix® data type</th>
<th>Oracle® data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>NUMBER(5,0)</td>
</tr>
<tr>
<td>INTEGER</td>
<td>NUMBER(10,0)</td>
</tr>
<tr>
<td>BIGINT</td>
<td>NUMBER(20,0)</td>
</tr>
<tr>
<td>INT8</td>
<td>NUMBER(20,0)</td>
</tr>
<tr>
<td>DECIMAL(p,s)</td>
<td>NUMBER(p,s)</td>
</tr>
<tr>
<td>DECIMAL(p)</td>
<td>FLOAT(p * 3.32193)</td>
</tr>
<tr>
<td>DECIMAL (not recommended)</td>
<td>FLOAT</td>
</tr>
<tr>
<td>MONEY(p,s)</td>
<td>NUMBER(p,s)</td>
</tr>
<tr>
<td>MONEY(p)</td>
<td>NUMBER(p,2)</td>
</tr>
<tr>
<td>MONEY</td>
<td>NUMBER(16,2)</td>
</tr>
<tr>
<td>SMALLFLOAT</td>
<td>BINARY_FLOAT</td>
</tr>
<tr>
<td>FLOAT[(p)]</td>
<td>BINARY_DOUBLE</td>
</tr>
</tbody>
</table>

Avoid dividing integers in SQL statements. If you do divide an integer, use the TRUNC() function with Oracle®.

When creating a table directly in Oracle's sqlplus with the INTEGER, SMALLINT types, Oracle will create columns with the native NUMBER(38,0) type. As result, it is not possible (for fgldbsch) to distinguish the original type names used in CREATE TABLE from the native NUMBER(38,0) type or NUMBER(38) type (where scale defaults to zero). In the next example, all columns will be of type NUMBER(38,0):

```
$ sqlplus ...
sql> CREATE TABLE mytab (  
col1 INTEGER,  
col2 SMALLINT,  
col3 NUMBER(38),  
...
```

When extracting the database schema with fgldbsch, NUMBER, NUMBER(p>32) and NUMBER(p>32,s) types will by default give an extraction error. However, these types can be converted to DECIMAL(32) and DECIMAL(32,s) with the -cv option, by using the "B" character at positions 22 (for NUMBER) and 23 (for NUMBER(p>32[s])).

**Note:** When fetching a NUMBER[(p>32,s)] into a BDL DECIMAL(32[l,s]) type, if the value stored in the NUMBER column has more than 32 digits, it will be rounded to fit into a DECIMAL(32), or the overflow error -1226 will occur when fetching into a DECIMAL(32,s). Note that it must be allowed to fetch numeric expressions such as 1/3 (=0.333333333333....) into a DECIMAL(p,s), even if such expression will produce more than 32 digits with Oracle.

When creating a table in a BDL program with DECIMAL(p), this type is converted to native Oracle® FLOAT(p*3.32193). When creating a table in a BDL program with DECIMAL (without precision) this type is converted to native Oracle® FLOAT. The native Oracle® FLOAT[(p)] type can be extracted by fgldbsch, but Oracle's FLOAT has a higher precision than the BDL DECIMAL type, which can lead to value rounding when fetching rows.

With Oracle® versions older than 10g, when creating tables in a BDL program with SMALLFLOAT or FLOAT types, these types are mapped to NUMBER (The native Oracle® FLOAT(b) type could have been used, but this type is reserved to map DECIMAL(p) types). Starting with Oracle® 10g, SMALLFLOAT or FLOAT types will respectively be converted to BINARY_FLOAT and BINARY_DOUBLE native Oracle® types, which can be extracted by fgldbsch and mapped back to BDL SMALLFLOAT and FLOAT respectively in the .sch file.
When casting a numeric expression such as `CAST(SUM(col) AS DECIMAL)`, with Informix SQL this results in a real numeric value. With Oracle® the DECIMAL type becomes a NUMBER(*,0) and the result is a whole number. The DECIMAL type name in the CAST() expression is not translated by the Oracle drivers because this type conversion is only done for DDL statement (CREATE TABLE / ALTER TABLE).

**Note:** As a general recommendation, do not use `DECIMAL[(p)]` or `SMALLFLOAT/FLOAT` floating point types in business applications. These types should only be used for scientific data storage.

The numeric types translation can be controlled with the following FGLPROFILE entries:

```
<table>
<thead>
<tr>
<th>Configuration</th>
<th>True</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbi.database.dsnname.ifxemul.datatype.smallint</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dbi.database.dsnname.ifxemul.datatype.integer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dbi.database.dsnname.ifxemul.datatype.bigint</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dbi.database.dsnname.ifxemul.datatype.int8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dbi.database.dsnname.ifxemul.datatype.decimal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dbi.database.dsnname.ifxemul.datatype.money</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dbi.database.dsnname.ifxemul.datatype.float</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dbi.database.dsnname.ifxemul.datatype.smallfloat</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

**Using portable data types** on page 614

Only a limited set of data types are really portable across several database engines.

**fgldbsch** on page 2079

The `fgldbsch` tool generates the database schema files from an existing database.

**DATE and DATETIME data types**

**Informix®**

Informix® provides two data types to store date and time information:

- **DATE** = for year, month and day storage.
- **DATETIME** = for year to fraction (1-5) storage.

The DATE type is stored as an INTEGER with the number of days since 1899/12/31.

The DATETIME type can be defined with various time units, by specifying a start and end qualifier. For example, you can define a datetime to store an hour-to-second time value with `DATETIME HOUR TO SECOND`.

The values of Informix® DATETIME can be represented with a character string literal, or as DATETIME() literals:

```
'2017-12-24 15:45:12.345'  -- a DATETIME YEAR TO FRACTION(3)
'15:45'   -- a DATETIME HOUR TO MINUTE
DATETIME(2017-12-24 12:45) YEAR TO MINUTE
DATETIME(12:45:56.333) HOUR TO FRACTION(3)
```

Informix® is able to convert quoted strings to DATE/DATETIME data, if the string contains matching environment parameters. The string to date conversion rules for DATE is defined by the DBDATE environment variable. The string to datetime format for DATETIME is defined by the GL_DATETIME environment variable.

**Note:** Within Genero programs, the string representation for DATETIME values is always ISO (YYYY-MM-DD hh:mm:ss.ffffff)

Informix® supports date arithmetic on DATE and DATETIME values. The result of an arithmetic expression involving dates/times is an INTEGER number of days when only DATE values are used, and an INTERVAL value if a DATETIME is used in the expression.

Informix® automatically converts an INTEGER to a DATE when the integer is used to set a value of a date column.

Informix® provides the CURRENT q1 TO q2 operator, to get the system date/time on the server where the current database is located. When no qualifiers are specified, CURRENT returns a DATETIME YEAR TO
Informix also supports the SYSDATE operator, which returns the current system time as a DATETIME YEAR TO FRACTION(5).

Note: The USEOSTIME configuration parameter must be set to 1 in order to get the subsecond precision in CURRENT and SYSDATE operators. See Informix documentation for more details.

**ORACLE**

Oracle® provides the following data types to store date and time data:

- **DATE** = for year, month, day, hour, min, second storage.
- **TIMESTAMP** = for year, month, day, hour, min, second, fraction storage.

As in Informix®, Oracle® can convert quoted strings to DATE or TIMESTAMP data if the contents of the string matches the NLS date format parameters (NLS_DATE_FORMAT, NLS_TIMESTAMP_FORMAT).

The **TO_DATE()** and **TO_TIMESTAMP()** SQL functions convert strings to dates or timestamps, based on a given format. The **TO_CHAR()** SQL function allows you to convert dates or timestamps to strings, according to a given format.

In Oracle® the result of an arithmetic expression involving DATE values is a number of days as NUMBER type; the decimal part is the fraction of the day (0.5 = 12H00, 2.00694444 = (2 + (10/1440)) = 2 days and 10 minutes). The result of an expression involving Oracle® TIMESTAMP data is of type INTERVAL.

To compare dates that have time data in Oracle®, you can use the **ROUND()** or **TRUNC()** SQL functions.

Even if the keyword is the same, the Oracle® SYSDATE operator (using second precision) is not the exact equivalent of the Informix® SYSDATE operator (using subsecond precision). Use Oracle's SYSTIMESTAMP as equivalent for Informix SYSDATE operator.

See the Oracle® documentation for more details.

**Solution**

Use the following conversion rules to map Informix® date/time types to Oracle® date/time types:

**Table 259: Informix® data types and Oracle® equivalents**

<table>
<thead>
<tr>
<th>Informix® data type</th>
<th>Oracle®</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME YEAR TO YEAR</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME YEAR TO MONTH</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME YEAR TO DAY</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME YEAR TO HOUR</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME YEAR TO MINUTE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME YEAR TO SECOND</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME YEAR TO FRACTION(n)</td>
<td>TIMESTAMP(n)</td>
</tr>
<tr>
<td>DATETIME MONTH TO MONTH</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME MONTH TO DAY</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME MONTH TO HOUR</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME MONTH TO MINUTE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME MONTH TO SECOND</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME MONTH TO FRACTION(n)</td>
<td>TIMESTAMP(n)</td>
</tr>
</tbody>
</table>
## Informix® data type vs. Oracle®

<table>
<thead>
<tr>
<th>Informix® data type</th>
<th>Oracle®</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATETIME DAY TO DAY</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME DAY TO HOUR</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME DAY TO MINUTE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME DAY TO SECOND</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME DAY TO FRACTION(n)</td>
<td>TIMESTAMP(n)</td>
</tr>
<tr>
<td>DATETIME HOUR TO HOUR</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME HOUR TO MINUTE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME HOUR TO SECOND</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME HOUR TO FRACTION(n)</td>
<td>TIMESTAMP(n)</td>
</tr>
<tr>
<td>DATETIME MINUTE TO MINUTE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME MINUTE TO SECOND</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME MINUTE TO FRACTION(n)</td>
<td>TIMESTAMP(n)</td>
</tr>
<tr>
<td>DATETIME SECOND TO SECOND</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME SECOND TO FRACTION(n)</td>
<td>TIMESTAMP(n)</td>
</tr>
<tr>
<td>DATETIME FRACTION TO FRACTION(n)</td>
<td>TIMESTAMP(n)</td>
</tr>
</tbody>
</table>

The `DATE` and `DATETIME` types translation can be controlled with the following FGLPROFILE entries:

```plaintext
  dbi.database.dsname.ifxemul.datatype.date = \{ true | false \}
  dbi.database.dsname.ifxemul.datatype.datetime = \{ true | false \}
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

### Storing BDL DATE values

The Oracle® `DATE` type is used to store Genero BDL `DATE` values. However, keep in mind that the Oracle® `DATE` type stores also time (hh:mm:ss) information. The database interface automatically sets the time part to midnight (00:00:00) during input/output operations.

You must be very careful since manual modifications of the database might set the time part, for example:

```sql
UPDATE table SET date_col = SYSDATE
```

(SYSDATE is equivalent to `CURRENT YEAR TO SECOND` in Informix®).

After this type of update, when columns have date values with a time part different from midnight, some `SELECT` statements might not return all the expected rows.

When fetching Oracle® `DATE` values into Genero BDL `DATE` or `DATETIME` variables, the date and time information is directly set for the individual date/time parts and the conversion is straight forward. But when fetching an Oracle® `DATE` into a `CHAR` or `VARCHAR` variable, date to string conversion occurs. Since Oracle® dates are equivalent of Informix® `DATETIME YEAR TO SECOND`, the values are by default converted with the ISO format (YYYY-MM-DD hh:mm:ss), which is not the typical Informix® behavior where dates are formatted from the DBDATE environment variable. If your application fetches `DATE` values into `CHAR/VARCHAR` and you want to get the `DBDATE` conversion, you must set the following FGLPROFILE entry:

```plaintext
  dbi.database.dbname.ora.date.ifxfetch = true
```

**Note:** Oracle® does not support `INTEGER` to `DATE` automatic conversion.
Storing BDL DATETIME values

Informix® DATETIME data with any precision from YEAR to SECOND is stored in Oracle® DATE columns. The database interface makes the conversion automatically. Missing date or time parts default to 1900-01-01 00:00:00. For example, when using a DATETIME HOUR TO MINUTE with the value of "11:45", the Oracle® DATE value will be "1900-01-01 11:45:00".

Informix® DATETIME YEAR TO FRACTION(n) data is stored in Oracle® TIMESTAMP columns. The TIMESTAMP data type can store up to 9 digits in the fractional part, and therefore can store all precisions of Informix® DATETIME.

Important:

• Most arithmetic expressions involving dates (for example, to add or remove a number of days from a date) will produce the same result with Oracle®. But keep in mind that Oracle® evaluates date arithmetic expressions to NUMBER (days.fraction) while Informix® evaluates to INTEGER when only DATE values are used in the expression, or to INTERVAL values if at least one DATETIME is used in the expression.

• Even if a configuration parameter exists to get the Informix® behavior, avoid fetching date values into CHAR or VARCHAR, to bypass the DBDATE / ISO format conversion difference with Oracle®.

Date/time SQL functions

Table 260: Informix® and Oracle® date/time SQL functions

<table>
<thead>
<tr>
<th>Informix®</th>
<th>Oracle®</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>trunc( sysdate )</td>
</tr>
<tr>
<td>current year to second</td>
<td>sysdate</td>
</tr>
<tr>
<td>sysdate</td>
<td>systimestamp</td>
</tr>
<tr>
<td>day( value )</td>
<td>to_number( to_char( value, 'dd' ) )</td>
</tr>
<tr>
<td>extend( dtvalue, first to last )</td>
<td>to_date( nvl( to_char( dtvalue, 'fmt-mask' ), '19000101000000' ), 'fmt-mask' )</td>
</tr>
<tr>
<td>mdy(m,d,y)</td>
<td>to_date( to_char(m,'09')</td>
</tr>
<tr>
<td>month( date )</td>
<td>to_number( to_char( date, 'mm' ) )</td>
</tr>
<tr>
<td>weekday( date )</td>
<td>to_number( to_char( date, 'd' ) ) -1</td>
</tr>
<tr>
<td>year( date )</td>
<td>to_number( to_char( date, 'yyyy' ) )</td>
</tr>
<tr>
<td>date( &quot;string&quot;</td>
<td>integer )</td>
</tr>
</tbody>
</table>

Related concepts

Date/time literals in SQL statements on page 635

Good practices for date and time handling in SQL.

INTERVAL data type

Informix®

Informix® provides the INTERVAL data type to store a value that represents a span of time.

INTERVAL types are divided into two classes:

• year-month intervals. For example: INTERVAL YEAR(5) TO MONTH
• *day-time* intervals. For example: INTERVAL DAY(9) TO SECOND

INTERVAL columns can be defined with various time units, by specifying a start and end qualifier. For example, you can define an interval to store a number of hours and minutes with INTERVAL HOUR\(n\) TO MINUTE, where \(n\) defines the maximum number of digits for the hours unit.

The values of Informix® INTERVAL can be represented with a character string literal, or as INTERVAL() literals:

```plaintext
'-9834 15:45:12.345'  -- an INTERVAL DAY(6) TO FRACTION(3)
'7623-11'   -- an INTERVAL YEAR(9) TO MONTH
INTERVAL(18734:45) HOUR(5) TO MINUTE
INTERVAL(-7634-11) YEAR(5) TO MONTH
```

**ORACLE**

ORACLE provides an INTERVAL data type similar to Informix®, implementing two classes (YEAR TO MONTH and DAY TO SECOND).

However, Oracle’s intervals cannot be defined with a time units different from the two interval classes. For example, you cannot define an INTERVAL HOUR TO MINUTE in Oracle.

**Note:** The ORACLE INTERVAL DAY TO SECOND \(n\) contains the fractional part of seconds and therefore is equivalent to the Informix® INTERVAL DAY TO FRACTION\(n\) type.

**Solution**

Informix® INTERVAL YEAR\(n\) TO MONTH data is stored in Oracle INTERVAL YEAR\(n\) TO MONTH columns. These data types are equivalent.

Informix® INTERVAL DAY\(n\) TO FRACTION\(p\) data is stored in Oracle INTERVAL DAY\(n\) TO SECOND\(p\) columns. These data types are equivalent.

Other Informix® INTERVAL types must be stored in CHAR\()\) columns, because the high qualifier precision cannot be specified with Oracle INTERVALs. For example, Informix® INTERVAL HOUR(5) TO MINUTE has no native equivalent in Oracle.

The INTERVAL types translation can be controlled with the following FGLPROFILE entry:

```plaintext
dbi.database.dsnname.ifxemul.datatype.interval = \true \false
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

*Using portable data types* on page 614

Only a limited set of data types are really portable across several database engines.

**SERIAL and BIGSERIAL data types**

**Informix®**

Informix® supports the SERIAL, BIGSERIAL data types to produce automatic integer sequences:

• SERIAL can produce 32 bit integers (INTEGER)
• BIGSERIAL can produce 64 bit integers (BIGINT)
• SERIAL8 is a synonym for BIGSERIAL

Steps to use serials with Informix®:

1. Create the table with a column using SERIAL, or BIGSERIAL.
2. To generate a new serial, no value or a zero value is specified in the INSERT statement:

   ```sql
   INSERT INTO tbl1 ( c ) VALUES ( 'aa' )
   ```
**3.** After `INSERT`, the new value of a `SERIAL` column is provided in `SQLCA.SQLERRD[2]`, while the new value of a `BIGSERIAL` value must be fetched with a `SELECT dbinfo('bigserial')` query.

Informix® allows you to insert rows with a value different from zero for a serial column. Using an explicit value will automatically increment the internal serial counter, to avoid conflicts with future `INSERT` statements that are using a zero value:

```sql
CREATE TABLE tab ( k SERIAL); -- internal counter = 0
INSERT INTO tab VALUES ( 0 ); -- internal counter = 1
INSERT INTO tab VALUES ( 10 ); -- internal counter = 11
DELETE FROM tab; -- internal counter = 11
INSERT INTO tab VALUES ( 0 ); -- internal counter = 12
```

**ORACLE**

Oracle® provides several solutions to implement auto-incremented columns:

1. Sequence objects can be created to generate numbers (`CREATE SEQUENCE, seqname.currval`).
2. Since Oracle® 12c, it is possible to reference a sequence in `DEFAULT ON NULL` column clauses.
3. Since Oracle® 12c, you can define columns with the `GENERATE ... AS IDENTITY` clause. However, with pure Oracle SQL, there is no way to get the last generated number. This is only possible with PL/SQL by using the `RETURNING ... INTO ...` clause.

Details about Oracle® sequences:

- Sequences are totally detached from tables.
- The purpose of sequences is to provide unique integer numbers.
- Sequences are identified by a sequence name.
- To create a sequence, you must use the `CREATE SEQUENCE` statement. Once a sequence is created, it is permanent (like a table).
- To get a new sequence value, you must use the `nextval` keyword, preceded by the name of the sequence. The `seqname.nextval` expression can be used in `INSERT` statements:

```sql
INSERT INTO tab1 VALUES ( tab1_seq.nextval, ... )
```

- To get the last generated number, Oracle® provides the `currval` keyword:

```sql
SELECT seqname.currval FROM DUAL
```

- In order to improve performance, Oracle® can handle a set of sequences in the cache (See `CREATE SEQUENCE` syntax in the Oracle® documentation).

**Solution**

**Note:** For best SQL portability when using different types of databases, consider using sequences as described in **Solution 3: Use native SEQUENCE database objects** on page 628.

The `SERIAL` data type can be emulated with sequences used in `INSERT` triggers or with the `DEFAULT ON NULL` clause.

The method used to emulate `SERIAL` types is defined by the `ifxemul.datatype.serial.emulation FGLPROFILE` parameter:

```sql
dbi.database.dbname.ifxemul.datatype.serial.emulation
  = _"native"_"native2"_"regtable"
```

- `native`: uses insert triggers with sequences.
- `native2`: uses `DEFAULT ON NULL` column clause with sequences.
- regtable: uses insert triggers with the SERIALREG table.

The default emulation technique is "native".

**Note:** Genero does use Oracle's GENERATED ... AS IDENTITY feature to emulate Informix SERIALs, because it is not possible to easily get the last generated/inserted number, in order to fill SQLCA.SQLERRD[2].

The serial type emulation can be enabled or disabled with the following FGLPROFILE entries:

```
dbi.database.dbname.ifxemul.datatype.serial = [true|false]
dbi.database.dbname.ifxemul.datatype.serial8 = [true|false]
dbi.database.dbname.ifxemul.datatype.bigserial = [true|false]
```

**Important:** The "regtable" emulation based on the SERIALREG table is provided to simplify the migration from Informix. We strongly recommend that you use the native or native2 method instead. The "native2" method is the fastest solution when inserting a large number of rows in the database.

**Disabling automatic serial retrieval for SQLCA.SQLERRD[2]**

SERIAL emulation can be totally disabled by setting the ifxemul.datatype.serial FGLPROFILE entry to false:

```
dbi.database.dbname.ifxemul.datatype.serial = false
```

For Informix® compatibility, after an INSERT statement, the ODI drivers automatically execute another SQL query (or do a DB client API call when possible), to get the last generated serial, and fill the SQLCA.SQLERRD[2] register. This results in some overhead that can be avoided, if the SQLCA.SQLERRD[2] register is not used by the program.

When SERIAL emulation is required (to create temp tables with a serial column during program execution), and the SQLCA.SQLERRD[2] register does not need to be filled, (typically because you use your own method to retrieve the last generated serial), you can set the ifxemul.datatype.serial.sqlerrd2 FGLPROFILE entry to false. This will avoid the automatic retrieval of last serial value to fill SQLCA.SQLERRD[2]:

```
dbi.database.dbname.ifxemul.datatype.serial.sqlerrd2 = false
```

See also `db_get_last_serial()` on page 2308.

**Notes common to all serial emulation modes**

When a BDL program executes a CREATE [TEMP] TABLE with a SERIAL column, the Oracle® interface automatically creates the additional SQL objects (column clauses, sequences or triggers) to generate numbers when an INSERT statement is performed.

Users executing programs which create tables with SERIAL columns must have the CONNECT and RESOURCE roles assigned to create triggers and sequences.

SERIAL[(n)] data types are converted to NUMBER(10,0), while BIGSERIAL[(n)] is replaced by NUMBER(20,0).

For SERIAL types, the SQLCA.SQLERRD[2] register is filled as expected with the last generated serial value. However, since SQLCA.SQLERRD[2] is defined as an INTEGER, it cannot hold values from BIGSERIAL (NUMBER(20,0)) auto-incremented columns. If you are using BIGSERIAL columns, you must fetch the sequence pseudo-column CURR_VAL or fetch the LASTSERIAL column from the SERIALREG table, if used.

Check whether your application uses tables with a SERIAL column that can contain a NULL value: INSERT statements using NULL for the SERIAL column will produce a new serial value:

```
INSERT INTO tab ( col1, col2 ) VALUES ( NULL, 'data' )
```
This behavior is mandatory in order to support INSERT statements that do not use the serial column:

```
INSERT INTO tab (col2) VALUES ('data')
```

For SQL portability, it is recommended to review INSERT statements to remove the SERIAL column from the list. For example, the following statement:

```
INSERT INTO tab (col1,col2) VALUES (0, p_value)
```

can be converted to:

```
INSERT INTO tab (col2) VALUES (p_value)
```

Static SQL INSERT using records defined from the schema file must also be reviewed:

```
DEFINE rec LIKE tab.*
INSERT INTO tab VALUES (rec.*) -- will use the serial column
```

can be converted to:

```
INSERT INTO tab VALUES rec.* -- without parentheses, serial column is removed
```

When using the Static SQL INSERT or UPDATE syntax using record.* without parentheses, make sure that your database schema files contain information about serials. This information can be lost when extracting the schema from an Oracle® database. See Database Schema for more details about the serial flag in column type encoding (data type code must be 6).

If the "native" or "regtable" emulation is used, inserting rows with Oracle® tools like SQL*Plus or SQL*Loader will execute the INSERT triggers. When loading big tables, you can disable triggers with ALTER TRIGGER [ENABLE | DISABLE] (see Oracle® documentation for more details). After reactivation of the serial triggers, the SERIAL sequences must be re-initialized (use serialpkg.create_sequence('tab','col')) or re-execute the PL/SQL script containing the sequence and trigger creation.

**Using the native serial emulation**

Each table having a SERIAL column needs an INSERT TRIGGER and a SEQUENCE dedicated to SERIAL generation.

To know how to write those sequences and triggers, you can create a small Genero program that creates a table with a SERIAL column. Set the FGLSQLDEBUG environment variable and run the program. The debug output will show you the native SQL commands to create the sequence and the trigger.

For temporary tables, the trigger and the sequence are dropped automatically after a "DROP TABLE temptab" or when the program disconnects from the database.

**Using the native2 serial emulation**

With this emulation, a SERIAL type is converted to a DEFAULT ON NULL clause using a sequence created automatically by the database driver, for example:

```
CREATE TABLE t1 ( mykey SERIAL(100), .... )
```

is converted to:

```
CREATE SEQUENCE t1_srl INCREMENT BY 1 START WITH 100
CREATE TABLE t1 (mykey NUMBER(10,0) DEFAULT ON NULL t1_srl.nextval , ...
```
For **temporary tables**, the sequence is dropped automatically after a "DROP TABLE temptab" or when the program disconnects from the database.

**Note:** The native2 serial emulation uses the DEFAULT ON NULL clause, supported by Oracle, starting from version 12.1. When using this serial emulation with an Oracle database version prior to 12.1, a CREATE TABLE statement using a serial will produce the SQL error -6370.

### Using the regtable serial emulation

Each table having a SERIAL column needs an INSERT TRIGGER which uses the SERIALREG table dedicated to SERIAL registration.

First, you must prepare the database and create the SERIALREG table as follows:

```sql
CREATE TABLE serialreg (  
    tablename VARCHAR2(50) NOT NULL,  
    lastserial NUMBER(20,0) NOT NULL,  
    PRIMARY KEY ( tablename )  
);
```

**Important:** This table must exist in the database before creating the serial triggers.

In database creation scripts, all SERIAL[(n)] data types must be converted to INTEGER data types and you must create one trigger for each table. SERIAL8/BIGSERIAL columns must be converted to NUMBER(20,0). To know how to write those triggers, you can create a small Genero program that creates a table with a SERIAL column. Set the FGLSQLDEBUG environment variable and run the program. The debug output will show you the native trigger creation command.

The serial production is based on the SERIALREG table which registers the last generated number for each table. If you delete rows of this table, sequences will restart at start values and you might get duplicated values.

For **temporary tables**, the trigger is dropped automatically after a "DROP TABLE temptab" or when the program disconnects from the database.

### Related concepts

- **Auto-incremented columns (serials)** on page 626
  How to implement automatic record keys.
- **FGLPROFILE entries for core language** on page 256
  This is a summary of FGLPROFILE entries supported by the core BDL.
- **The SQLCA diagnostic record** on page 594
  The SQLCA variable is a predefined record containing SQL statement execution information.
- **Debugging SQL statements** on page 603
  The runtime system can display debug information for SQL statements executed by the program.

### ROWID columns

**Informix®**

When creating a table, Informix® automatically adds a ROWID integer column (applies to non-fragmented tables only).

The **ROWID** column is auto-filled with a unique number and can be used like a primary key to access a given row.

**Note:** Informix® ROWID usage was a common practice in the early days of Informix® 4GL programming. Today it is recommended to define all your database tables with a PRIMARY KEY to uniquely identify rows.

With Informix®, the SQLCA.SQLERRD[6] register contains the ROWID of the last row affected by an INSERT, UPDATE or DELETE statement.
**Oracle**

Oracle® supports ROWIDs, but the data type is different from Informix® ROWIDs: Oracle® rowids are `CHAR(18)`.

For example:

```
AAAA8mAALAAAAQkAAA
```

Since Oracle® rowids are physical addresses, they cannot be used as permanent row identifiers: After a DELETE, an INSERT statement might reuse the physical place of the deleted row, to store the new row.

The Oracle Call Level (OCI) provides functions to get the rowid of the last row related to an INSERT, UPDATE, DELETE statement, ora FETCH of a cursor declared with a SELECT FOR UPDATE statement.

**Solution**

If your Genero BDL application uses Informix® rowid columns, review the program logic to use the primary keys when available, or use Oracle® rowids.

To hold Oracle rowids, the type of variables containing ROWID values must be changed to `CHAR(18)`.

**Note:** Informix® INTEGER rowids fit in a `CHAR(18)` variable.

When connected to an Oracle® database, all references to `SQLCA.SQLERRD[6]` must be reviewed, because this register can not contain the rowid of the last affected row. However, after an INSERT, UPDATE or DELETE statement, or after a FETCH of a cursor declared with a SELECT FOR UPDATE, the `SQLCA.SQLERRM` register is filled with the character extended (base 64) representation of the rowid of the last affected or fetched row.

**Note:** The `SQLCA.SQLERRM` register only contains a single rowid, even if multiple rows were affected by the INSERT, UPDATE or DELETE statement.

For databases where the keyword of the rowid pseudo-column is different than "ROWID", the translation can be controlled with the following FGLPROFILE entry:

```
dbi.database.dsnname.ifxemul.rowid = \ true \ false 
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

- [Using ROWID columns](#) on page 644
- Automatic ROWID columns is not a common database feature.

**The RAW data type**

ORACLE supports the RAW data type to hold binary data. This data type is for example used to return values from the `SYS_GUID()` SQL function.

**Solution**

The ORACLE RAW values can be converted to a character string in the hexadecimal notation.

When fetching rows from the database, the database driver will automatically convert ORACLE RAW values to hexadecimal. On the other hand, when using SQL parameters, the database driver will convert hexadecimal VARCHAR strings to binary data.

Since each byte is represented with two characters in the hexadecimal notation, you must define a VARCHAR(N*2) variable to hold the values of a native RAW(N) column.

When extracting a database schema with the `fgldbsch` tool, the ORACLE RAW(N) type is converted to VARCHAR2(N*2).
TEXT and BYTE (LOB) types

**Informix**
Informix® provides the TEXT, BYTE, CLOB and BLOB data types to store very large texts or binary data. Legacy Informix® 4GL applications typically use the TEXT and BYTE types.
Genero BDL does not support the Informix® CLOB and BLOB types.

**ORACLE**
ORACLE provides CLOB, BLOB, and BFILE data types to store large text and binary data.
The ORACLE large object types are similar to Informix® LOB types.

**Solution**
The ORACLE database interface can convert BDL TEXT data to ORACLE CLOB and BYTE data to ORACLE BLOB.
Genero TEXT/BYTE program variables have a limit of 2 gigabytes. Make sure that the large object data does not exceed this limit.
The ORACLE BFILE type is not supported by Genero BDL.
The TEXT and BYTE types translation can be controlled with the following FGLPROFILE entries:

```
dbi.database.dsnname.ifxemul.text = { true | false }
dbi.database.dsnname.ifxemul.byte = { true | false }
```
For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**
**Using portable data types** on page 614
Only a limited set of data types are really portable across several database engines.

**Table constraints**

**Informix**
Informix® supports primary key, unique, foreign key, default and check constraints.
The constraint naming syntax is different in Informix® and most other databases: Informix expects the constraint name after the constraint definition:

```
CREATE TABLE emp (  
  ...  
  emp_code CHAR(10) UNIQUE CONSTRAINT pk_emp,  
  ...  
)
```
While other SQL database brands require to specify the constraint name before the constraint definition:

```
CREATE TABLE emp (  
  ...  
  emp_code CHAR(10) CONSTRAINT pk_emp UNIQUE,  
  ...  
)
```

**ORACLE**
ORACLE supports primary key, unique, foreign key, default and check constraints.
The constraint naming clause must be placed before the constraint specification.

**Primary keys**

Like Informix®, ORACLE creates an index to enforce PRIMARY KEY constraints (some RDBMS do not create indexes for constraints). Using CREATE UNIQUE INDEX to define unique constraints is obsolete (use primary keys or a secondary key instead).

**Unique constraints**

Like Informix®, ORACLE creates an index to enforce UNIQUE constraints (some RDBMS do not create indexes for constraints).

When using a unique constraint, Informix® allows only one row with a NULL value, while ORACLE allows several rows with NULL! Using CREATE UNIQUE INDEX is obsolete.

**Foreign keys**

Both Informix® and ORACLE support the ON DELETE CASCADE option. To defer constraint checking, Informix® provides the SET CONSTRAINT command while ORACLE provides the ENABLE and DISABLE clauses.

**Check constraints**

The check condition may be any valid expression that can be evaluated to TRUE or FALSE, including functions and literals. You must verify that the expression is not Informix® specific.

**Null constraints**

Informix® and ORACLE support not null constraints, but Informix® does not allow you to give a name to NOT NULL constraints.

**Constraint naming syntax**

The database interface does not convert constraint naming expressions when creating tables from BDL programs. Review the database creation scripts to adapt the constraint naming clauses for ORACLE.

**Related concepts**

- **Data definition statements** on page 613
- It is recommended to avoid use of DDL in programs.

**Name resolution of SQL objects**

**Informix®**

Informix® uses the following form to identify an SQL object:

```
database[@dbservername]:][{owner|"owner"}].identifier
```

The ANSI convention is to use double quotes for identifier delimiters (For example: "customer"."cust_name").

Informix® database object names are not case-sensitive in non-ANSI databases. When using double-quoted identifiers, Informix® becomes case sensitive.

With non-ANSI Informix® databases, you do not have to give a schema name before the tables when executing an SQL statement:

```
SELECT ... FROM customer WHERE ...
```

In Informix® ANSI compliant databases:

- The table name must include "owner", unless the connected user is the owner of the database object.
• The database server shifts the owner name to uppercase letters before the statement executes, unless the owner name is enclosed in double quotes.

**ORACLE**

With Oracle®, an object name takes the following form:

```sql
[(schema|"schema")](identifier|"identifier")[@database-link]
```

Oracle® has separate namespaces for different classes of objects (tables, views, triggers, indexes, clusters).

Object names are limited to 30 chars in ORACLE.

Unlike Informix®, Oracle® database object names are stored in UPPERCASE in system catalogs. That means that `SELECT "col1" FROM "tab1"` will produce an error because those objects are identified by "COL1" and "TAB1" in Oracle® system catalogs.

An Oracle® database schema is owned by a user (usually, the application administrator) and this user must create PUBLIC SYNONYMS to provide a global scope for his table names. PUBLIC SYNONYMS can have the same name as the schema objects they point to.

**Solution**

To write portable SQL, regarding database object names:

1. Use simple database object names (without any owner/schema prefix)
2. Do not use double quotes to surround database object identifiers.
3. If needed, define public synonyms to reference database objects in others databases/schema.
4. Specify database object identifiers in lowercase.

See also Naming database objects on page 637.

Without double quotes around the database object names, all names will be converted to uppercase letters by ORACLE before executing the SQL.

Check that you do not use single-quoted or double-quoted table names or column names in your source. Those quotes must be removed because the database interface automatically converts double quotes to single quotes, and Oracle® does not allow single quotes as database object name delimiters.

See also the issue Database Concepts

**NULLs in indexed columns**

Oracle btree indexes do not store null values, while Informix® btree indexes do. This means that if you index a single column and select all the rows where that column is null, Informix® will do an indexed read to fetch just those rows, but Oracle will do a sequential scan of all rows to find them. Having an index unusable for "is null" criteria can also completely change the behavior and performance of more complicated selects without causing a sequential scan.

**Solution**

Declare the indexed columns as NOT NULL with a default value and change the program logic. If you do not want to change the programs, partitioning the table so that the nulls have a partition of their own will reduce the sequential scan to just the nulls (un-indexed) partition, which is relatively fast.

**Data manipulation**

Oracle Database related data manipulation topics.

**Reserved words**

**Informix®**

With Informix®, it is possible to create database objects with reserved words.
For example:

```sql
CREATE TABLE table ( char CHAR(10) );
```

Indeed this is not good practice, but Informix® SQL allows this to be backward compatible when introducing a new keyword in the SQL syntax.

Most other database systems do not allow reserved words as database identifiers. If your legacy code is using SQL reserved words of the target database SQL syntax, an error will be thrown at CREATE TABLE execution.

**ORACLE**

SQL object names like table and column names cannot be SQL reserved words in ORACLE.

An example of a common word which is part of the ORACLE SQL grammar is 'level'.

**Solution**

Table or column names which are ORACLE reserved words must be renamed.

ORACLE reserved keywords are listed in the ORACLE documentation, or Oracle 8i provides the V $RESERVED_WORDS view to track Oracle reserved words. All BDL application sources must be verified. To check if a given keyword is used in a source, you can use UNIX™ 'grep' or 'awk' tools. Most modifications can be done automatically with UNIX™ tools like 'sed' or 'awk'.

**Outer joins**

**Informix® OUTER() syntax**

In Informix® SQL, outer joins can be defined in the FROM clause with the OUTER keyword:

```sql
SELECT ... FROM a, OUTER (b)  
WHERE a.key = b.akey
```

```sql
SELECT ... FROM a, OUTER(b,OUTER(c))  
WHERE a.key = b.akey  
AND b.key1 = c.bkey1 AND b.key2 = c.bkey2
```

Informix® also supports the ANSI OUTER join syntax, which is the recommended way to specify outer joins with recent SQL database engines:

```sql
SELECT ... FROM cust LEFT OUTER JOIN order  
ON cust.key = order.custno
```

**ORACLE**

Oracle supports the ANSI outer join syntax:

```sql
SELECT ...  
FROM cust LEFT OUTER JOIN order  
    LEFT OUTER JOIN item  
    ON order.key = item.ordno  
    ON cust.key = order.custno  
WHERE order.cdate > current date
```

The ORACLE legacy outer join specification uses the (+) notation.
The (+) symbol must be placed after columns of the tables which must have NULL values when no record matches the condition:

```
SELECT ... FROM a, b WHERE a.key = b.key (+)
```

```
SELECT ... FROM a, b, c WHERE a.key = b.akey (+)
  AND b.key1 = c.bkey1 (+)
  AND b.key2 = c.bkey2 (+)
```

When using additional conditions on outer tables, the (+) operator also has to be used. For example:

```
SELECT ... FROM a, OUTER(b) WHERE a.key = b.akey AND b.colx > 10
```

Must be converted to:

```
SELECT ... FROM a, b WHERE a.key = b.akey (+)
  AND b.colx (+) > 10
```

The ORACLE outer joins restriction:

In a query that performs outer joins of more than two pairs of tables, a single table can only be the NULL generated table for one other table. The following case is not allowed: `WHERE a.col = b.col (+) AND b.col (+) = c.col`

**Solution**

The Genero database drivers can convert Informix Informix® OUTER specifications to ANSI outer joins.

**Note:** For better SQL portability, use the ANSI outer join syntax instead of the old Informix® OUTER syntax.

The outer join translation can be controlled with the following FGLPROFILE entry:

```
dbi.database.dsname.ifxemul.outers = {true | false}
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**• Prerequisites:**

1. In the FROM clause, the main table must be the first item and the outer tables must be listed from left to right in the order of outer levels.

   Example which does not work:

   ```
   ... FROM OUTER(tab2), tab1
   ```

2. The outer join in the WHERE clause must use the table name as prefix:

   ```
   ... WHERE tab1.col1 = tab2.col2
   ```

**• Restrictions:**

1. Statements composed by 2 or more SELECT instructions are not supported:

   ```
   SELECT ... UNION SELECT ...
   ```

   or:

   ```
   SELECT ... WHERE col IN (SELECT...)
   ```

2. Additional conditions on outer table columns cannot be detected and therefore are not supported:

   ```
   ... FROM tab1, OUTER(tab2)
   WHERE tab1.col1 = tab2.col2
   ```
AND tab2.colx > 10

3. Using subscript in outer conditions:

... FROM tab1, OUTER(tab2)
    WHERE tab1.col1[1,3] = tab2.col2[1,3]

• Notes:

1. Table aliases are detected in OUTER expressions.

OUTER example with table alias:

... OUTER(tab1 alias1) ...

2. In the outer join, outertab.col can be placed on both right or left sides of the equal sign:

... WHERE outertab.col1 = maintab.col2

3. Table names detection is not case-sensitive:

SELECT ... FROM tab1, TAB2
    WHERE tab1.col1 = tab2.col2

4. Temporary tables are supported in OUTER specifications:

CREATE TEMP TABLE tt1 ( ... )
SELECT ... FROM tab1, OUTER(tt1) ...

Related concepts

Outer joins on page 643
Use standard ISO outer join syntax instead of the old IBM® Informix® OUTER() syntax.

Transactions handling

Informix®

With the Informix® native mode (non ANSI):

• Transactions blocks start with BEGIN WORK and terminate with COMMIT WORK or ROLLBACK WORK.
• Statements executed outside a transaction are automatically committed.
• DDL statements can be executed (and canceled) in transactions.

UPDATE tab1 SET ... -- auto-committed
BEGIN WORK -- start of TX block
UPDATE tab1 SET ...
UPDATE tab2 SET ...
...
COMMIT WORK -- end of TX block

Informix® version 11.50 introduces savepoints:

SAVEPOINT name [UNIQUE]
ROLLBACK [WORK] TO SAVEPOINT [name]
RELEASE SAVEPOINT name

ORACLE

With ORACLE transactions:

• Beginnings of transactions are implicit.
• A transaction ends with a COMMIT or ROLLBACK statement.
• The current transaction is automatically committed when a DDL statement is executed.

```
UPDATE tab1 SET ... -- start of TX block
UPDATE tab2 SET ...
...               -- end of TX block
COMMIT
```

ORACLE supports savepoints too. However, there are differences:

• Savepoints cannot be declared as UNIQUE
• Rollback must always specify the savepoint name
• You cannot release savepoints (RELEASE SAVEPOINT)

**Solution**

Regarding transaction control instructions, BDL applications do not have to be modified in order to work with ORACLE. The Informix® behavior is simulated with an autocommit mode in the ORACLE interface. A switch to the explicit commit mode is done when a `BEGIN WORK` is performed by the BDL program.

When executing a DDL statement inside a transaction, ORACLE automatically commits the transaction. Therefore, you must extract the DDL statements from transaction blocks.

If you want to use savepoints, do not use the UNIQUE keyword in the savepoint declaration, always specify the savepoint name in ROLLBACK TO SAVEPOINT, and do not drop savepoints with RELEASE SAVEPOINT.

See also **SELECT FOR UPDATE**

**Related concepts**

**Database transactions** on page 600

Database transactions define a set of SQL instructions to be executed as a whole, or rolled back as a whole.

**Temporary tables**

**Informix®**

Informix® temporary tables are created with the `CREATE TEMP TABLE` DDL instruction or with `SELECT ... INTO TEMP` statement:

```
CREATE TEMP TABLE tt1 ( pkey INT, name VARCHAR(50) )
CREATE TEMP TABLE tt2 ( pkey INT, name VARCHAR(50) ) WITH NO LOG
SELECT * FROM tab1 WHERE pkey > 100 INTO TEMP tt2
```

Temporary tables are automatically dropped when the SQL session ends, but they can also be dropped with the DROP TABLE command. There is no name conflict when several users create temporary tables with the same name.

BDL REPORTs can create a temporary table when the rows are not sorted externally (by the source SQL statement). Informix® allows you to create indexes on temporary tables. No name conflict occurs when several users create an index on a temporary table by using the same index identifier.

When creating temporary tables in Informix®, the WITH NO LOG clause can be used to avoid the overhead of recording DML operations in transaction logs.

**Oracle**

Oracle® 11g and 12c supports global temporary tables that can be shared among several processes: Only data is temporary and local to an SQL process, the table structure is common to all programs.

Oracle 18c provides private temporary tables, similar to Informix temporary tables: The table structure and data is private to the program. However, private temporary tables in Oracle 18c have several limitations that need to be considered before usage.
Solution

In accordance with some prerequisites, temporary table creation in BDL programs can be supported by the database interface.

**Important:** When creating a temporary table, you perform a Data Definition Language statement. Oracle® automatically commits the current transaction when executing a DDL statement. Therefore, you must avoid temporary table creation/destruction in transactions.

The general FGLPROFILE entry to control temporary table emulation is:

```
  dbi.database.dsname.ifxemul.temptables = { true | false }
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Important:**

Simple Informix-style SQL statement creating temporary tables can be converted to a native SQL equivalent instruction. However, complex SQL statements such as `SELECT .. INTO TEMP` with subqueries may fail. In such cases, create a view from the complex query and then create the temp table from the view. Or, disable Informix emulation and use the native SQL syntax to create the temporary table (EXECUTE IMMEDIATE "/* fglhint_no_ifxemul */ ...")

With Informix SQL, if the source table has a column defined as SERIAL or BIGSERIAL, a `SELECT ... INTO TEMP` will produce a new temp table with an auto-incremented serial column. With the `SELECT ... INTO TEMP` emulation for non-Informix databases, not using the native sequence generators (such as IDENTITY columns in SQL Server), the resulting temporary table will get a simple INTEGER or BIGINT column, instead of an auto-incremented column.

The temporary table emulation can use permanent tables, global temporary tables or (since Oracle 18c) private temporary tables.

Use the following FGLPROFILE entry, to define the type of temporary table emulation:

```
  dbi.database.dbname.ifxemul.temptables.emulation = { "default" |
  "private" |
  "global" }
```

By default, the database driver uses regular permanent tables. The default emulation provides maximum compatibility with Informix® temporary tables, but requires real table creation which can be a significant overhead with Oracle®. For more details, see Using the default temporary table emulation on page 944.

The "private" temp table emulation uses native Oracle® Private Temporary Tables, which are only visible for the current SQL session. However, the private emulation mode has some limitations and constraints requiring code review. For more details, see Using the private temporary table emulation on page 949.

The "global" temp table emulation uses native Oracle® Global Temporary Tables, requiring only one initial table creation and thus making programs run faster. However, the global emulation mode has some limitations requiring code review. For more details, see Using the global temporary table emulation on page 947.

**Related concepts**

Temporary tables on page 641

Syntax for temporary table creation is not unique across all database engines.

Oracle DB specific FGLPROFILE parameters on page 673

*Using the default temporary table emulation*

The default temporary table emulation is provided to achieve a high level of Informix® compatibility, when minimal code change is required.
In order to use the default temporary table emulation, define the following FGLPROFILE entry (or leave it unset):

```
  dbi.database.dbname.ifxemul.temptables.emulation = "default"
```

**How does the default temp table emulation work?**

- Informix® `CREATE TEMP TABLE` and `SELECT INTO TEMP` statements are automatically converted to Oracle® "CREATE TABLE". The name of the temporary table is converted to a unique table name.
- By default, Oracle tables are created in the current schema (i.e. no schema prefix is added to the generated table name). In order to control the schema where the Oracle tables are created, use the following FGLPROFILE parameters:

  ```
  dbi.database.dsname.ora.temptables.schema.source = \"login\" \n  dbi.database.dsname.ora.temptables.schema.command = \"SELECT ...\"
  ```

  When specifying "login" in the `ora.temptables.schema.source` parameter, the user name passed to the connection instruction will be used as schema name. When specifying "command" as schema source, the schema name will be produced by the SELECT statement defined in the `ora.temptables.schema.command` parameter.

  Schema specification for temporary table emulations is mandatory when you have created a DB user for each end user, and you force common schema usage with the `ora.schema` parameter:

  ```
  dbi.database.dsname.ora.schema = "app_owner"
  ```

  In this case, it is better to add the current user as schema name for temporary tables, to get Oracle tables created in the current user schema, instead of creating tables in the common schema: Creating tables in another schema requires CREATE ANY TABLE / DROP ANY TABLE privileges for each DB user, which is not possible in an organization using strong security policy.

- By default, temporary tables are created in a dedicated tablespace named "TEMPTABS". Of course the TEMPTABS tablespace must exist before running programs, otherwise temporary table creation will fail. Using a specific tablespace for temporary tables allows you to specify storage options, for example to use a physical device which can be different from the disk drive used for real data storage. Additionally, backups of permanent application tables can be performed without the data of temporary tables.

  If required, you can specify another tablespace name for Oracle tables with the following FGLPROFILE entry:

  ```
  dbi.database.dsname.ora.temptables.tablespace = "mytemptabs"
  ```

  **Note:** This tablespace must be a permanent tablespace.

- Starting with Oracle 10g, dropped tables are saved in the recycle bin by default. You may want to avoid the recycle bin feature at the database level or session level with:

  ```
  ALTER SYSTEM SET recyclebin = OFF scope=spfile
  ```

  or, at current session level only:

  ```
  ALTER SESSION SET recyclebin = OFF
  ```

- Once the temporary table has been created, all other SQL statements performed in the current SQL session are parsed to convert the original table name to the corresponding unique table name.
- When the BDL program disconnects from the database (for example, when it ends or when a CLOSE DATABASE instruction is executed), the tables which have not been removed with an explicit "DROP TABLE" are automatically removed by the database interface. However, if the program crashes, the tables will remain in the database, so you may need to cleanup the database from time to time.
**Prerequisites when using the default temp table emulation**

- Application users must have sufficient privileges to create database tables in the schema used to create the Oracle table (usually, "CONNECT" and "RESOURCE" roles).
- Create a tablespace dedicated to the Oracle tables created by this emulation. The default tablespace is named "TEMPTABS", but it can be another tablespace defined by the `ora.temptables.tablespace` FGLPROFILE entry as described before.

The tablespace must be of type "permanent", as it will hold permanent tables used to emulate Informix® temp tables.

Make sure the tablespace is big enough to hold all the data, and check for automatic extension.

When using a PDB, the tablespace must be created in the context of the PDB.

```sql
CREATE TABLESPACE temptabs
  DATAFILE 'file-path' SIZE 1M AUTOEXTEND ON;
-- Give privileges on temptabs tablespace to other users
ALTER USER dbuser QUOTA UNLIMITED ON TEMPTABS;
```

For more details, see "CREATE TABLESPACE" in the Oracle documentation.

**Limitations of the default temp table emulation**

- When using the default emulation, the real name of an emulated temporary table will have the following format:

  `tt<number>_original_name`

  Where `number` is the Oracle AUDSID session id returned by:

  ```sql
  SELECT USERENV('SESSIONID') FROM DUAL
  ```

  As Oracle 9i and 10g table names cannot exceed 30 characters in length, and since session ids are persistent over server shutdown, you must pay attention to the names of your temporary tables. For example, if you create a temp table with the name TEMP_CUSTOMER_INVOICES (22c) it leaves 30 - (3 + 22) = 5 characters left for the session id, which gives a limit of 99999 sessions.

  To work around this limitation, you can provide your own SQL command to generate a unique session id with the following FGLPROFILE entry:

  ```sql
  dbi.database.dbname.ora.sid.command = "select ..."
  ```

  As an example, you can use the SID column value from V$SESSION:

  ```sql
  SELECT SID FROM V$SESSION WHERE AUDSID = USERENV('SESSIONID')
  ```

- Application tables and columns cannot use the same table name as the name generated for temporary tables. Make sure you are not using table or column names with the format:

  `tt<number>_name`

- Tokens matching the original table names are converted to unique names in all SQL statements. Make sure you are not using the temp table name for other database objects, like columns. The following example illustrates this limitation:

  ```sql
  CREATE TABLE tab1 ( key INTEGER, tmp1 CHAR(20) )
  CREATE TEMP TABLE tmp1 ( col1 INTEGER, col2 CHAR(20) )
  SELECT tmp1 FROM tab1 WHERE ...
  ```
Maintenance of default temp table emulation

• If you want to list the tables created by a specific user, do this:

```
SELECT * FROM ALL_TABLES WHERE OWNER = 'user_name'
```

As with other database object names, the user name is stored in uppercase letters if it has been created without using double quotes (create user scott ... = stored name is "SCOTT").

Creating indexes on temporary tables with default temp table emulation

• Indexes created on temporary tables must have unique names also. The database interface detects CREATE INDEX statements which are using temporary tables and converts the index name to unique names. Indexes created on temporary tables follow the same storage and schema settings as for the CREATE TABLE statements: Schema and tablespace specification of FGLPROFILE will also be applied to CREATE INDEX commands.
• DROP INDEX statements are also detected to replace the original index name by the real name.

SERIALs in temporary table creation with default temp table emulation

• You can use the SERIAL data type when creating a temporary table.
  Sequences and triggers will be created in the current schema.

  See issue about SERIALs for more details.

Using the global temporary table emulation

The global temporary table emulation is provided to get the benefit of Oracle's GLOBAL TEMPORARY TABLES, by sharing the same table structure with multiple SQL sessions, reducing the cost of the CREATE TABLE statement execution. However, this emulation does not provide the same level of Informix compatibility as the default emulation, and must be used carefully.

In order to use the global temporary table emulation, define the following FGLPROFILE entry:

```
dbi.database.dbname.ifxemul.temptables.emulation = "global"
```

How does the global temp table emulation work?

• Informix CREATE TEMP TABLE and SELECT INTO TEMP statements are automatically converted to Oracle "CREATE GLOBAL TEMPORARY TABLE" statements. The original table name is kept, but it gets by default a "TEMPTABS" schema prefix, to share the underlying table structure with other database users.

In order to control the schema where the Oracle global temporary tables are created, use the following FGLPROFILE parameters:

```
  dbi.database.dsname.ora.temptables.schema.source = "login" \
  dbi.database.dsname.ora.temptables.schema.command = "SELECT ...
```

Note: The purpose of global temporary tables is to create the table once in Oracle and have all users share the same table structure. Therefore the default "TEMPTABS" schema is considered sufficient for this emulation method.

• The Global Temporary Tables are created with the "ON COMMIT PRESERVE ROWS" option, to keep the rows in the table when a transaction ends.

• If the global temporary table to be created exists already, error ORA-00955 will be ignored by the database driver. This allows to do several CREATE TEMP TABLE statements in your programs with no SQL error, to emulate the Informix behavior. This works fine as long as the table name is unique for a given structure (column count and data types must match).

  Note: Since DROP TABLE statements are converted to DELETE statements, you might want to use Oracle sqlplus to issue a real DROP TABLE of global temporary tables, to make a real cleanup from time to time.
• By default, global temporary table is created without any tablespace option, and thus will be created in the default tablespace assigned to the current user.

If required, you can specify another tablespace name for Oracle® tables with the following FGLPROFILE entry:

```plaintext
dbi.database.dsname.ora.temptables.tablespace = "mytemptabs"
```

**Note:** This tablespace must be a temporary tablespace.

• Once the Global Temporary Table has been created, all other SQL statements performed in the current SQL session are parsed to convert the original table name to `schema.original-tablename`.

• When doing a DROP TABLE `temp-table` statement in the program, the database driver converts it to a DELETE statement, to remove all data added by the current session. A next CREATE TEMP TABLE or SELECT INTO TEMP will fail with error ORA-00955 but since this error is ignored, it will be transparent for the program. We cannot use TRUNCATE TABLE because that would require at least DROP ANY TABLE privileges for all users.

• When the BDL program disconnects from the database (for example, when it ends or when a CLOSE DATABASE instruction is executed), the tables that have not been dropped by the program with an explicit DROP TABLE statement will be automatically cleaned by Oracle.

### Prerequisites when using the global temp table emulation

• You must create a database user (schema) dedicated to this emulation, the default name is “TEMPTABS”:

```sql
CREATE USER temptabs IDENTIFIED BY pswd;
```

• If your programs need to create temporary tables on the fly with this method, you must grant CREATE ANY TABLE + CREATE ANY INDEX system privilege to all DB users. This is not a good practice for security reasons. You better "prepare" the database by creating the Global Temporary Table (when using the default schema, create it with the TEMPTABS user). Do not forget to specify ON COMMIT PRESERVE ROWS option. Then grant INSERT, UPDATE, DELETE and SELECT object privileges to PUBLIC, for example:

```sql
CREATE GLOBAL TEMPORARY TABLE temptabs.mytable
  ( k INT PRIMARY KEY, c CHAR(10) ) ON COMMIT PRESERVE ROWS;
CREATE UNIQUE INDEX temptabs.ix1 ON temptabs.mytable ( c );
GRANT SELECT, UPDATE, INSERT, DELETE ON temptabs.mytable TO PUBLIC;
```

For testing purpose, consider using a user with DBA privileges, to simplify the configuration.

### Limitations of the global temp table emulation

• Global Temporary Tables are shared by multiple users/sessions. In order to have the global emulation working properly, each temporary table name must be unique for a given table structure, for all programs. Avoid using temp tables names such as "tmp1". It is recommended to use table names as follows:

```sql
CREATE TEMP TABLE custinfo_1 ( cust_id INTEGER, cust_name VARCHAR(50) );
CREATE TEMP TABLE custinfo_2 ( cust_id INTEGER, cust_name VARCHAR(50), cust_addr VARCHAR(200) );
```

• Tokens matching the original table names are converted to unique names in all SQL statements. Make sure you are not using the temp table name for other database objects, like columns. The following example illustrates this limitation:

```sql
CREATE TABLE tab1 ( key INTEGER, tmp1 CHAR(20) );
CREATE TEMP TABLE tmp1 ( col1 INTEGER, col2 CHAR(20) );
```
SELECT tmp1 FROM tab1 WHERE ...

Creating indexes on temporary tables with global temp table emulation

- Indexes created on temporary tables get also the TEMPTABS schema prefix by default. Indexes created on temporary tables follow the same storage and schema settings as for the CREATE GLOBAL TEMPORARY TABLE statements: Schema and tablespace specification of FGLPROFILE will also be applied to CREATE INDEX commands.
- When executing a DROP INDEX statement on a temporary table in a program, the database driver just ignores the statement.

SERIALs in temporary table creation with global temp table emulation

- You can use the SERIAL data type when creating a temporary table.
  Sequences and triggers will be created in the TEMPTABS schema too.
  See issue about SERIALs for more details.

Using the private temporary table emulation

The private temporary table emulation feature can be used to benefit from the native Oracle® 18c private temporary tables.

Important: Oracle® private temporary tables have several limitations compared to Informix® temporary tables. Using this feature requires code review to overcome these limitations.

In order to use private temporary table emulation, define the following FGLPROFILE entry:

```
dbi.database.dbname.ifxemul.temptables.emulation = "private"
```

How does the private temp table emulation work?

- Informix® CREATE TEMP TABLE and SELECT INTO TEMP statements are automatically converted to Oracle® "CREATE PRIVATE TEMPORARY TABLE".
- The original name of the temporary table must be converted to an Oracle private temporary table name (PTT) with a specific prefix, which is defined by the private_temp_table_prefix system parameter (default is ORA$PTT_).
  The detection of the PTT prefix can be controlled with FGLPROFILE entries, to get the value of private_temp_table_prefix. For more details, see Determine the private temporary table prefix on page 950
- Since Oracle private temporary tables are only visible to the current user and SQL session, it is not possible to specify a schema prefix.
- To mimic the behavior of Informix temporary tables, Oracle private temporary tables are created with the ON COMMIT PRESERVE DEFINITION clause. The temp table will then remain when a transaction ends.
- By default, private temporary tables are created in the default temporary table space. If needed, you can specify another tablespace name for Oracle private temporary tables with the following FGLPROFILE entry:

```
dbi.database.dsnname.ora.temptables.tablespace = "mytemptabs"
```

Note: This tablespace must be a temporary tablespace.
- Once the temporary table has been created, all other SQL statements performed in the current SQL session are parsed to convert the original table name to the corresponding private temporary table name.
- When the BDL program disconnects from the database, Oracle will automatically drop private temporary tables created during the SQL session.
Prerequisites when using the private temp table emulation

- By default (with Oracle 18.3), the maximum number of private temporary tables allowed per user is 16. Consider increasing this number, if your programs create more temporary tables. Otherwise, Oracle will produce the error "ORA-32460: maximum number of private temporary tables per session exceeded". The max number of private temp tables can be changed with the _ptt_max_num hidden system parameter:

```sql
ALTER SYSTEM SET "_ptt_max_num"=32 SCOPE = BOTH;
```

- For more details, check also "CREATE PRIVATE TEMPORARY TABLE" in the Oracle documentation.

Limitations of the private temp table emulation

- By default (in Oracle 18.3), the max number of private temp tables per user is 16. This can be changed by a system parameter.
- It is not possible to define table constraints such as NOT NULL or PRIMARY KEY, etc.
- Index creation is not allowed on private temp tables. Therefore, reports with internal ORDER BY cannot work.
- Columns can't have DEFAULT values. Thus, the native2 serial emulation cannot work.
- Triggers are not supported. Therefore, native and regtable serial emulations cannot work with private temp tables.
- There is no such concept as Informix CREATE TEMP TABLE ... WITH NO LOG. By default, Oracle private temporary table data is always logged for transaction rollback.
- Tokens matching the original table names are converted to unique names in all SQL statements. Make sure you are not using the temp table name for other database objects, like columns. The following example illustrates this limitation:

```sql
CREATE TABLE tab1 ( key INTEGER, tmp1 CHAR(20) )
CREATE TEMP TABLE tmp1 ( col1 INTEGER, col2 CHAR(20) )
SELECT tmp1 FROM tab1 WHERE ...
```

Serial emulation with private temporary tables

**Important:** Because Oracle private temporary tables do not support default value specifications (used for native2 serial emulation) and triggers (used for native and regtable serial emulations), it is not possible to emulate Informix serial types with Oracle private temporary tables.

Determine the private temporary table prefix

The default prefix for Oracle private temporary tables is "ORA$PTT_". This prefix can be changed by database administrators.

To get the current PTT prefix, the Oracle ODI driver can read the `private_temp_table_prefix` system parameter from the `v$parameter` system view:

```sql
SELECT value FROM v$parameter WHERE name = 'private_temp_table_prefix';
```

However, regular DB users do not have the permission to read this system view.

Therefore, the detection of the PTT prefix can be controlled with FGLPROFILE entries, to avoid the need to access the `v$parameter` system view.

The next FGLPROFILE entry defines the way to get the prefix of PTTs:

```json
dbi.database.mydb.ora.temptables.private.prefix.source = { "value" |
"command" }
```

When the source is defined as "value", the next FGLPROFILE entry defines the value of the PTT prefix:

```json
dbi.database.mydb.ora.temptables.private.prefix.value = "myprefix"
```
When the source is defined as "value", but no `dbi.database.dsnname.ora.private.prefix.value` entry is defined, the ODI driver uses "ORA$PTT_".

When the source is defined as "command", the next FGLPROFILE entry defines the SQL command to be executed to get the PTT prefix:

```sql
dbi.database.mydb.ora.temptables.private.prefix.command = "select 'myprefix' from dual"
```

When the source is defined as "command", but no `dbi.database.dsnname.ora.private.prefix.command` entry is defined, the ODI driver will try to read from `v$parameter` with:

```sql
select value from v$parameter where name = 'private_temp_table_prefix'
```

**Note:** Read privileges on the `v$parameter` system view is not allowed for regular DB users by default (GRANT SELECT permission must be performed for all DB users).

Otherwise, when none of the above entries are defined, the Oracle ODI driver uses "ORA$PTT_" by default. If the default PTT prefix was not changed in the database, regular DB users do not need GRANT SELECT permission on `v$parameter`.

**Related concepts**
- [SERIAL and BIGSERIAL data types](page 931)

**Substrings in SQL**

**Informix**

Informix® SQL statements can use subscripts on columns defined with the character data type:

```sql
SELECT ... FROM tab1 WHERE col1[2,3] = 'RO'
SELECT ... FROM tab1 WHERE col1[10] = 'R' -- Same as col1[10,10]
UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...
SELECT ... FROM tab1 ORDER BY col1[1,3]
```

**Important:** With other database servers as Informix®, when the subscript notation is used to modify column values in UPDATE statement, or as ORDER BY element, you will get an SQL error:

```sql
UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...
SELECT ... FROM tab1 ORDER BY col1[1,3]
```

**ORACLE**

ORACLE provides the `SUBSTR()` function, to extract a substring from a string expression:

```sql
SELECT .... FROM tab1 WHERE SUBSTR(col1,2,2) = 'RO'
SELECT SUBSTR('Some text',6,3)FROM DUAL -- Gives 'tex'
```

**Solution**

Replace all Informix® `col[x,y]` right-value expressions by `SUBSTR(col,x,y-x+1)`.

Rewrite UPDATE and ORDER BY clauses using `col[x,y]` expressions.

The translation of `col[x,y]` expressions can be controlled with the following FGLPROFILE entry:

```sql
dbi.database.dsnname.ifxemul.colsubs = \[true \| false \]
```

For more details see [IBM Informix emulation parameters in FGLPROFILE](page 668).
Related concepts
Substring expressions on page 644
Handle substrings expressions with different database engines.

The LENGTH() function

Informix®
Informix® provides the LENGTH() function to count the number of bytes of a character string expression:

```
SELECT LENGTH("aaa"), LENGTH(col1) FROM table
```

Informix® LENGTH() does not count the trailing blanks for CHAR or VARCHAR expressions, while Oracle counts the trailing blanks.
Informix® LENGTH() returns 0 when the given string is empty. That means, \( \text{LENGTH('')} = 0 \).

ORACLE
Oracle supports the LENGTH() function, but there are some differences with Informix® LENGTH().
The Oracle LENGTH() function counts trailing blanks. When using a CHAR column, values are blank padded, and the function returns the size of the CHAR column. When using a VARCHAR column, trailing blanks are significant, and the function returns the number of characters, including trailing blanks.
Because ORACLE handles empty strings ('') as NULL values, writing LENGTH('') is equivalent to LENGTH(NULL). In this case, the function returns NULL.

Solution
Check if the trailing blanks are significant when using the LENGTH() SQL function in your application.
To count the number of character by ignoring the trailing blanks, use the RTRIM() function:

```
SELECT LENGTH(RTRIM(col1)) FROM table
```

SQL conditions which verify that the result of LENGTH() is greater that a given number do not have to be changed, because the expression evaluates to false if the given string is empty (NULL>n):
```
SELECT * FROM x WHERE LENGTH(col)>0
```
Only SQL conditions that compare the result of LENGTH() to zero will not work if the column is NULL. You must check your BDL code for such conditions:
```
SELECT * FROM x WHERE LENGTH(col)=0
```
In this case, you must add a test to verify if the column is null:
```
SELECT * FROM x WHERE ( LENGTH(col)=0 OR col IS NULL )
```
In addition, when retrieving the result of a LENGTH() expression into a BDL variable, you must check that the variable is not NULL.
In ORACLE, you can use the NVL() function in order to get a non-null value:
```
SELECT * FROM x WHERE NVL(LENGTH(c),0)=0
```
Since Informix® supports the NVL() function, you can write the same SQL for both Informix® and ORACLE, as shown in this example.
Consider to create a user-defined function that implements the behavior of the Informix LENGTH() function:

```sql
CREATE OR REPLACE FUNCTION vlength(
    value IN VARCHAR2
) RETURN INTEGER
AUTHID CURRENT_USER
IS
    BEGIN
        RETURN NVL(LENGTH(RTRIM(value)),0);
    END;
/
```

The translation of `LENGTH()` expressions can be controlled with the following FGLPROFILE entry:

```
dbi.database.dsnname.ifxemul.length = true | false
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**
The `LENGTH()` function in SQL on page 647
The semantics of the `LENGTH()` SQL function differs according to the database engine.

**Empty character strings**

**Informix®**
Informix® SQL considers empty strings ("" or ") as a non-NULL string with a length of zero.

**Note:**
In Genero BDL, when setting a variable with an empty string constant, it is automatically set to a NULL value:

```sql
DEFINE x char(10)
LET x = ""
IF x IS NULL THEN -- evaluates to TRUE
    ...
END IF
```

**ORACLE**
Oracle® SQL considers empty string literals ("" ) as NULL.

Using literal string values that are empty ("") for INSERT or UPDATE statements will result in the storage of NULL values with Oracle®, while Informix® stores the value as a string with a length of zero:

```sql
INSERT INTO tab1 ( col1, col2 ) VALUES ( NULL, '' )
```

Using comparison operators (col="") with Oracle® makes no sense, because an empty string is equivalent to NULL: The correct SQL expression is (col IS NULL).

```sql
SELECT * FROM tab1 WHERE col2 IS NULL
```

With Oracle, we must also distinguish CHAR and VARCHAR2. CHARs are blank padded. In SQL, "" empty string literals are NULL, but in PL/SQL, assigning a CHAR variable with "" results as NOT NULL, because CHAR is blank-padded. However, a VARCHAR2 variable set with '' becomes NULL:

```sql
SET SERVEROUTPUT ON;
DROP TABLE tab1;
CREATE TABLE tab1 ( pk NUMBER, c1 CHAR(10), c2 VARCHAR2(10) );
```
DECLARE
    v1 CHAR(10);
    v2 VARCHAR2(10);
BEGIN
    v1 := '';
    v2 := '';
    IF v1 IS NULL THEN
        dbms_output.put_line('v1 is null');
    ELSE
        dbms_output.put_line('v1 = ' || v1);
    END IF;
    IF v2 IS NULL THEN
        dbms_output.put_line('v2 is null');
    ELSE
        dbms_output.put_line('v2 = ' || v2);
    END IF;
    INSERT INTO tab1 VALUES ( 101, '', '' );
    INSERT INTO tab1 VALUES ( 102, v1, v2 );
END;
/
SELECT pk, NVL(c1,'NULL'), NVL(c2,'NULL') FROM tab1 ORDER BY pk;
QUIT

Output:

<table>
<thead>
<tr>
<th>PK</th>
<th>NVL(C1,'NULL')</th>
<th>NVL(C2,'NULL')</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>102</td>
<td>v1</td>
<td>v2</td>
</tr>
</tbody>
</table>

Solution

To increase portability, it is recommended that you avoid the usage of literal string values with a length of zero in SQL statements. Instead, use the NULL constant, or program variables.

String delimiters

**Informix®**

The ANSI SQL string delimiter character is the single quote (`'string'`), while double quotes are used to delimit database object names:

```sql
SELECT ... WHERE "tabname"."colname" = 'a string value'
```

In Informix® databases created in native mode (non-ANSI), you can use double quotes as string delimiters:

```sql
SELECT ... WHERE tabname.colname = 'a string value'
```

This is important, since many BDL programs use that character to delimit the strings in SQL commands.

**Note:** This problem concerns only double quotes within SQL statements. Double quotes used in pure BDL string expressions are not subject to SQL compatibility problems.
ORACLE

ORACLE follows the ANSI SQL specification, using single quotes for string delimiters and double quotes for database object names.

Solution

When using Static SQL statements, the fglcomp compiler converts string literals using double quotes to string literals with single quotes:

```bash
$ cat s.4gl
MAIN
  DEFINE n INT
  SELECT COUNT(*) INTO n FROM tab1 WHERE coll = "abc"
END MAIN
$ fglcomp -S s.4gl
s.4gl^3^SELECT COUNT(*) FROM tab1 WHERE coll = 'abc'
```

However, SQL statements created dynamically are not modified by the Genero compiler.

The Genero database interface can automatically replace all double quotes by single quotes in SQL statements. This applies to static and dynamic SQL statements.

The translation of double quoted expression to single quoted expressions can be controlled with the following FGLPROFILE entry:

```
dbi.database.dbname.ifxemul.dblquotes = true
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

However, database object names must not be delimited by double quotes, because the database interface cannot determine the difference between a database object name and a quoted string! For example, if the program executes the SQL statement:

```sql
... WHERE "tabname"."colname" = "a string value"
```

replacing all double quotes by single quotes would produce:

```sql
... WHERE 'tabname'.'colname' = 'a string value'
```

This would produce an error since 'tabname'.colname' is not allowed by ORACLE.

Escaped string delimiters can be used inside strings like the following:

```
'This is a single quote: ''
'This is a single quote: \\''
"This is a double quote: """
"This is a double quote: \\"
```

Although double quotes are replaced automatically in SQL statements, it is recommended that you use only single quotes to enforce portability.

Related concepts

String literals in SQL statements on page 634
Single quotes is the standard for delimiting string literals in SQL.

**Single row SELECT**

**Informix®**

With Informix®, you must use the system table with a condition on the table id:

```
SELECT user FROM systables WHERE tabid=1
```

**ORACLE**

Oracle provides the DUAL table to generate one row only:

```
SELECT user FROM DUAL
```

**Solution**

Check the BDL sources for "FROM systables WHERE tabid=1" and use dynamic SQL to resolve this problem.

Consider writing a FUNCTION which produces the FROM and WHERE part, depending on the target database type.

**MATCHES and LIKE**

**Informix®**

Informix® supports MATCHES and LIKE operators in SQL statements. 

MATCHES expects * and ? wild-card characters, while LIKE uses the % and _ wild-cards as equivalents.

```
( col MATCHES 'Smi*' AND col NOT MATCHES 'R?x' )
( col LIKE 'Smi%' AND col NOT LIKE 'R_x' )
```

MATCHES accepts also brackets notation, to specify a set of matching characters at a given position:

```
( col MATCHES '[Pp]aris' )
( col MATCHES '[0-9][a-z]*' )
```

**ORACLE**

Oracle® does not provide an equivalent of the Informix® MATCHES operator.

The LIKE operator is supported.

**Important:** With Oracle®, columns defined as CHAR (N) are blank padded, and trailing blanks are significant in LIKE expressions. As a result, with a CHAR (5) value such as 'abc ' (with 2 trailing blanks), the expression (colname LIKE 'ab_') will not match. To workaround this behavior, you can use (RTRIM(colname) LIKE 'pattern'). However, consider adding the condition AND (colname LIKE 'pattern%') to force the DB server to optimize the query of the column as indexed. The CONSTRUCT instruction uses this technique when the entered criteria does not end with a * star wildcard.

**Solution**

The database driver is able to translate Informix® MATCHES expressions to LIKE expressions, when no [ ] bracket character ranges are used in the MATCHES operand.

The MATCHES to LIKE expression translation is controlled by the following FGLPROFILE entry:

```
dbi.database.dbname.ifxemul.matches = 1 true 1 false 1
```
**Important:** Only [NOT] MATCHES followed by a search pattern provided as a string literal can be converted by ODI drivers. A [NOT] MATCHES followed by a ? question mark parameter place holder is not translated!

For maximum portability, consider replacing the MATCHES expressions with LIKE expressions in all SQL statements.

Avoid using CHAR (N) types for variable length character data (such as name, address).

**Related concepts**

MATCHES and LIKE operators on page 645
Use the standard LIKE operator instead of the MATCHES operator.

Query by example (CONSTRUCT) on page 1565
The CONSTRUCT instruction implements database query criteria input in an application form.

**SELECT * (asterisk)**

**Informix**
Informix® allows you to use the star character in the select list along with other expressions:

```
SELECT col1, * FROM tab1 ...
```

**ORACLE**
Oracle® does not support the asterisk notation after another expression in the SELECT list.
Use the table name as a prefix to the star:

```
SELECT col1, tab1.* FROM tab1 ...
```

**Solution**
Always use the table name before the star.

**Row limiting clause**

**Informix**
Informix® SQL supports the SKIP and FIRST/LIMIT keywords to limit the number of rows of a result set:

```
SELECT SKIP 10 FIRST 20 customer.* FROM customer ... ORDER BY cust_name
```

This Informix SQL syntax is not portable.

Recent database engines support the row limiting clause syntax defined by the SQL standard:

```
SELECT ... OFFSET n ROWS FETCH FIRST m ROWS ONLY
```

This should be the preferred syntax to be used, if all target database types support this SELECT clause.

The ODI database drivers can convert the Informix SQL SKIP/FIRST row limiting clause to a native SQL equivalent, if the row limiting clause parameters are simple integer literals (the clause is not translated when using SQL parameters / program variables).

**Important:** In addition to the SKIP/FIRST clause of the projection clause, Informix SQL supports also a LIMIT clause after the ORDER BY clause:

```
SELECT customer.* FROM customer ... ORDER BY cust_name LIMIT 10
```
This Informix SQL syntax construction is not converted by the ODI drivers. To benefit from the conversion, review the code to use the Informix SQL SKIP/FIRST clause instead.

**ORACLE**

Oracle® supports the following row limiting clause:

```
SELECT ... ORDER BY ... OFFSET n ROWS FETCH FIRST m ROWS ONLY
```

**Solution**

The Informix SQL row limiting clause can be converted by the Oracle® driver to the native SQL equivalent clause, when the parameters are simple integer literals.

**Note**: The row limiting clause must not use SQL parameters. Only row limiting clauses using integer constants will be converted.

The translation of the Informix SQL row limiting clause can be controlled with the following FGLPROFILE entry:

```
dbi.database.dsnname.ifxemul.rowlimiting = true false
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

Row limiting clause (SELECT) on page 650

How to use the right clause to limit the number of rows produced by a SELECT statement?

**BDL programming**

Oracle Database related programming topics.

**SQL errors on PREPARE**

**Informix**

With Informix®, a PREPARE instruction returns an SQL error in case of problem:

```
TRY
  PREPARE stmt FROM "SELECT * FROM WHERE pk=1" -- table is missing!
CATCH
  DISPLAY "SQL ERROR:", SQLCA.SQLCODE
END TRY
```

**ORACLE**

The Oracle® interface is implemented with the Oracle® Call Interface (OCI). This library does not provide a way to send SQL statements to the database server during the BDL PREPARE instruction, as in the Informix® interface.

When preparing an SQL statement with the BDL PREPARE or DECLARE instruction, no SQL error will be returned if the SQL statement is invalid. However, an SQL error will occur after the OPEN/FOREACH/EXECUTE instructions.

**Solution**

Make sure your BDL programs do not test the STATUS or SQLCA.SQLCODE variable just after PREPARE instructions.

Change the program logic in order to handle the SQL errors when opening the cursors (OPEN) or when executing SQL statements (EXECUTE).

**Related concepts**

The SQLCA diagnostic record on page 594
The SQLCA variable is a predefined record containing SQL statement execution information.

**INSERT cursors**

**Informix**

Informix provides *insert cursors* to optimize row creation in a database. An insert cursor is declared as a cursor, and rows as added with the **PUT** instruction. The rows are buffered and sent to the database server when executing a **FLUSH** instruction, or when the cursor is closed with **CLOSE**. When using transactions in Informix®, the **OPEN**, **PUT** and **FLUSH** instructions must be executed within a transaction block.

```sql
DECLARE c1 CURSOR FOR INSERT INTO tab1 ... 
BEGIN WORK 
OPEN c1  
WHILE ... 
  PUT c1 USING var-list 
END WHILE 
CLOSE c1  
COMMIT WORK 
```

**ORACLE**

ORACLE does not support insert cursors.

**Solution**

Insert cursors are emulated by the database interface, using basic **INSERT** SQL instructions. The performances might be not as good as with Informix®, but the feature is fully supported.

**Related concepts**

- Insert cursors on page 633
- Using insert cursors with non-Informix databases.

**Cursors WITH HOLD**

**Informix**

Informix® closes opened cursors automatically when a transaction ends, unless the **WITH HOLD** option is used in the **DECLARE** instruction:

```sql
DECLARE c1 CURSOR WITH HOLD FOR SELECT ... 
OPEN c1  
BEGIN WORK 
FETCH c1 ... 
COMMIT WORK 
FETCH c1 ... 
CLOSE c1 
```

**ORACLE**

With ORACLE, all cursors are **WITH HOLD** cursors by default, unless the **FOR UPDATE** clause is used in the **SELECT** statement.

If the **SELECT** statement uses the **FOR UPDATE** clause, ORACLE will close the cursor automatically when a transaction ends.
Solution

BDL cursors declared WITH HOLD remain open even after terminating a transaction with a COMMIT WORK or ROLLBACK WORK.

For consistency with other database brands, database cursors that are not declared WITH HOLD are automatically closed, when a COMMIT WORK or ROLLBACK WORK is performed.

**Important:** Opening a WITH HOLD cursor declared with a SELECT FOR UPDATE results in an SQL error; in the same conditions, this does not normally appear with Informix®. Review the program logic in order to find another way to set locks.

**Related concepts**

Cursors WITH HOLD on page 631
Programming WITH HOLD cursors using SELECT with and without FOR UPDATE clause.

**SELECT ... FOR UPDATE**

**Informix®**

Legacy BDL programs typically use a cursor with SELECT FOR UPDATE to implement pessimistic locking and avoid several users editing the same rows:

```sql
DECLARE cc CURSOR FOR
SELECT ... FROM tab WHERE ...
FOR UPDATE
OPEN cc
FETCH cc <-- lock is acquired
...
CLOSE cc <-- lock is released
```

The row must be fetched in order to set the lock.

If the cursor is local to a transaction, the lock is released when the transaction ends. If the cursor is declared WITH HOLD, the lock is released when the cursor is closed.

Informix® provides the SET LOCK MODE instruction to define the lock wait timeout:

```sql
SET LOCK MODE TO { WAIT | NOT WAIT | WAIT seconds }
```

The default mode is NOT WAIT.

**ORACLE**

ORACLE allows individual and exclusive row locking with:

```sql
SELECT ... FOR UPDATE [OF col-list]
```

A lock is acquired for each selected row when the cursor is opened, before the first fetch.

Cursors using SELECT ... FOR UPDATE are automatically closed when the transaction ends.

Locks are not released when a cursor is closed.

ORACLE's locking granularity is at the row level.

The NOWAIT keyword can be used in SELECT ... FOR UPDATE statement, the return immediately if the row is already locked by another user:

```sql
SELECT ... FOR UPDATE [OF col-list] NOWAIT
```
Solution

Important: Cursors declared with SELECT ... FOR UPDATE using the WITH HOLD clause cannot be supported with ORACLE.

The database interface is based on an emulation of an Informix® engine using transaction logging.

Opening a SELECT ... FOR UPDATE cursor declared outside a transaction will raise an SQL error -255 (not in transaction).

When using pessimistic locking with DECLARE ... CURSOR FOR SELECT ... FOR UPDATE, review the program logic to have OPEN and CLOSE instructions inside transactions (BEGIN WORK + COMMIT WORK / ROLLBACK WORK).

See also Cursors with Hold and UPDATE/DELETE WHERE CURRENT OF for more details.

Related concepts
Cursors WITH HOLD on page 631
Programming WITH HOLD cursors using SELECT with and without FOR UPDATE clause.

UPDATE/DELETE ... WHERE CURRENT OF

Informix®

Informix® allows positioned UPDATEs and DELETEs with the "WHERE CURRENT OF cursor" clause, if the cursor has been DECLARED with a SELECT ... FOR UPDATE statement.

ORACLE

The UPDATE/DELETE ... WHERE CURRENT OF cursor statements are not support by the Oracle database API.

However, ROWIDs can be used for positioned updates and deletes.

Solution

UPDATE/DELETE ... WHERE CURRENT OF instructions are emulated by the ORACLE database interface by using ROWIDs:

The ORACLE database interface replaces WHERE CURRENT OF cursor by WHERE ROWID=:rid and sets the value of the ROWID returned by the last FETCH done with the given cursor.

As a replacement of WHERE CURRENT OF, if the database table is defined with a primary key column, use the value fetched from the SELECT [FOR UPDATE] cursor in the WHERE clause of the UPDATE/DELETE statement.

Related concepts
Positioned UPDATE/DELETE on page 631
Using positioned updates/deletes with named database cursors.

LOAD and UNLOAD

Informix®

Informix® provides two SQL instructions to export / import data from / into a database table:

The UNLOAD instruction copies rows from a database table into a text file:

UNLOAD TO "filename.unl" SELECT * FROM tab1 WHERE ..

The LOAD instructions insert rows from a text file into a database table:

LOAD FROM "filename.unl" INSERT INTO tab1
ORACLE

ORACLE provides tools like SQL*Plus and SQL*Loader to load/unload data from a database.

Solution

LOAD and UNLOAD instruction are implemented in the Genero BDL runtime system with basic INSERT (for LOAD) or SELECT (for UNLOAD) SQL commands. The LOAD and UNLOAD instruction can be supported with various database servers.

However, LOAD and UNLOAD require the description of the column types in order to work, that can lead to some differences in the data formatting.

Note: If no transaction is started, the LOAD instruction will automatically execute a BEGIN WORK and COMMIT WORK when finished, or ROLLBACK WORK if a row insertion failed while loading. Terminating a transaction will automatically close cursors not defined WITH HOLD option. To workaround this situation, see more details in the LOAD on page 736 reference topic.

The LOAD and UNLOAD BDL instructions are supported with ORACLE with some limitations:

• There is a difference when using ORACLE DATE columns. DATE columns created in the ORACLE database are equivalent to Informix® DATETIME YEAR TO SECOND columns. In LOAD and UNLOAD, all ORACLE DATE columns are treated as Informix® DATETIME YEAR TO SECOND columns and thus will be unloaded with the "YYYY-MM-DD hh:mm:ss" format.

• Informix® INTEGER and SMALLINT are mapped to ORACLE NUMBER(?) columns. Those values will be unloaded as Informix® DECIMAL(10) and DECIMAL(5) values, that is, with a trailing dot-zero ".0".

• When using an Informix® database, simple dates are unloaded using the DBDATE format (ex:"23/12/1998"). Unloading from an Informix® database for loading into an ORACLE database is not supported.

Related concepts
LOAD and UNLOAD instructions on page 639
The LOAD and UNLOAD instructions can produce different data formats depending on the database server type.

SQL Interruption

Informix®

With Informix®, it is possible to interrupt a long running query if the SQL INTERRUPT ON option.

ORACLE

Oracle supports SQL Interruption: The db client must issue an OCIBreak() OCI call to interrupt a query.

Solution

The ORACLE database driver supports SQL interruption and converts the native SQL error code -1013 to the Informix® error code -213.

Related concepts
Using SQL interruption on page 601
Interrupt long running SQL queries, or interrupt queries waiting for locked data.

Scrollable cursors

Informix®

Informix® SQL and Genero BDL support scrollable cursors when you specify the SCROLL clause in the DECLARE cursor instruction:

DECLARE c1 SCROLL CURSOR FOR SELECT ...
Important: Informix does not allow to fetch TEXT/BYTE columns with scrollable cursors. If you declare a scroll cursor with a SELECT containing TEXT/BYTE columns, Informix will produce the SQL error -611 when executing the OPEN instruction.

ORACLE

Oracle OCI supports native scrollable cursors with the OCI_STMT_SCROLLABLE_READONLY statement attribute.

Solution

The Oracle database driver uses native scrollable cursors by setting the OCI_STMT_SCROLLABLE_READONLY statement attribute.

With Oracle is it possible to use LOB columns in a scrollable cursor. However, you should consider to use only simple data types for scrollable cursors, and fetch TEXT/BYTE data in a secondary SELECT statement using the primary key of the current row.

Related concepts

Scrollable cursors on page 623
How scrollable cursors can be supported on different databases.

Stored procedure calls

Oracle supports stored procedures and stored functions as a group of PL/SQL statements that you can call by name.

To create a stored procedure in an Oracle database, use the CREATE PROCEDURE statement. See reference documentation for more details.

Stored procedures with output parameters

Oracle stored procedures or stored functions must be called with the input and output parameters specification in the USING clause of the EXECUTE, OPEN or FOREACH instruction. As in normal dynamic SQL, parameters must correspond by position, and the IN/OUT/INOUT options must match the parameter definition of the stored procedure.

To execute the stored procedure, you must include the procedure in an anonymous PL/SQL block with BEGIN and END keywords:

```sql
PREPARE stmt FROM "begin procl(?,?,?); end;"
```

Remark: Oracle stored procedures do not specify the size of number and character parameters. The size of output values (especially character strings) are defined by the calling context (i.e. the data type of the variable used when calling the procedure). When you pass a CHAR(10) to the procedure, the returning value will be filled with blanks to reach a size of 10 bytes.

Note that for technical reasons, the Oracle driver uses dynamic binding with OCIBindDynamic(). The Oracle Call Interface does not support stored procedures parameters with the CHAR data type when using dynamic binding. You must use VARCHAR2 instead of CHAR to define character string parameters for stored procedures.

Here is a complete example creating and calling a stored procedure with output parameters:

```sql
MAIN
   DEFINE n INTEGER
   DEFINE d DECIMAL(6,2)
   DEFINE c VARCHAR(200)
   DATABASE test1
   EXECUTE IMMEDIATE
     "create procedure procl("
     | "   p1 in int,"
     | "   p2 in out number,"
     | "   p3 in out varchar2"
     | " )"
     | " is begin"
```
```
"  p2:= p1 + 0.23;"
"  p3:= 'Value = ' || to_char(p1);"
"end;"
PREPARE stmt FROM "begin procl(?,?,?); end;"
LET n = 111
EXECUTE stmt USING n IN, d INOUT, c INOUT
DISPLAY d
DISPLAY c
END MAIN
```

### Stored functions with a return value

To execute the stored function returning a value, you must include the function in an anonymous PL/SQL block with `BEGIN` and `END` keywords, and use an assignment expression to specify the place holder for the returning value:

```
PREPARE stmt FROM "begin ?:= func1(?,?,?); end;"
```

### Stored procedures with result set

Oracle supports result set generation from stored procedures with the concept of cursor variables (`REF CURSOR`). Note that Genero does not support cursor references produced by Oracle stored procedures or functions.

### PostgreSQL

#### Supported versions

Genero BDL supports the following PostgreSQL versions:

- PostgreSQL 9.4 to 9.6
- PostgreSQL 10
- PostgreSQL 11
- PostgreSQL 12

#### Installation (Runtime Configuration)

PostgreSQL related installation topics.

**Install PostgreSQL and create a database - database configuration/design tasks**

If you are tasked with installing and configuring the database, here is a list of steps to be taken:

1. Compile and install the PostgreSQL Server on your computer. PostgreSQL is a free database, you can download the sources from [www.postgresql.org](http://www.postgresql.org).
2. Read PostgreSQL installation notes for details about the "data" directory creation with the `initdb` utility:

   ```
   $ initdb -D $PGDIR/data
   ```

3. Set configuration parameters in `postgresql.conf`:
   a) When using a PostgreSQL version prior to version 12: If ROWIDs are used in the applications, these are converted to PostgreSQL OIDs. However, this feature is disabled by default. Starting with PostgreSQL version 12, OIDs are no longer supported.

   In order to get OIDs in PostgreSQL versions prior to version 12, set the `default_with_oid` server parameter to `on`.

4. Start the PostgreSQL process to listen to database client connections:

   ```
   $ pg_ctl -D $PGDIR/data -l $PGDIR/logfile start -o "-p 5435"
   ```

**Important:** If you want to connect through TCP (for example from a Windows™ PostgreSQL client), you must start PostgreSQL with the `-i` option and setup the "pg_hba.conf" file for security (trusted hosts and users).
5. Create a PostgreSQL database with the `createdb` utility, by specifying the character set of the database:

```
$ createdb -h hostname --port 5435 dbname --encoding encoding --locale locale
```

6. If you plan to use SERIAL emulation, you need the `plpgsql` procedure language, because the database interface uses this language to create serial triggers.

Execute the following SQL to check that the `plpgsql` language is available in your PostgreSQL server:

```
dbname=> SELECT lanowner, lanname FROM pg_language WHERE lanname = 'plpgsql';
<table>
<thead>
<tr>
<th>lanowner</th>
<th>lanname</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>plpgsql</td>
</tr>
</tbody>
</table>
(1 row)
```

7. Connect to the database as the administrator user and create a database user dedicated to your application, the application administrator:

```
dbname=> CREATE USER appadmin PASSWORD 'password';
CREATE USER
dbname=> GRANT ALL PRIVILEGES ON DATABASE dbname TO appadmin;
GRANT
dbname=> \
```

8. Create the application tables.

   Convert Informix® data types to PostgreSQL data types. See Data type conversion table: Informix to PostgreSQL on page 971 for more details.

9. If you plan to use the SERIAL emulation, you must prepare the database.

   See SERIAL and BIGSERIAL data types on page 980 for more details.

### Prepare the runtime environment - connecting to the database

1. In order to connect to PostgreSQL, you must have a PostgreSQL database driver "dbmpgs" in `$FGLDIR/dbdrivers`.

   On HP/UX LP64, the PostgreSQL database driver must be linked with the libxnet library if you want to use networking.

2. The PostgreSQL client software is required to connect to a database server.

   Check whether the PostgreSQL client library (`libpq.*`) is installed on the machine where the BDL programs run.

3. Make sure that the PostgreSQL client environment variables are properly set.

   Check, for example, `PGDIR` (the path to the installation directory), `PGDATA` (the path to the data files directory), etc. See the PostgreSQL documentation for more details.

4. Check the database client locale settings (for example, set the `PGCLIENTENCODING` environment variable).

   The database client locale must match the locale used by the runtime system (`LC_ALL`, `LANG`).

5. Verify the environment variable defining the search path for the PostgreSQL database client shared libraries (`libpq.so` on UNIX™, `LIBPQ.DLL` on Windows™).

### Table 261: Shared library environment setting for PostgreSQL

<table>
<thead>
<tr>
<th>PostgreSQL version</th>
<th>Shared library environment setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL 9.2 and higher</td>
<td><code>UNIX™</code>]: Add <code>$PGDIR/lib</code> to <code>LD_LIBRARY_PATH</code> (or its equivalent).</td>
</tr>
<tr>
<td></td>
<td><code>Windows™</code>: Add <code>%PGDIR%\bin</code> to <code>PATH</code>.</td>
</tr>
</tbody>
</table>
6. To verify if the PostgreSQL client environment is correct, you can start the PostgreSQL command interpreter:

```
$ psql dbname -U appadmin -W
```

7. Set up the FGLPROFILE entries for database connections.
   a) Define the PostgreSQL database driver:

   ```
   dbi.database.dbname.driver = "dbmpgs"
   ```

   b) The 'source' parameter defines the name of the PostgreSQL database, as well as additional connection parameters if needed, such as the server host name, the TCP port and specific PostgresSQL connection options.

   ```
   dbi.database.dbname.source = "test1"
   ```

   The source parameter must have the following form:

   ```
   dbname[@host[:port]]?options
   ```

   where:

   - `dbname` defines the name of the PostgreSQL database
   - `host` defines the server host name, or IP address (IPv6 host address needs to be enclosed it in square brackets)
   - `port` defines the TCP port
   - `options` is a URI-style query string defining PostgreSQL connection parameters

   For example:

   ```
   mydb@orion:5433?connect_timeout=10&application_name=myapp
   ```

   c) Define pre-fetch rows.

   To improve performances, you can define the number of result set rows that the driver must prefetch:

   ```
   dbi.database.dbname.pgs.prefetch.rows = integer
   ```

   This will be applied to all application cursors.

   The default is 50 rows. Do not change the default, except if it gives really better performances: This can blow up memory usage for each DB client process.

   For more details, see PostgreSQL specific FGLPROFILE parameters on page 676.

Database concepts
PostgreSQL related database concepts topics.

Database concepts
Like Informix® servers, PostgreSQL can handle multiple database entities. Tables created by a user can be accessed without the owner prefix by other users as long as they have access privileges to these tables.

Tip: If you have several Informix database entities, migrating from the Informix database to another database it is a good opportunity to centralize all tables in a single database. To avoid conflicts with table names, use a prefix when needed.

Solution
Create a PostgreSQL database for each Informix® database.
Data storage concepts

When converting from Informix® to PostgreSQL, the aim is to try to preserve as much of the data storage information as possible in the process. The most important storage decisions made for Informix® database objects (like initial sizes and physical placement) can be applied to the PostgreSQL database.

Storage concepts are quite similar in Informix® and in PostgreSQL, but the names are different.

Concurrency management

Data consistency and concurrency concepts

- **Data Consistency** applies to situations when readers want to access data currently being modified by writers.
- **Concurrent Data Access** applies to situations when several writers are accessing the same data for modification.
- **Locking Granularity** defines the amount of data concerned when a lock is set (for example, row, page, table).

Informix®

Informix® uses a locking mechanism to handle data consistency and concurrency. When a process changes database information with UPDATE, INSERT or DELETE, an exclusive lock is set on the touched rows. The lock remains active until the end of the transaction. Statements performed outside a transaction are treated as a transaction containing a single operation and therefore release the locks immediately after execution. SELECT statements can set shared locks, depending on isolation level. In case of locking conflicts (for example, when two processes want to acquire an exclusive lock on the same row for modification, or when a writer is trying to modify data protected by a shared lock), the behavior of a process can be changed by setting the lock wait mode.

Control:

- Lock wait mode: SET LOCK MODE TO ...
- Isolation level: SET ISOLATION TO ...
- Locking granularity: CREATE TABLE ... LOCK MODE {PAGE|ROW}
- Explicit exclusive lock: SELECT ... FOR UPDATE

Defaults:

- The default isolation level is READ COMMITTED.
- The default lock wait mode is NOT WAIT.
- The default locking granularity is PAGE.

PostgreSQL

When data is modified, exclusive locks are set and held until the end of the transaction. For data consistency, PostgreSQL uses a multi-version consistency model: A copy of the original row is kept for readers before performing writer modifications. Readers do not have to wait for writers as in Informix®. The simplest way to think of the PostgreSQL implementation of read consistency is to imagine each user operating a private copy of the database, hence the multi-version consistency model. Since PostgreSQL 9.4, the lock wait mode for the current SQL session can be changed by updating the 'lock_timeout' parameter of the pg_settings system view. Locks are set at the row level in PostgreSQL and this cannot be changed.

Control:

- Lock wait mode: UPDATE pg_settings SET setting=ms WHERE name='lock_timeout'
- Isolation level: SET TRANSACTION ISOLATION LEVEL ...
- Explicit exclusive lock: SELECT ... FOR UPDATE

Defaults:

- The default isolation level is Read Committed.

The main difference between Informix® and PostgreSQL is that readers do not have to wait for writers in PostgreSQL.
**Solution**

The `SET ISOLATION TO ...` Informix® syntax is replaced by `SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL ...` in PostgreSQL. The next table shows the isolation level mappings done by the PostgreSQL database driver:

**Table 262: Isolation level mappings done by the PostgreSQL database driver**

<table>
<thead>
<tr>
<th>SET ISOLATION instruction in program</th>
<th>Native SQL command</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SET ISOLATION TO DIRTY READ</code></td>
<td><code>SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL READ COMMITTED</code></td>
</tr>
<tr>
<td><code>SET ISOLATION TO COMMITTED READ [READ COMMITTED] [RETAIN UPDATE LOCKS]</code></td>
<td><code>SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL READ COMMITTED</code></td>
</tr>
<tr>
<td><code>SET ISOLATION TO CURSOR STABILITY</code></td>
<td><code>SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL READ COMMITTED</code></td>
</tr>
<tr>
<td><code>SET ISOLATION TO REPEATABLE READ</code></td>
<td><code>SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL SERIALIZABLE</code></td>
</tr>
</tbody>
</table>

For portability, it is recommended that you work with Informix® in the read committed isolation level, make processes wait for each other (lock mode wait), and create tables with the "lock mode row" option.

When using `SET LOCK MODE ...` in the programs, it will be converted to an `UPDATE pg_setting` instruction for PostgreSQL:

**Table 263: SET LOCK MODE as handled by the PostgreSQL database driver**

<table>
<thead>
<tr>
<th>SET LOCK MODE instruction in program</th>
<th>Native SQL command</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SET LOCK MODE TO WAIT</code></td>
<td><code>UPDATE pg_settings SET setting=0 WHERE name='lock_timeout'</code></td>
</tr>
<tr>
<td><code>SET LOCK MODE TO WAIT seconds</code></td>
<td><code>UPDATE pg_settings SET setting= (seconds*1000) WHERE name='lock_timeout'</code></td>
</tr>
<tr>
<td><code>SET LOCK MODE TO NOT WAIT</code></td>
<td><code>UPDATE pg_settings SET setting=1 WHERE name='lock_timeout'</code></td>
</tr>
</tbody>
</table>

See the Informix® and PostgreSQL documentation for more details about data consistency, concurrency and locking mechanisms.

**Related concepts**

**Concurrent data access** on page 599
Understanding concurrent data access and data consistency.

**Optimistic locking** on page 624
Implementing optimistic locking to handle access concurrently to the same database records.

**Cursors WITH HOLD** on page 631
Programming WITH HOLD cursors using SELECT with and without FOR UPDATE clause.

**Transactions handling**

**Informix®**

With the Informix® native mode (non ANSI):

- Transactions blocks start with BEGIN WORK and terminate with COMMIT WORK or ROLLBACK WORK.
- Statements executed outside a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.

```
UPDATE tab1 SET ...   -- auto-committed
BEGIN WORK            -- start of TX block
UPDATE tab1 SET ...
UPDATE tab2 SET ...
...                  -- start of TX block
COMMIT WORK           -- end of TX block
```

Informix® version 11.50 introduces savepoints:

```
SAVEPOINT name [UNIQUE]
ROLLBACK [WORK] TO SAVEPOINT _name_ 1
RELEASE SAVEPOINT name
```

**PostgreSQL**

PostgreSQL supports transaction with savepoints:

- Transactions are started with BEGIN WORK.
- Transactions are validated with COMMIT WORK.
- Transactions are canceled with ROLLBACK WORK.
- Savepoints can be placed with SAVEPOINT name.
- Transactions can be rolled back to a savepoint with ROLLBACK TO SAVEPOINT name.
- Savepoints can be released with RELEASE SAVEPOINT name.
- Statements executed outside of a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.
- If an SQL error occurs in a transaction, the whole transaction is aborted.

Transactions in stored procedures: avoid using transactions in stored procedures to allow the client applications to handle transactions, depending on the transaction model.

The main difference between Informix® and PostgreSQL resides in the fact that PostgreSQL cancels the entire transaction if an SQL error occurs in one of the statements executed inside the transaction. The following code example illustrates this difference:

```
CREATE TABLE tab1 ( k INT PRIMARY KEY, c CHAR(10) )
WHENEVER ERROR CONTINUE
BEGIN WORK
INSERT INTO tab1 ( 1, 'abc' )
INSERT INTO tab1 ( 1, 'abc' )
-- PK constraint violation = SQL Error, whole TX is aborted
COMMIT WORK
```

With Informix® and PostgreSQL, this code will leave the table with one row inside, since the first INSERT statement succeeded. With PostgreSQL, the table will remain empty after executing this piece of code, because the server will rollback the whole transaction. To workaround this problem in PostgreSQL you can use SAVEPOINT as described in Solution on page 970.
Solution
Informix® transaction handling commands are automatically converted to PostgreSQL instructions to start, validate or cancel transactions.

Regarding the transaction control instructions, the BDL applications do not have to be modified in order to work with PostgreSQL.

You must review the SQL statements inside BEGIN WORK / COMMIT WORK instruction and check if these can raise an SQL error. The SQL statements that can potentially raise an SQL error must be protected with a SAVEPOINT. If an error occurs, just rollback to the savepoint:

```sql
MAIN
    DATABASE test1
    CREATE TABLE tab1 ( k INT PRIMARY KEY, c CHAR(10) )
    WHENEVER ERROR CONTINUE
    BEGIN WORK
    INSERT INTO tab1 VALUES ( 1, 'abc' )
    CALL sql_protect()
    INSERT INTO tab1 VALUES ( 1, 'abc' )
    -- PK constraint violation = SQL Error
    CALL sql_unprotect()
    COMMIT WORK
END MAIN

FUNCTION sql_protect()
    IF fgl_db_driver_type() !="pgs" THEN
        RETURN
    END IF
    SAVEPOINT _sql_protect_
END FUNCTION

FUNCTION sql_unprotect()
    IF fgl_db_driver_type() !="pgs" THEN
        RETURN
    END IF
    IF SQLCA.SQLCODE < 0 THEN
        ROLLBACK WORK TO SAVEPOINT _sql_protect_
    ELSE
        RELEASE SAVEPOINT _sql_protect_
    END IF
END FUNCTION
```

Note: If you want to use savepoints, do not use the UNIQUE keyword in the savepoint declaration, always specify the savepoint name in ROLLBACK TO SAVEPOINT, and do not drop savepoints with RELEASE SAVEPOINT.

Related concepts
Database transactions on page 600
Database transactions define a set of SQL instructions to be executed as a whole, or rolled back as a whole.

Database users
Informix®

Until version 11.70.xC2, Informix® database users must be created at the operating system level and must be members of the 'informix' group.

Starting with 11.70.xC2, Informix® supports database-only users with the CREATE USER instruction, as in most other db servers.

Any database user must have sufficient privileges to connect and use resources of the database; user rights are defined with the GRANT command.
**PostgreSQL**

PostgreSQL users must be registered in the database with the `CREATE USER` command:

```sql
dbname=# CREATE USER user-name PASSWORD 'password';
CREATE USER
dbname=# GRANT ALL PRIVILEGES ON DATABASE dbname TO user-name;
GRANT
```

**Solution**

Based on the application logic (if it is a multiuser application), you have to create one or several PostgreSQL users.

**Related concepts**

[Database users and security](#) on page 608

Properly identifying database users allows to use database security and audit features.

**Setting privileges**

**Informix®**

Informix® users must have at least the `CONNECT` privilege to access the database:

```sql
GRANT CONNECT TO username
```

Application administration users need the `RESOURCE` privilege to create tables:

```sql
GRANT RESOURCE TO username
```

Since version 7.20, Informix® supports database roles:

```sql
GRANT rolename TO username
```

**PostgreSQL**

PostgreSQL supports the concept of *roles* to grant or revoke permissions to a group of users.

See PostgreSQL documentation for more details.

**Solution**

Informix® and PostgreSQL user privileges management are quite similar.

**Data dictionary**

PostgreSQL related data dictionary topics.

**Data type conversion table: Informix to PostgreSQL**

**Table 264: Data type conversion table (Informix to PostgreSQL)**

<table>
<thead>
<tr>
<th>Informix® data types</th>
<th>PostgreSQL data types (since 8.4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR (n)</td>
<td>CHAR (n)</td>
</tr>
<tr>
<td>VARCHAR (n[,m])</td>
<td>VARCHAR (n)</td>
</tr>
<tr>
<td>LVARCHAR (n[,m])</td>
<td>VARCHAR (n)</td>
</tr>
<tr>
<td>NCHAR (n)</td>
<td>N/A</td>
</tr>
<tr>
<td>NVARCHAR (n[,m])</td>
<td>N/A</td>
</tr>
<tr>
<td>Informix® data types</td>
<td>PostgreSQL data types (since 8.4)</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>---------------------------------------------------</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>INT2</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INT4</td>
</tr>
<tr>
<td>BIGINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>INT8</td>
<td>BIGINT (see note 1)</td>
</tr>
<tr>
<td>SERIAL[(start)]</td>
<td>INTEGER (see note 1)</td>
</tr>
<tr>
<td>BIGSERIAL[(start)]</td>
<td>BIGINT (see note 1)</td>
</tr>
<tr>
<td>SERIAL8[(start)]</td>
<td>BIGINT (see note 1)</td>
</tr>
<tr>
<td>DOUBLE PRECISION / FLOAT[(n)]</td>
<td>FLOAT4</td>
</tr>
<tr>
<td>REAL / SMALLFLOAT</td>
<td>FLOAT8</td>
</tr>
<tr>
<td>DECIMAL(p,s)</td>
<td>DECIMAL(p,s)</td>
</tr>
<tr>
<td>DECIMAL(p)</td>
<td>DECIMAL (no precision = floating point)</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL (no precision = floating point)</td>
</tr>
<tr>
<td>MONEY(p,s)</td>
<td>DECIMAL(p,s)</td>
</tr>
<tr>
<td>MONEY(p)</td>
<td>DECIMAL(p,2)</td>
</tr>
<tr>
<td>MONEY</td>
<td>DECIMAL(16,2)</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME HOUR TO MINUTE</td>
<td>TIME(0) WITHOUT TIME ZONE</td>
</tr>
<tr>
<td>DATETIME HOUR TO SECOND</td>
<td>TIME(0) WITHOUT TIME ZONE</td>
</tr>
<tr>
<td>DATETIME HOUR TO FRACTION(p)</td>
<td>TIME(p) WITHOUT TIME ZONE</td>
</tr>
<tr>
<td>DATETIME YEAR TO MONTH</td>
<td>TIMESTAMP(0) WITHOUT TIME ZONE</td>
</tr>
<tr>
<td>DATETIME YEAR TO DAY</td>
<td>TIMESTAMP(0) WITHOUT TIME ZONE</td>
</tr>
<tr>
<td>DATETIME YEAR TO HOUR</td>
<td>TIMESTAMP(0) WITHOUT TIME ZONE</td>
</tr>
<tr>
<td>DATETIME YEAR TO MINUTE</td>
<td>TIMESTAMP(0) WITHOUT TIME ZONE</td>
</tr>
<tr>
<td>DATETIME YEAR TO SECOND</td>
<td>TIMESTAMP(0) WITHOUT TIME ZONE</td>
</tr>
<tr>
<td>DATETIME YEAR TO FRACTION(p)</td>
<td>TIMESTAMP(p) WITHOUT TIME ZONE</td>
</tr>
<tr>
<td>INTERVAL YEAR[(p)] TO MONTH</td>
<td>INTERVAL YEAR TO MONTH</td>
</tr>
<tr>
<td>INTERVAL YEAR[(p)] TO YEAR</td>
<td>INTERVAL YEAR</td>
</tr>
<tr>
<td>INTERVAL MONTH[(p)] TO MONTH</td>
<td>INTERVAL MONTH</td>
</tr>
<tr>
<td>INTERVAL DAY[(p)] TO FRACTION(n)</td>
<td>INTERVAL DAY TO SECOND(n)</td>
</tr>
<tr>
<td>INTERVAL DAY[(p)] TO SECOND</td>
<td>INTERVAL DAY TO SECOND(0)</td>
</tr>
<tr>
<td>INTERVAL DAY[(p)] TO MINUTE</td>
<td>INTERVAL DAY TO MINUTE</td>
</tr>
<tr>
<td>INTERVAL DAY[(p)] TO HOUR</td>
<td>INTERVAL DAY TO HOUR</td>
</tr>
<tr>
<td>INTERVAL DAY[(p)] TO DAY</td>
<td>INTERVAL DAY</td>
</tr>
<tr>
<td>INTERVAL HOUR[(p)] TO FRACTION(n)</td>
<td>INTERVAL HOUR TO SECOND(n)</td>
</tr>
<tr>
<td>INTERVAL HOUR[(p)] TO SECOND</td>
<td>INTERVAL HOUR TO SECOND(0)</td>
</tr>
</tbody>
</table>
**Informix® data types** | **PostgreSQL data types (since 8.4)**
--- | ---
INTERVAL HOUR[(p)] TO MINUTE | INTERVAL HOUR TO MINUTE
INTERVAL HOUR[(p)] TO HOUR | INTERVAL HOUR
INTERVAL MINUTE[(p)] TO FRACTION(n) | INTERVAL MINUTE TO SECOND(n)
INTERVAL MINUTE[(p)] TO SECOND | INTERVAL MINUTE TO SECOND(0)
INTERVAL MINUTE[(p)] TO MINUTE | INTERVAL MINUTE
INTERVAL SECOND[(p)] TO FRACTION(n) | INTERVAL SECOND(n)
INTERVAL SECOND[(p)] TO SECOND | INTERVAL SECOND(0)
INTERVAL SECOND[(p)] TO MINUTE | INTERVAL SECOND(n)
TEXT | TEXT
BYTE | BYTEA

Notes:

1. For more details about serial emulation, see SERIAL and BIGSERIAL data types on page 980.

**BOOLEAN data type**

**Informix®**

Informix® supports the BOOLEAN data type, which can store 't' or 'f' values.

Genero BDL implements the BOOLEAN data type in a different way: A BOOLEAN variable stores integer values 1 or 0 (for TRUE or FALSE). This type is designed to hold the result of a boolean expression.

**PostgreSQL**

PostgreSQL supports the BOOLEAN data type and stores 't' or 'f' values for TRUE and FALSE representation.

**Note:** It is not possible to insert the integer values 1 or 0 into PostgreSQL BOOLEAN columns: Values must be true, false, 't', 'f', '1' or '0'.

**Solution**

The PostgreSQL database interface converts the BOOLEAN integer values to a CHAR(1) values '1' or '0'.

The BOOLEAN type translation can be controlled with the following FGLPROFILE entry:

```
  dbi.database.dsnname.ifxemul.datatype.boolean = { true \ false }
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

Using portable data types on page 614

Only a limited set of data types are really portable across several database engines.

**CHAR and VARCHAR data types**

**Informix®**

Informix® supports the following character data types:
Table 265: Informix® character data types

<table>
<thead>
<tr>
<th>Informix® data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR (n)</td>
<td>SBCS and MBCS character data (max is 32767 bytes)</td>
</tr>
<tr>
<td>VARCHAR (n, m)</td>
<td>SBCS and MBCS character data (max is 255 bytes)</td>
</tr>
<tr>
<td>NCHAR (n)</td>
<td>Same as CHAR, with specific collation order</td>
</tr>
<tr>
<td>NVARCHAR (n, m)</td>
<td>Same as VARCHAR, with specific collation order</td>
</tr>
<tr>
<td>LVARCHAR (n)</td>
<td>Max size varies depending on the IDS version</td>
</tr>
</tbody>
</table>

With Informix®, both CHAR/VARCHAR and NCHAR/NVARCHAR data types can be used to store single-byte or multibyte encoded character strings. The only difference between CHAR/VARCHAR and NCHAR/NVARCHAR is in how they use sorting: N [VAR]CHAR types use the collation order, while [VAR]CHAR types use the byte order.

The character set used to store strings in CHAR/VARCHAR/NCHAR/NVARCHAR columns is defined by the DB_LOCALE environment variable.

The character set used by applications is defined by the CLIENT_LOCALE environment variable.

Informix® uses Byte Length Semantics (the size N that you specify in [VAR]CHAR (N) is expressed in bytes, not characters as in some other databases)

**PostgreSQL**

PostgreSQL supports following data types to store character data:

Table 266: PostgreSQL character data types

<table>
<thead>
<tr>
<th>PostgreSQL data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR (n)</td>
<td>SBCS or MBCS character data using the database character set, where n is specified in characters (max is 10485760 characters)</td>
</tr>
<tr>
<td>VARCHAR (n)</td>
<td>SBCS or MBCS character data using the database character set, where n is specified in characters (max is 10485760 characters); The length specification is optional.</td>
</tr>
<tr>
<td>TEXT</td>
<td>SBCS or MBCS character data using the database character set (max is 1Gb)</td>
</tr>
</tbody>
</table>

In PostgreSQL, CHAR, VARCHAR and TEXT types store data in single byte or multibyte character sets. For CHAR and VARCHAR, the size is specified in a number of characters, not bytes. The character set used to store data for these types is defined by the database character set, which can be specified when you create the database with the `createdb` tool or the CREATE DATABASE SQL command.

**Note:** The VARCHAR type of PostgreSQL can be used without a length specification. If no size is specified, the column accepts strings of any size. However, as Genero BDL needs to know the size of CHAR and VARCHAR columns to define fields and program variables from a schema file, it is not recommended to create tables in PostgreSQL having VARCHAR columns without size specification. If you try to extract a schema with `fgldbsch`, this tool will report that the VARCHAR column cannot be converted to a BDL type for the .sch file.

Automatic character set conversion between the PostgreSQL client and server is supported. You must properly specify the client character set for PostgreSQL. This can be done in different ways, with the SET CLIENT_ENCODING TO SQL command for example, or with configuration parameters. See the PostgreSQL documentation for more details.

**Solution**

Informix® CHAR (N) types must be mapped to PostgreSQL CHAR (N) types, and Informix® VARCHAR (N) or LVARCHAR (N) columns must be mapped to PostgreSQL VARCHAR (N).
**Note:** When creating a table from the BDL program with NCHAR or NVARCHAR types, the type names will be left as is and produce an SQL error because these types are not supported by PostgreSQL.

You can store single-byte or multibyte character strings in PostgreSQL CHAR, VARCHAR and TEXT columns.

PostgreSQL uses character length semantics: When you define a CHAR(20) and the database character set is multibyte, the column can hold more bytes/characters than the Informix® CHAR(20) type, when using byte length semantics.

When using a multibyte character set (such as UTF-8), define database columns with the size in character units, and use character length semantics in BDL programs with FGL_LENGTH_SEMANTICS=CHAR.

When extracting a database schema from a PostgreSQL database, the fgl dbsch schema extractor uses the size of the column in characters, not the octet length. If you have created a CHAR(10 (characters)) column a in PostgreSQL database using the UTF-8 character set, the .sch file will get a size of 10, that will be interpreted following FGL_LENGTH_SEMANTICS as a number of bytes or characters.

Do not forget to properly define the database client character set, which must correspond to the runtime system character set.

See also the section about Localization.

The CHAR/VARCHAR type translation can be controlled with the following FGLPROFILE entries:

```plaintext
dbi.database.dsnname.ifxemul.datatype.char = [ true | false ]
dbi.database.dsnname.ifxemul.datatype.varchar = [ true | false ]
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

**CHAR and VARCHAR types** on page 616
Using the CHAR and VARCHAR data types with different databases.

**Numeric data types**

**Informix®**

Informix® supports several data types to store numbers:

**Table 267: Informix® numeric data types**

<table>
<thead>
<tr>
<th>Informix® data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>16 bit signed integer</td>
</tr>
<tr>
<td>INTEGER</td>
<td>32 bit signed integer</td>
</tr>
<tr>
<td>BIGINT</td>
<td>64 bit signed integer</td>
</tr>
<tr>
<td>INT8</td>
<td>64 bit signed integer (replaced by BIGINT)</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>Equivalent to DECIMAL(16)</td>
</tr>
<tr>
<td>DECIMAL(p)</td>
<td>Floating-point decimal number (max precision is 32)</td>
</tr>
<tr>
<td>DECIMAL(p,s)</td>
<td>Fixed-point decimal number (max precision is 32)</td>
</tr>
<tr>
<td>MONEY</td>
<td>Equivalent to DECIMAL(16,2)</td>
</tr>
<tr>
<td>MONEY(p)</td>
<td>Equivalent to DECIMAL(p,2) (max precision is 32)</td>
</tr>
<tr>
<td>MONEY(p,s)</td>
<td>Equivalent to DECIMAL(p,s) (max precision is 32)</td>
</tr>
<tr>
<td>REAL / SMALLFLOAT</td>
<td>32-bit floating point decimal (C float)</td>
</tr>
</tbody>
</table>
PostgreSQL supports the following data types to store numbers:

**Table 268: PostgreSQL numeric data types**

<table>
<thead>
<tr>
<th>PostgreSQL data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT2 / SMALLINT</td>
<td>16 bit signed integer</td>
</tr>
<tr>
<td>INT4 / INTEGER</td>
<td>32 bit signed integer</td>
</tr>
<tr>
<td>INT8 / BIGINT</td>
<td>64 bit signed integer</td>
</tr>
<tr>
<td>DECIMAL(p, s)</td>
<td>Decimals with precision and scale (fractional part)</td>
</tr>
<tr>
<td>DECIMAL(p)</td>
<td>Integers with p digits (no fractional part)</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>Floating point numbers (no limit)</td>
</tr>
<tr>
<td>FLOAT4</td>
<td>16 bit variable precision</td>
</tr>
<tr>
<td>FLOAT8</td>
<td>32 bit variable precision</td>
</tr>
</tbody>
</table>

ANSI types like SMALLINT, INTEGER, FLOAT are supported by PostgreSQL as aliases to INT2, INT4 and FLOAT8 native types.

Informix® DECIMAL (p) floating point types are converted to DECIMAL without precision/scale, to store any floating point number in PostgreSQL.

**Solution**

Use the following conversion rules to map Informix numeric types to PostgreSQL numeric types:

**Table 269: Informix® data types and PostgreSQL equivalents**

<table>
<thead>
<tr>
<th>Informix® data type</th>
<th>PostgreSQL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>INT2</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INT4</td>
</tr>
<tr>
<td>INT8 / BIGINT</td>
<td>INT8</td>
</tr>
<tr>
<td>DECIMAL(p, s)</td>
<td>DECIMAL(p, s)</td>
</tr>
<tr>
<td>DECIMAL(p)</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>MONEY(p, s)</td>
<td>DECIMAL(p, s)</td>
</tr>
<tr>
<td>MONEY(p)</td>
<td>DECIMAL(p, 2)</td>
</tr>
<tr>
<td>MONEY</td>
<td>DECIMAL(16, 2)</td>
</tr>
<tr>
<td>SMALLFLOAT</td>
<td>FLOAT4</td>
</tr>
<tr>
<td>FLOAT[(n)]</td>
<td>FLOAT8</td>
</tr>
</tbody>
</table>
The numeric types translation can be controlled with the following FGLPROFILE entries:

```
dbi.database.dsname.ifxemul.datatype.smallint = \{ true \} \{ false \}
dbi.database.dsname.ifxemul.datatype.integer = \{ true \} \{ false \}
dbi.database.dsname.ifxemul.datatype.bigint = \{ true \} \{ false \}
dbi.database.dsname.ifxemul.datatype.int8 = \{ true \} \{ false \}
dbi.database.dsname.ifxemul.datatype.decimal = \{ true \} \{ false \}
dbi.database.dsname.ifxemul.datatype.money = \{ true \} \{ false \}
dbi.database.dsname.ifxemul.datatype.float = \{ true \} \{ false \}
dbi.database.dsname.ifxemul.datatype.smallfloat = \{ true \} \{ false \}
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

**Using portable data types** on page 614

Only a limited set of data types are really portable across several database engines.

**DATE and DATETIME data types**

**Informix®**

Informix® provides two data types to store date and time information:

- **DATE** = for year, month and day storage.
- **DATETIME** = for year to fraction (1-5) storage.

The DATE type is stored as an INTEGER with the number of days since 1899/12/31.

The DATETIME type can be defined with various time units, by specifying a start and end qualifier. For example, you can define a datetime to store an hour-to-second time value with **DATE DATETIME HOUR TO SECOND**.

The values of Informix® DATETIME can be represented with a character string literal, or as **DATETIME()** literals:

```
'2017-12-24 15:45:12.345' -- a DATETIME YEAR TO FRACTION(3)
'15:45' -- a DATETIME HOUR TO MINUTE
DATETIME(2017-12-24 12:45) YEAR TO MINUTE
DATETIME(12:45:56.333) HOUR TO FRACTION(3)
```

Informix® is able to convert quoted strings to DATE/DATETIME data, if the string contains matching environment parameters. The string to date conversion rules for DATE is defined by the DBDATE environment variable. The string to datetime format for DATETIME is defined by the GL_DATETIME environment variable.

**Note:** Within Genero programs, the string representation for DATETIME values is always ISO (YYYY-MM-DD hh:mm:ss.fffffff)

Informix® supports date arithmetic on DATE and DATETIME values. The result of an arithmetic expression involving dates/times is an INTEGER number of days when only DATE values are used, and an INTERVAL value if a DATETIME is used in the expression.

Informix® automatically converts an INTEGER to a DATE when the integer is used to set a value of a date column.

Informix® provides the **CURRENT q1 TO q2** operator, to get the system date/time on the server where the current database is located. When no qualifiers are specified, **CURRENT** returns a **DATE TIME YEAR TO FRACTION(3)**. Informix also supports the **SYSDATE** operator, which returns the current system time as a **DATETIME YEAR TO FRACTION(5)**.

**Note:** The **USEOSTIME** configuration parameter must be set to 1 in order to get the subsecond precision in **CURRENT** and **SYSDATE** operators. See Informix documentation for more details.

**PostgreSQL**

PostgreSQL supports the following data types to store date/time values:
Table 270: PostgreSQL date/time data types

<table>
<thead>
<tr>
<th>PostgreSQL data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>for year, month, day storage</td>
</tr>
<tr>
<td>TIME (p) (\text{WITH</td>
<td>WITHOUT}) TIME ZONE</td>
</tr>
<tr>
<td>TIMESTAMP (p) (\text{WITH</td>
<td>WITHOUT}) TIME ZONE</td>
</tr>
</tbody>
</table>

When the precision \(p\) is not specified, the PostgreSQL time/timestamp type precision of the fractional part is not fixed.

PostgreSQL can convert quoted strings to date time data depending on the `DateStyle` session parameter. PostgreSQL always accepts ISO date time strings.

With PostgreSQL, the date format can be defined with the `SET DATESTYLE` SQL command.

With PostgreSQL, the result of an arithmetic expression involving `DATE` values is an `INTEGER` representing a number of days.

Solution

Use the following conversion rules to map Informix® date/time types to PostgreSQL date/time types:

Table 271: Informix® data types and PostgreSQL equivalents

<table>
<thead>
<tr>
<th>Informix® data type</th>
<th>PostgreSQL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME HOUR TO MINUTE</td>
<td>TIME(0) WITHOUT TIME ZONE</td>
</tr>
<tr>
<td>DATETIME HOUR TO SECOND</td>
<td>TIME(0) WITHOUT TIME ZONE</td>
</tr>
<tr>
<td>DATETIME HOUR TO FRACTION(p)</td>
<td>TIME(p) WITHOUT TIME ZONE</td>
</tr>
<tr>
<td>DATETIME YEAR TO MONTH</td>
<td>TIMESTAMP(0) WITHOUT TIME ZONE</td>
</tr>
<tr>
<td>DATETIME YEAR TO DAY</td>
<td>TIMESTAMP(0) WITHOUT TIME ZONE</td>
</tr>
<tr>
<td>DATETIME YEAR TO HOUR</td>
<td>TIMESTAMP(0) WITHOUT TIME ZONE</td>
</tr>
<tr>
<td>DATETIME YEAR TO MINUTE</td>
<td>TIMESTAMP(0) WITHOUT TIME ZONE</td>
</tr>
<tr>
<td>DATETIME YEAR TO SECOND</td>
<td>TIMESTAMP(0) WITHOUT TIME ZONE</td>
</tr>
<tr>
<td>DATETIME YEAR TO FRACTION(p)</td>
<td>TIMESTAMP(p) WITHOUT TIME ZONE</td>
</tr>
</tbody>
</table>

Note: When the PostgreSQL `TIME/TIMESTAMP` precision \(p\) is not specified, the number of digits in the fractional part is variable. This prevents a proper type mapping, for example when extracting column types with `fgldbsch` from a PostgreSQL database. When creating table columns in PostgreSQL, always use `TIME\(p\)/TIMESTAMP\(p\)` types, and specify 0 (zero) if there are no fraction of seconds to be stored.

The `DATE` and `DATETIME` types translation can be controlled with the following `FGLPROFILE` entries:

```sql
  dbi.database.dsnname.ifxemul.datatype.date = \{ true \} false \}
  dbi.database.dsnname.ifxemul.datatype.datetime = \{ true \} false \}
```

For more details see IBM Informix emulation parameters in `FGLPROFILE` on page 668.

PostgreSQL and Informix® `DATE` types are equivalent and store year, month, day values.
**Important:** (FGL-4680): Starting with Genero 3.10.07, the PostgreSQL ODI driver can detect the current date format used in the SQL session, and you can safely use `SET DATESTYLE` if needed. Older versions of Genero PostgreSQL expect `DATE` values in ISO format. Do not change the `DATE` format with the `SET DATESTYLE` instruction with Genero versions before 3.10.07.

PostgreSQL `TIME(n)` WITHOUT TIME ZONE data type can be used to store Informix® `DATETIME HOUR TO FRACTION(n)`, `DATETIME HOUR TO SECOND`, `DATETIME HOUR TO MINUTE` values, and any other `DATETIME` type with qualifiers `HOUR`, `MINUTE`, `SECOND` and `FRACTION(n)` . Missing time parts default to `00:00:00.0`. For example, when using a `DATETIME MINUTE TO FRACTION(3)` with the value of "45:23.999", the PostgreSQL `TIME(3)` WITHOUT TIME ZONE value will be "00:45:23.999".

Informix® `DATETIME` values with any qualifiers from `YEAR` to `FRACTION(5)` can be stored in PostgreSQL `TIMESTAMP(n)` WITHOUT TIME ZONE columns. Missing date or time parts default to 1900-01-01 00:00:00.0. For example, when using a `DATETIME DAY TO MINUTE` with the value of "23 11:45", the PostgreSQL `TIMESTAMP(0)` WITHOUT TIME ZONE value will be "1900-01-23 11:45:00".

**Note:** Complex `DATETIME` expressions (involving `INTERVAL` values for example) are Informix-specific and have no equivalent in PostgreSQL.

**Related concepts**
Date/time literals in SQL statements on page 635
Good practices for date and time handling in SQL.

**INTERVAL data type**

**Informix**
Informix® provides the `INTERVAL` data type to store a value that represents a span of time.

INTERVAL types are divided into two classes:

- **year-month** intervals. For example: `INTERVAL YEAR(5) TO MONTH`
- **day-time** intervals. For example: `INTERVAL DAY(9) TO SECOND`

INTERVAL columns can be defined with various time units, by specifying a start and end qualifier. For example, you can define an interval to store a number of hours and minutes with `INTERVAL HOUR(n) TO MINUTE`, where `n` defines the maximum number of digits for the hours unit.

The values of Informix® `INTERVAL` can be represented with a character string literal, or as `INTERVAL()` literals:

```
'-9834 15:45:12.345'  -- an INTERVAL DAY(6) TO FRACTION(3)
'7623-11'   -- an INTERVAL YEAR(9) TO MONTH
INTERVAL(18734:45) HOUR(5) TO MINUTE
INTERVAL(-7634-11) YEAR(5) TO MONTH
```

**PostgreSQL**
PostgreSQL provides an `INTERVAL` data type which is equivalent to the Informix® `INTERVAL`:

- It is possible to specify the interval class / precision with `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE` and `SECOND` [(p)] fields.
- Fractional part of seconds can be defined with up to 6 digits.
- The interval value range is from -178000000 to +178000000 years.
- Input and output format can be controlled with the `SET interval style` command.

**Solution**
Use the following conversion rules to map Informix® numeric types to PostgreSQL numeric types:
### Table 272: Informix® data types and PostgreSQL equivalents

<table>
<thead>
<tr>
<th>Informix® data type</th>
<th>PostgreSQL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTERVAL YEAR[(p)] TO MONTH</td>
<td>INTERVAL YEAR TO MONTH</td>
</tr>
<tr>
<td>INTERVAL YEAR[(p)] TO YEAR</td>
<td>INTERVAL YEAR</td>
</tr>
<tr>
<td>INTERVAL MONTH[(p)] TO MONTH</td>
<td>INTERVAL MONTH</td>
</tr>
<tr>
<td>INTERVAL DAY[(p)] TO FRACTION(n)</td>
<td>INTERVAL DAY TO SECOND(n)</td>
</tr>
<tr>
<td>INTERVAL DAY[(p)] TO SECOND</td>
<td>INTERVAL DAY TO SECOND(0)</td>
</tr>
<tr>
<td>INTERVAL DAY[(p)] TO MINUTE</td>
<td>INTERVAL DAY TO MINUTE</td>
</tr>
<tr>
<td>INTERVAL DAY[(p)] TO HOUR</td>
<td>INTERVAL DAY TO HOUR</td>
</tr>
<tr>
<td>INTERVAL HOUR[(p)] TO FRACTION(n)</td>
<td>INTERVAL HOUR TO SECOND(n)</td>
</tr>
<tr>
<td>INTERVAL HOUR[(p)] TO SECOND</td>
<td>INTERVAL HOUR TO SECOND(0)</td>
</tr>
<tr>
<td>INTERVAL HOUR[(p)] TO MINUTE</td>
<td>INTERVAL HOUR TO MINUTE</td>
</tr>
<tr>
<td>INTERVAL MINUTE[(p)] TO FRACTION(n)</td>
<td>INTERVAL MINUTE TO SECOND(n)</td>
</tr>
<tr>
<td>INTERVAL MINUTE[(p)] TO SECOND</td>
<td>INTERVAL MINUTE TO SECOND(0)</td>
</tr>
<tr>
<td>INTERVAL SECOND[(p)] TO MINUTE</td>
<td>INTERVAL SECOND</td>
</tr>
<tr>
<td>INTERVAL SECOND[(p)] TO SECOND</td>
<td>INTERVAL SECOND(0)</td>
</tr>
<tr>
<td>INTERVAL FRACTION TO FRACTION(n)</td>
<td>INTERVAL SECOND</td>
</tr>
</tbody>
</table>

The PostgreSQL database interface converts the Informix-style `INTERVAL` type to the native PostgreSQL `INTERVAL` type.

**Important:** The PostgreSQL database driver forces the interval style session parameter to 'iso_8601', this is required to insert and fetch interval database with the libpq CAPI functions. You must not change this setting during program execution.

While PostgreSQL intervals support up to 9 digits for the higher unit like Informix®, year values range from -178000000 to +178000000 only. This limitation exists in PostgreSQL 8.4 and maybe solved in future versions.

The `INTERVAL` types translation can be controlled with the following FGLPROFILE entry:

```
  dbi.database.dsnname.ifxemul.datatype.interval = \true \false
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

- Using portable data types on page 614

Only a limited set of data types are really portable across several database engines.

### SERIAL and BIGSERIAL data types

**Informix®**

Informix® supports the SERIAL, BIGSERIAL data types to produce automatic integer sequences:

- SERIAL can produce 32 bit integers (INTEGER)
- **BIGSERIAL** can produced 64 bit integers (**BIGINT**)
- **SERIAL8** is a synonym for **BIGSERIAL**

Steps to use serials with Informix®:

1. Create the table with a column using **SERIAL**, or **BIGSERIAL**.
2. To generate a new serial, no value or a zero value is specified in the **INSERT** statement:
   ```sql
   INSERT INTO tab1 ( c ) VALUES ( 'aa' )
   INSERT INTO tab1 ( k, c ) VALUES ( 0, 'aa' )
   ```
3. After **INSERT**, the new value of a **SERIAL** column is provided in **SQLCA.SQLERRD[2]**, while the new value of a **BIGSERIAL** value must be fetched with a **SELECT dbinfo('bigserial')** query.

Informix® allows you to insert rows with a value different from zero for a serial column. Using an explicit value will automatically increment the internal serial counter, to avoid conflicts with future **INSERT** statements that are using a zero value:

```sql
CREATE TABLE tab ( k SERIAL); -- internal counter = 0
INSERT INTO tab VALUES ( 0 ); -- internal counter = 1
INSERT INTO tab VALUES ( 10 ); -- internal counter = 10
INSERT INTO tab VALUES ( 0 ); -- internal counter = 11
DELETE FROM tab; -- internal counter = 11
INSERT INTO tab VALUES ( 0 ); -- internal counter = 12
```

**PostgreSQL**

PostgreSQL **SERIAL** data type:

- PostgreSQL's **SERIAL** data type has the same name as in Informix®, but it behaves differently.
- You cannot define a start value (**SERIAL(100)**).
- You cannot specify zero as serial value to get a new serial, the PostgreSQL serial is based on default values, thus you must omit the serial column in the **INSERT** statement.
- When you **INSERT** a row with a specific value for the serial column, the underlying sequence will not be incremented. As result, the next **INSERT** that does not specify the serial column may get a new sequence that was already inserted explicitly.
- With some old versions of PostgreSQL, when you drop the table you must drop the sequence too.

PostgreSQL sequences:

- The purpose of sequences is to provide unique integer numbers.
- To create a sequence, you must use the **CREATE SEQUENCE** statement.
- To get a new sequence value, you must use the **nextval()** function:
  ```sql
  INSERT INTO tab1 VALUES ( nextval('tab1_seq'), ... )
  ```
- To get the last generated number, PostgreSQL provides the **currval()** function:
  ```sql
  SELECT currval('tab1_seq')
  ```

**Solution**

**Note:** For best SQL portability when using different types of databases, consider using sequences as described in Solution 3: Use native SEQUENCE database objects on page 628.

The Informix® **SERIAL** data type can be emulated with three different methods.
The method used to emulate SERIAL types is defined by the ifxemul.datatype.serial.emulation FGLPROFILE parameter:

```
dbi.database.dbname.ifxemul.datatype.serial.emulation = "native"|"regtable"|"trigseq"
```

- **native**: uses the native PostgreSQL serial data type.
- **regtable**: uses insert triggers with the SERIALREG table.
- **trigseq**: uses insert triggers with sequences.

The default emulation technique is "native".

The serial types emulation can be enabled or disabled with the following FGLPROFILE entries:

```
dbi.database.dbname.ifxemul.datatype.serial = [true|false]
dbi.database.dbname.ifxemul.datatype.serial8 = [true|false]
dbi.database.dbname.ifxemul.datatype.bigserial = [true|false]
```

### Disabling automatic serial retrieval for SQLCA.SQLERRD[2]

SERIAL emulation can be totally disabled by setting the ifxemul.datatype.serial FGLPROFILE entry to false:

```
dbi.database.dbname.ifxemul.datatype.serial = false
```

For Informix® compatibility, after an INSERT statement, the ODI drivers automatically execute another SQL query (or do a DB client API call when possible), to get the last generated serial, and fill the SQLCA.SQLERRD[2] register. This results in some overhead that can be avoided, if the SQLCA.SQLERRD[2] register is not used by the program.

When SERIAL emulation is required (to create temp tables with a serial column during program execution), and the SQLCA.SQLERRD[2] register does not need to be filled, (typically because you use your own method to retrieve the last generated serial), you can set the ifxemul.datatype.serial.sqlerrd2 FGLPROFILE entry to false. This will avoid the automatic retrieval of last serial value to fill SQLCA.SQLERRD[2]:

```
dbi.database.dbname.ifxemul.datatype.serial.sqlerrd2 = false
```

See also `db_get_last_serial()` on page 2308.

### Using the native serial emulation

The "native" mode is the default serial emulation mode, using the native PostgreSQL SERIAL data type. In this mode, the original type name will be left untouched by the SQL Translator and you will get the behavior of the PostgreSQL SERIAL column type, based on sequences.

**Note:** INSERT statements cannot use the serial column, even with a value zero. When using a NULL value, PostgreSQL will report an non-null constraint error. Therefore, the serial column must be omitted from the INSERT statement.

See also the PostgreSQL documentation for more details about the native SERIAL type.

### Using the regtable serial emulation

With the "regtable" mode, the SERIAL data type is emulated with a PostgreSQL INTEGER data type and INSERT triggers using the table SERIALREG which is dedicated to sequence production. After an insert, sqlca.sqlerrd[2] register holds the last generated serial value. BIGSERIAL and SERIAL8 types can be converted to BIGINT in PostgreSQL, but the sqlca.sqlerrd[2] register cannot be used since it is defined as an INTEGER type.
The triggers can be created manually during the application database installation procedure, or automatically from a BDL program: When a BDL program executes a CREATE [TEMP] TABLE with a SERIAL column, the database interface automatically converts the SERIAL data type to INTEGER and dynamically creates the triggers.

You must create the SERIALREG table as follows:

```
CREATE TABLE SERIALREG (
    TABLENAME VARCHAR(50) NOT NULL,
    LASTSERIAL DECIMAL(20,0) NOT NULL,
    PRIMARY KEY ( TABLENAME )
)
```

**Important**: The SERIALREG table must be created before the triggers. The serial production is based on the SERIALREG table which registers the last generated number for each table. If you delete rows of this table, sequences will restart at 1 and you will get unexpected data.

In database creation scripts, all SERIAL[(n)] data types must be converted to INTEGER data types and you must create one trigger for each table. To know how to write those triggers, you can create a small Genero program that creates a table with a SERIAL column. Set the FGLSQLDEBUG environment variable and run the program. The debug output will show you the native trigger creation command.

With this emulation mode, INSERT statements using NULL for the SERIAL column will produce a new serial value:

```
INSERT INTO tab ( col1, col2 ) VALUES ( NULL, 'data' )
```

This behavior is mandatory in order to support INSERT statements that do not use the serial column:

```
INSERT INTO tab (col2) VALUES ('data')
```

Check if your application uses tables with a SERIAL column that can contain a NULL value. Consider removing the serial column from the INSERT statements.

**Using the trigseq serial emulation**

With "trigseq", the SERIAL data type is emulated with a PostgreSQL INTEGER data type and INSERT triggers using a sequence tablename_seq. After an insert, sqlca.sqlerrd[2] register holds the last generated serial value.

The triggers can be created manually during the application database installation procedure, or automatically from a BDL program: When a BDL program executes a CREATE [TEMP] TABLE with a SERIAL column, the database interface automatically converts the SERIAL data type to INTEGER and dynamically creates the triggers.

In database creation scripts, all SERIAL[(n)] data types must be converted to INTEGER data types and you must create one trigger for each table. To know how to write those triggers, you can create a small Genero program that creates a table with a SERIAL column. Set the FGLSQLDEBUG environment variable and run the program. The debug output will show you the native trigger creation command.

With this emulation mode, INSERT statements using NULL for the SERIAL column will produce a new serial value:

```
INSERT INTO tab ( col1, col2 ) VALUES ( NULL, 'data' )
```

This behavior is mandatory in order to support INSERT statements which do not use the serial column:

```
INSERT INTO tab (col2) VALUES ('data')
```

Check if your application uses tables with a SERIAL column that can contain a NULL value. Consider removing the serial column from the INSERT statements.
Notes common to all serial emulation modes

Since sqlca.sqlerrd[2] is defined as an INTEGER, it cannot hold values from BIGSERIAL (BIGINT) auto incremented columns. If you are using BIGSERIAL columns, you must query the sequence pseudo-column CURRVAL() or fetch the LASTSERIAL column from the SERIALREG table if used.

For SQL portability, it is recommended to review INSERT statements to remove the SERIAL column from the list.

For example, the following statement:

```
INSERT INTO tab (col1,col2) VALUES ( 0 , p_value )
```

can be converted to:

```
INSERT INTO tab (col2) VALUES (p_value)
```

Static SQL INSERT using records defined from the schema file must also be reviewed:

```
DEFINE rec LIKE tab.*
INSERT INTO tab VALUES ( rec.*) -- will use the serial column
```

can be converted to:

```
INSERT INTO tab VALUES rec.* -- without parentheses, serial column is removed
```

**Important:** When using the Static SQL INSERT and UPDATE syntax using record.* without parentheses, make sure that you database schema files contain information about serials: This information can be lost when extracting the schema from a PostgreSQL database which does not use native serial emulation. See Database Schema for more details about the serial flag in column type encoding (data type code must be 6)

**Related concepts**

- [Auto-incremented columns (serials)] on page 626
- How to implement automatic record keys.

**ROWID columns**

**Informix®**

When creating a table, Informix® automatically adds a ROWID integer column (applies to non-fragmented tables only).

The ROWID column is auto-filled with a unique number and can be used like a primary key to access a given row.

**Note:** Informix® ROWID usage was a common practice in the early days of Informix® 4GL programming. Today it is recommended to define all your database tables with a PRIMARY KEY to uniquely identify rows.

With Informix®, the SQLCA.SQLERRD[6] register contains the ROWID of the last row affected by an INSERT, UPDATE or DELETE statement.

**PostgreSQL**

With PostgreSQL versions prior to version 12, tables can be created with an OID (Object Identifier) column, that is similar to Informix® ROWID columns.

**Important:** Starting with PostgreSQL version 12, OID columns are no longer supported, and the ROWID emulation is no longer possible with the PostgreSQL ODI driver.

The type of PostgreSQL OID columns is INTEGER.

**Important:** Since PostgreSQL version 8.1, OID columns are no longer enabled by default. To have OID columns created for tables, you need to set the default_with_oid parameter to "on" in the postgresql.conf file.
The PostgreSQL C API provides the PQoidValue() function, to get the OID of the last inserted row. Unlike Informix® SQLCA.SQLERRD[6], PostgreSQL does not return the OID of the last row modified by an UPDATE, or removed by a DELETE.

Solution

For PostgreSQL versions prior to version 12, the database can automatically convert ROWID keywords to OID.

If PostgreSQL OID columns are available and enabled in the database, SQL statements such as "SELECT ROWID FROM" and "UPDATE .. WHERE ROWID = ?" will work as with Informix®.

To enable OID columns with PostgreSQL versions prior to version 12, the default_with_oid PostgreSQL server parameter must be set to "on". See Database configuration and design tasks for more details.

Note: SQLCA.SQLERRD[6] is partly supported: It works only for INSERT statements. As PostgreSQL does not return the OID of the last row modified by an UPDATE, or removed by a DELETE, this register will not contain the OID of the last modified or deleted row. All references to SQLCA.SQLERRD[6] should be reviewed.

As a general programming pattern, consider using PRIMARY KEY columns to identify database table rows.

For databases where the keyword of the rowid pseudo-column is different than "ROWID", the translation can be controlled with the following FGLPROFILE entry:

```
dbi.database.dsnname.ifxemul.rowid = true false
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

Using ROWID columns on page 644

Automatic ROWID columns is not a common database feature.

**TEXT and BYTE (LOB) types**

**Informix**

Informix® provides the TEXT, BYTE, CLOB and BLOB data types to store very large texts or binary data.

Legacy Informix® 4GL applications typically use the TEXT and BYTE types.

Genero BDL does not support the Informix® CLOB and BLOB types.

**PostgreSQL**

PostgreSQL provides the TEXT and BYTEA data types for large objects storage. With these data types, large objects are handled as a whole.

PostgreSQL also supports LOB storage through the large objects facility based on stream-style access. The large object facility is provided as a set of C and SQL API functions to create / delete / modify large objects identified by a unique object id (OID). For example, the lo_create(-1) SQL function will create a new large object and return a new object id that will be used to handle the LOB. See PostgreSQL documentation for more details.

**Solution**

TEXT and BYTE data can be stored in PostgreSQL TEXT and BYTEA columns.

The TEXT and BYTE types translation can be controlled with the following FGLPROFILE entries:

```
dbi.database.dsnname.ifxemul.text = true false
dbi.database.dsnname.ifxemul.byte = true false
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.
Genero BDL does not interface automatically with the PostgreSQL Large Object facility. However, the OID values can be stored in BIGINT variables, and you can use server-side LOB functions to convert large objects to BYTEA data, that can be fetched into BYTE variables. The next code example creates a table with an OID column, imports a LOB from an image file, and then fetches the LOB back into a BYTE:

```sql
MAIN
  DEFINE img BYTE, obj_id BIGINT

  CONNECT TO "test1+driver='dbmpgs'" USER "postgres" USING "fourjs"

  # Need superuser privileges to create the LOB....
  WHENEVER ERROR CONTINUE
  DROP TABLE t1
  WHENEVER ERROR STOP
  EXECUTE IMMEDIATE "create table t1 ( k int, image oid )"
  GRANT SELECT ON t1 TO PUBLIC
  INSERT INTO t1 VALUES ( 1, lo_import("/var/images/landscape.png") )
  DISPLAY "obj_id = ", obj_id
  EXECUTE IMMEDIATE "grant select on large object "||obj_id||" to public"

  # Next block can be executed by any user:
  LOCATE img IN FILE -- a temp file will be used
  SELECT loread(lo_open(image, 262144), 1000000)
    INTO img FROM t1 WHERE k=1
  DISPLAY length(img)

  # Delete the object
  SELECT lo_unlink(obj_id) FROM t1 WHERE k=1

  DROP TABLE t1
END MAIN
```

Related concepts
- Using portable data types on page 614

Only a limited set of data types are really portable across several database engines.

**Table constraints**

**Informix**

Informix® supports primary key, unique, foreign key, default and check constraints.

The constraint naming syntax is different in Informix® and most other databases: Informix expects the constraint name after the constraint definition:

```sql
CREATE TABLE emp (...
    emp_code CHAR(10) UNIQUE CONSTRAINT pk_emp,
    ...
)
```

While other SQL database brands require to specify the constraint name before the constraint definition:

```sql
CREATE TABLE emp (...
    emp_code CHAR(10) CONSTRAINT pk_emp UNIQUE,
    ...
)
```
**PostgreSQL**

PostgreSQL support primary key, unique, foreign key, default and check constraints.

**Constraint naming syntax**

The constraint naming clause must be placed before the constraint specification.

**Unique constraints**

*Note:* When using a unique constraint, Informix® allows only one row with a NULL value, while PostgreSQL allows several rows with NULL!

**Solution**

The database interface does not convert constraint naming expressions when creating tables from BDL programs. Review the database creation scripts to adapt the constraint naming clauses for PostgreSQL.

**Related concepts**

Data definition statements on page 613

It is recommended to avoid use of DDL in programs.

**Name resolution of SQL objects**

**Informix®**

Informix® uses the following form to identify an SQL object:

```
database[@dbservername]::{owner|"owner"}.]identifier
```

The ANSI convention is to use double quotes for identifier delimiters (For example: "customer"."cust_name").

Informix® database object names are not case-sensitive in non-ANSI databases. When using double-quoted identifiers, Informix® becomes case sensitive.

With non-ANSI Informix® databases, you do not have to give a schema name before the tables when executing an SQL statement:

```
SELECT ... FROM customer WHERE ...
```

In Informix® ANSI compliant databases:

- The table name must include "owner", unless the connected user is the owner of the database object.
- The database server shifts the owner name to uppercase letters before the statement executes, unless the owner name is enclosed in double quotes.

**PostgreSQL**

With PostgreSQL, a database object name takes the following form:

```
[[database.]schema.]identifier
```

You can define a list of schemas to be searched when the database object is not qualified. The list of schemas to look in is specified with `SET search_path TO schema-1, schema-2,...` for an SQL session (it can also be defined at the database level and user definition level).

When creating a table without a schema prefix, the table is created in the "public" schema. This schema is listed in the search path by default.
Solution

To write portable SQL, regarding database object names:

1. Use simple database object names (without any owner/schema prefix)
2. Do not use double quotes to surround database object identifiers.
3. If needed, define public synonyms to reference database objects in others databases/schema.
4. Specify database object identifiers in lowercase.

See also Naming database objects on page 637.

Use the following FGLPROFILE entry to define the schema search path for programs connecting to PostgreSQL:

```
dbi.database.stores.pgs.schema = "\"$user\",public,stock"
```

For more details see PostgreSQL specific FGLPROFILE parameters on page 676.

Data manipulation

PostgreSQL related data manipulation topics.

Reserved words

Informix®

With Informix®, it is possible to create database objects with reserved words.

For example:

```
CREATE TABLE table ( char CHAR(10) );
```

Indeed this is not good practice, but Informix® SQL allows this to be backward compatible when introducing a new keyword in the SQL syntax.

Most other database systems do not allow reserved words as database identifiers. If your legacy code is using SQL reserved words of the target database SQL syntax, an error will be thrown at CREATE TABLE execution.

PostgreSQL

SQL object names like table and column names cannot be SQL reserved words in PostgreSQL.

Solution

Table or column names which are PostgreSQL reserved words must be renamed.

Outer joins

Informix® OUTER() syntax

In Informix® SQL, outer joins can be defined in the FROM clause with the OUTER keyword:

```
SELECT ... FROM a, OUTER (b) 
WHERE a.key = b.akey

SELECT ... FROM a, OUTER (b, OUTER (c)) 
WHERE a.key = b.akey
AND b.key1 = c.bkey1 AND b.key2 = c.bkey2
```

Informix® also supports the ANSI OUTER join syntax, which is the recommended way to specify outer joins with recent SQL database engines:

```
SELECT ... FROM cust LEFT OUTER JOIN order 
ON cust.key = order.custno
```
PostgreSQL

PostgreSQL supports the ANSI outer join syntax:

```
SELECT ... 
FROM cust LEFT OUTER JOIN order 
    LEFT OUTER JOIN item 
    ON order.key = item.ordno 
    ON cust.key = order.custno 
WHERE order.cdate > current date
```

Solution

The Genero database drivers can convert Informix Informix® OUTER specifications to ANSI outer joins.

**Note:** For better SQL portability, use the ANSI outer join syntax instead of the old Informix® OUTER syntax.

The outer join translation can be controlled with the following FGLPROFILE entry:

```
dbi.database.dsname.ifxemul.outers = true,false
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

- **Prerequisites:**
  1. In the FROM clause, the main table must be the first item and the outer tables must be listed from left to right in the order of outer levels.

     Example which does not work:

     ```
     ... FROM OUTER(tab2), tab1
     ```

  2. The outer join in the WHERE clause must use the table name as prefix:

     ```
     ... WHERE tab1.col1 = tab2.col2
     ```

- **Restrictions:**
  1. Statements composed by 2 or more SELECT instructions are not supported:

     ```
     SELECT ... UNION SELECT ...
     ```

     or:

     ```
     SELECT ... WHERE col IN (SELECT...)
     ```

  2. Additional conditions on outer table columns cannot be detected and therefore are not supported:

     ```
     ... FROM tab1, OUTER(tab2) 
     WHERE tab1.col1 = tab2.col2 
     AND tab2.colx > 10
     ```

  3. Using subscript in outer conditions:

     ```
     ... FROM tab1, OUTER(tab2) 
     WHERE tab1.col1[1,3] = tab2.col2[1,3]
     ```

- **Notes:**
  1. Table aliases are detected in OUTER expressions.
OUTER example with table alias:

... OUTER(tab1 alias1) ...

2. In the outer join, `outertab.col` can be placed on both right or left sides of the equal sign:

... WHERE outertab.col1 = maintab.col2

3. Table names detection is not case-sensitive:

```sql
SELECT ... FROM tab1, TAB2
WHERE tab1.col1 = tab2.col2
```

4. Temporary tables are supported in OUTER specifications:

```sql
CREATE TEMP TABLE tt1 ( ... )
SELECT ... FROM tab1, OUTER(tt1) ...
```

**Related concepts**

Outer joins on page 643

Use standard ISO outer join syntax instead of the old IBM® Informix® OUTER() syntax.

**Transactions handling**

**Informix®**

With the Informix® native mode (non ANSI):

- Transactions blocks start with `BEGIN WORK` and terminate with `COMMIT WORK` or `ROLLBACK WORK`.
- Statements executed outside a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.

```sql
UPDATE tab1 SET ... -- auto-committed
BEGIN WORK -- start of TX block
UPDATE tab1 SET ...
UPDATE tab2 SET ...
... COMMIT WORK -- end of TX block
```

Informix® version 11.50 introduces savepoints:

```sql
SAVEPOINT name [UNIQUE]
ROLLBACK [WORK] TO SAVEPOINT [name]  
RELEASE SAVEPOINT name
```

**PostgreSQL**

PostgreSQL supports transaction with savepoints:

- Transactions are started with `BEGIN WORK`.
- Transactions are validated with `COMMIT WORK`.
- Transactions are canceled with `ROLLBACK WORK`.
- Savepoints can be placed with `SAVEPOINT name`.
- Transactions can be rolled back to a savepoint with `ROLLBACK TO SAVEPOINT name`.
- Savepoints can be released with `RELEASE SAVEPOINT name`.
- Statements executed outside of a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.
- If an SQL error occurs in a transaction, the whole transaction is aborted.
Transactions in stored procedures: avoid using transactions in stored procedures to allow the client applications to handle transactions, depending on the transaction model.

The main difference between Informix® and PostgreSQL resides in the fact that PostgreSQL cancels the entire transaction if an SQL error occurs in one of the statements executed inside the transaction. The following code example illustrates this difference:

```
CREATE TABLE tab1 ( k INT PRIMARY KEY, c CHAR(10) )
WHENEVER ERROR CONTINUE
BEGIN WORK
INSERT INTO tab1 ( 1, 'abc' )
INSERT INTO tab1 ( 1, 'abc' )
  -- PK constraint violation = SQL Error, whole TX is aborted
COMMIT WORK
```

With Informix®, this code will leave the table with one row inside, since the first INSERT statement succeeded. With PostgreSQL, the table will remain empty after executing this piece of code, because the server will rollback the whole transaction. To workaround this problem in PostgreSQL you can use SAVEPOINT as described in Solution on page 991.

**Solution**

Informix® transaction handling commands are automatically converted to PostgreSQL instructions to start, validate or cancel transactions.

Regarding the transaction control instructions, the BDL applications do not have to be modified in order to work with PostgreSQL.

You must review the SQL statements inside BEGIN WORK / COMMIT WORK instruction and check if these can raise an SQL error. The SQL statements that can potentially raise an SQL error must be protected with a SAVEPOINT. If an error occurs, just rollback to the savepoint:

```
MAIN
  DATABASE test1
  CREATE TABLE tab1 ( k INT PRIMARY KEY, c CHAR(10) )
  WHENEVER ERROR CONTINUE
  BEGIN WORK
  INSERT INTO tab1 VALUES ( 1, 'abc' )
  CALL sql_protect()
  INSERT INTO tab1 VALUES ( 1, 'abc' )
  -- PK constraint violation = SQL Error
  CALL sql_unprotect()
  COMMIT WORK
END MAIN

FUNCTION sql_protect()
  IF fgl_db_driver_type()!="pgs" THEN
    RETURN
  END IF
  SAVEPOINT _sql_protect_
END FUNCTION

FUNCTION sql_unprotect()
  IF fgl_db_driver_type()!="pgs" THEN
    RETURN
  END IF
  IF SQLCA.SQLCODE < 0 THEN
    ROLLBACK WORK TO SAVEPOINT _sql_protect_
  ELSE
    RELEASE SAVEPOINT _sql_protect_
  END IF
END FUNCTION
```
Note: If you want to use savepoints, do not use the UNIQUE keyword in the savepoint declaration, always specify the savepoint name in ROLLBACK TO SAVEPOINT, and do not drop savepoints with RELEASE SAVEPOINT.

Related concepts

Database transactions on page 600
Database transactions define a set of SQL instructions to be executed as a whole, or rolled back as a whole.

Temporary tables

Informix®

Informix® temporary tables are created with the CREATE TEMP TABLE DDL instruction or with SELECT ... INTO TEMP statement:

```sql
CREATE TEMP TABLE tt1 ( pkey INT, name VARCHAR(50) )
CREATE TEMP TABLE tt2 ( pkey INT, name VARCHAR(50) ) WITH NO LOG
SELECT * FROM tab1 WHERE pkey > 100 INTO TEMP tt2
```

Temporary tables are automatically dropped when the SQL session ends, but they can also be dropped with the DROP TABLE command. There is no name conflict when several users create temporary tables with the same name.

BDL REPORTs can create a temporary table when the rows are not sorted externally (by the source SQL statement).

Informix® allows you to create indexes on temporary tables. No name conflict occurs when several users create an index on a temporary table by using the same index identifier.

When creating temporary tables in Informix®, the WITH NO LOG clause can be used to avoid the overhead of recording DML operations in transaction logs.

PostgreSQL

PostgreSQL support temporary tables as Informix® does, with a little syntax difference in the SELECT INTO TEMP instruction:

```sql
SELECT * INTO TEMP temptab FROM source
```

Solution

Temporary tables are well supported with native PostgreSQL temp tables.

Important:

Simple Informix-style SQL statement creating temporary tables can be converted to a native SQL equivalent instruction. However, complex SQL statements such as SELECT ... INTO TEMP with subqueries may fail. In such cases, create a view from the complex query and then create the temp table from the view. Or, disable Informix emulation and use the native SQL syntax to create the temporary table (EXECUTE IMMEDIATE "/* fglhint_no_ifxemul */ ...")

With Informix SQL, if the source table has a column defined as SERIAL or BIGSERIAL, a SELECT ... INTO TEMP will produce a new temp table with an auto-incremented serial column. With the SELECT ... INTO TEMP emulation for non-Informix databases, not using the native sequence generators (such as IDENTITY columns in SQL Server), the resulting temporary table will get a simple INTEGER or BIGINT column, instead of an auto-incremented column.

The general FGLPROFILE entry to control temporary table emulation is:

```sql
dbi.database.dsname.ifxemul.temptables = { true | false }
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.
**Related concepts**

Temporary tables on page 641

Syntax for temporary table creation is not unique across all database engines.

**Substrings in SQL**

**Informix®**

Informix® SQL statements can use subscripts on columns defined with the character data type:

```
SELECT ... FROM tab1 WHERE col1[2,3] = 'RO'
SELECT ... FROM tab1 WHERE col1[10] = 'R' -- Same as col1[10,10]
UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...
SELECT ... FROM tab1 ORDER BY col1[1,3]
```

**Important:** With other database servers as Informix®, when the subscript notation is used to modify column values in UPDATE statement, or as ORDER BY element, you will get and SQL error:

```
UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...
SELECT ... FROM tab1 ORDER BY col1[1,3]
```

**PostgreSQL**

provides the `SUBSTRING( ... from ... to ... )` function, to extract a substring from a string expression:

```
SELECT .... FROM tab1 WHERE SUBSTRING(col1 from 2 for 2) = 'RO'
SELECT SUBSTRING('Some text' from 6 for 3) ... -- Gives 'tex'
```

**Solution**

Replace all Informix® `col[x,y]` right-value expressions by `SUBSTRING( col from x for (y-x+1) ).`

Rewrite UPDATE and ORDER BY clauses using `col[x,y]` expressions.

The translation of `col[x,y]` expressions can be controlled with the following FGLPROFILE entry:

```
dbi.database.dsname.ifxemul.colsubs = \{ true \| false \}
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

Substring expressions on page 644

Handle substrings expressions with different database engines.

**String delimiters**

**Informix®**

The ANSI SQL string delimiter character is the single quote (`'string'`), while double quotes are used to delimit database object names:

```
SELECT ... WHERE "tabname"."colname" = 'a string value'
```

In Informix® databases created in native mode (non-ANSI), you can use double quotes as string delimiters:

```
SELECT ... WHERE tabname.colname = 'a string value'
```

This is important, since many BDL programs use that character to delimit the strings in SQL commands.
**Note:** This problem concerns only double quotes within SQL statements. Double quotes used in pure BDL string expressions are not subject to SQL compatibility problems.

**PostgreSQL**

PostgreSQL follows the ANSI SQL specification, using single quotes for string delimiters and double quotes for database object names.

**Solution**

When using Static SQL statements, the fglcomp compiler converts string literals using double quotes to string literals with single quotes:

```bash
$ cat s.4gl
MAIN
  DEFINE n INT
  SELECT COUNT(*) INTO n FROM tab1 WHERE col1 = "abc"
END MAIN

$ fglcomp -S s.4gl
s.4gl^3^SELECT COUNT(*) FROM tab1 WHERE col1 = 'abc'
```

However, SQL statements created dynamically are not modified by the Genero compiler.

The Genero database interface can automatically replace all double quotes by single quotes in SQL statements. This applies to static and dynamic SQL statements.

The translation of double quoted expression to single quoted expressions can be controlled with the following FGLPROFILE entry:

```bash
dbi.database.dbname.ifxemul.dblquotes = {true | false}
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

However, database object names must not be delimited by double quotes, because the database interface cannot determine the difference between a database object name and a quoted string! For example, if the program executes the SQL statement:

```
... WHERE "tabname"."colname" = "a string value"
```

replacing all double quotes by single quotes would produce:

```
... WHERE 'tabname'.'colname' = 'a string value'
```

This would produce an error since 'tabname'.'colname' is not allowed by ORACLE.

Escaped string delimiters can be used inside strings like the following:

```
'This is a single quote: ''
'This is a single quote: "
"This is a double quote: ""'
"This is a double quote: ""
```

Although double quotes are replaced automatically in SQL statements, it is recommended that you use only single quotes to enforce portability.

**Related concepts**

[String literals in SQL statements](#) on page 634
Single quotes is the standard for delimiting string literals in SQL.

**MATCHES and LIKE**

**Informix®**

Informix® supports MATCHES and LIKE operators in SQL statements.

MATCHES expects * and ? wild-card characters, while LIKE uses the % and _ wild-cards as equivalents.

```
( col MATCHES 'Smi*' AND col NOT MATCHES 'R?x' )
( col LIKE 'Smi%' AND col NOT LIKE 'R_x' )
```

MATCHES accepts also brackets notation, to specify a set of matching characters at a given position:

```
( col MATCHES '[Pp]aris' )
( col MATCHES '[0-9][a-z]*' )
```

**PostgreSQL**

PostgreSQL supports the LIKE operator, and the ~ operator which expects regular expressions as follows: ( col ~ 'a.*' )

PostgreSQL provides the SIMILAR TO operator, allowing character range specification as the Informix MATCHES operator:

```
( col SIMILAR TO '[Pp]ar%' )
```

**Important:** With PostgreSQL, columns defined as CHAR(N) are blank padded, and trailing blanks are significant in the LIKE expressions. As result, with a CHAR(5) value such as 'abc ' (with 2 trailing blanks), the expression (colname LIKE 'ab_') will not match. To workaround this behavior, you can do (RTRIM(colname) LIKE 'pattern'). However, consider adding the condition AND (colname LIKE 'pattern%') to force the DB server to optimize the query of the column if indexed. The CONSTRUCT instruction uses this technique when the entered criteria does not end with a * star wildcard.

**Solution**

The database driver converts Informix® MATCHES expressions to LIKE expressions, when no [ ] bracket character ranges are used in the MATCHES operand. When character ranges are used, the driver converts to a PostgreSQL SIMILAR TO expression, to find the same values as with the Informix MATCHES operator.

**Important:** Only [NOT] MATCHES followed by a search pattern provided as a string literal can be converted by ODI drivers. A [NOT] MATCHES followed by a ? question mark parameter placeholder is not translated!

For maximum portability, consider replacing the MATCHES expressions to LIKE expressions in all SQL statements of your programs.

With PostgreSQL, trailing blanks are significant when comparing CHAR() columns with LIKE or SIMILAR TO expressions. Consider adding an ending * when comparing with CHAR() columns. This is not needed for VARCHAR() columns.

Avoid using CHAR(N) types for variable length character data (such as name, address).

**Related concepts**

MATCHES and LIKE operators on page 645
Use the standard LIKE operator instead of the MATCHES operator.

The LENGTH() function

**Informix®**
Informix® provides the LENGTH() function to count the number of bytes of a character string expression:

```sql
SELECT LENGTH("aaa"), LENGTH(col1) FROM table
```

Informix® LENGTH() does not count the trailing blanks for CHAR or VARCHAR expressions, while Oracle counts the trailing blanks.

Informix® LENGTH() returns 0 when the given string is empty. That means, LENGTH('') = 0.

**PostgreSQL**
PostgreSQL supports the LENGTH() function, which is similar to Informix® LENGTH().

The PostgreSQL LENGTH() function ignores trailing blanks.

When passing NULL as parameter, the PostgreSQL LENGTH() function returns NULL.

Solution
The SQL LENGTH() function name can be used with PostgreSQL.

Related concepts
The LENGTH() function in SQL on page 647
The semantics of the LENGTH() SQL function differs according to the database engine.

Row limiting clause

**Informix®**
Informix® SQL supports the SKIP and FIRST/LIMIT keywords to limit the number of rows of a result set:

```sql
SELECT SKIP 10 FIRST 20 customer.* FROM customer ... ORDER BY cust_name
```

This Informix SQL syntax is not portable.

Recent database engines support the row limiting clause syntax defined by the SQL standard:

```sql
SELECT ... OFFSET n ROWS FETCH FIRST m ROWS ONLY
```

This should be the preferred syntax to be used, if all target database types support this SELECT clause.

The ODI database drivers can convert the Informix SQL SKIP/FIRST row limiting clause to a native SQL equivalent, if the row limiting clause parameters are simple integer literals (the clause is not translated when using SQL parameters / program variables).

Important: In addition to the SKIP/FIRST clause of the projection clause, Informix SQL supports also a LIMIT clause after the ORDER BY clause:

```sql
SELECT customer.* FROM customer ... ORDER BY cust_name LIMIT 10
```

This Informix SQL syntax construction is not converted by the ODI drivers. To benefit from the conversion, review the code to use the Informix SQL SKIP/FIRST clause instead.
**PostgreSQL**

PostgreSQL supports the following row limiting clause:

```sql
SELECT ... ORDER BY ... OFFSET n ROWS FETCH FIRST m ROWS ONLY
```

**Solution**

The Informix SQL row limiting clause can be converted by the PostgreSQL driver to the native SQL equivalent clause, when the parameters are simple integer literals.

**Note:** The row limiting clause must not use SQL parameters. Only row limiting clauses using integer constants will be converted.

The translation of the Informix SQL row limiting clause can be controlled with the following FGLPROFILE entry:

```
dbi.database.dsname.ifxemul.rowlimiting = true
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

- **Row limiting clause (SELECT)** on page 650
- How to use the right clause to limit the number of rows produced by a `SELECT` statement?

**BDL programming**

PostgreSQL related programming topics.

**SQL errors on PREPARE**

**Informix®**

With Informix®, a `PREPARE` instruction returns an SQL error in case of problem:

```sql
TRY
  PREPARE stmt FROM "SELECT * FROM WHERE pk=1" -- table is missing!
CATCH
  DISPLAY "SQL ERROR:", SQLCA.SQLCODE
END TRY
```

**PostgreSQL**

The PostgreSQL database driver is implemented with the PostgreSQL libpq API. This library does not provide a way to send SQL statements to the database server during the BDL `PREPARE` instruction, like the Informix® interface does.

When preparing an SQL statement with the BDL `PREPARE` or `DECLARE` instruction, no SQL error will be returned if the SQL statement is invalid. However, an SQL error will occur after the `OPEN` / `FOREACH` / `EXECUTE` instructions.

**Solution**

Make sure your BDL programs do not test the `STATUS` or `SQLCA.SQLCODE` variable just after `PREPARE` instructions.

Change the program logic in order to handle the SQL errors when opening the cursors (`OPEN`) or when executing SQL statements (`EXECUTE`).
INSERT cursors

Informix®
Informix® provides *insert cursors* to optimize row creation in a database. An insert cursor is declared as a cursor, and rows as added with the `PUT` instruction. The rows are buffered and sent to the database server when executing a `FLUSH` instruction, or when the cursor is closed with `CLOSE`. When using transactions in Informix®, the OPEN, PUT and FLUSH instructions must be executed within a transaction block.

```
DECLARE c1 CURSOR FOR INSERT INTO tab1 ...
BEGIN WORK
  OPEN c1
  WHILE ...
    PUT c1 USING var-list
  END WHILE
CLOSE c1
COMMIT WORK
```

PostgreSQL
PostgreSQL does not support insert cursors.

Solution
Insert cursors are emulated by the database interface, using basic `INSERT` SQL instructions.
The performances might be not as good as with Informix®, but the feature is fully supported.

Related concepts
- Insert cursors on page 633
- Using insert cursors with non-Informix databases.

Cursors WITH HOLD

Informix®
Informix® closes opened cursors automatically when a transaction ends, unless the WITH HOLD option is used in the DECLARE instruction:

```
DECLARE c1 CURSOR WITH HOLD FOR SELECT ...
BEGIN WORK
  OPEN c1
  BEGIN WORK
    FETCH c1 ...
    COMMIT WORK
  FETCH c1 ...
CLOSE c1
```

PostgreSQL
With PostgreSQL, opened cursors using `SELECT` statements without a `FOR UPDATE` clause are not closed when a transaction ends. All PostgreSQL cursors are WITH HOLD cursors, unless the `FOR UPDATE` clause issued in the `SELECT` statement.

**Note:** Native PostgreSQL WITH HOLD cursors are automatically closed, if the cursor is opened inside a transaction block, and the transaction is canceled with a rollback.

BDL WITH HOLD cursors declared with a `SELECT` ... `FOR UPDATE` cannot be supported with PostgreSQL: Native holdable cursors declared for update produce the following SQL error:

```
DECLARE CURSOR WITH HOLD ... FOR UPDATE is not supported
```
Solution

For consistency with other database brands, database cursors that are not declared WITH HOLD are automatically closed, when a COMMIT WORK or ROLLBACK WORK is performed.

**Important:** Opening a WITH HOLD cursor declared with a SELECT FOR UPDATE results in an SQL error; in the same conditions, this does not normally appear with Informix®. Review the program logic in order to find another way to set locks.

Cursors declared WITH HOLD and using a SELECT without the FOR UPDATE clause can be used, as long as the cursor is opened outside a transaction block.

**Important:** Since PostgreSQL automatically closes all WITH HOLD cursors opened in a transaction which is canceled by a rollback, do not open such cursor after a BEGIN WORK, that can potentially be terminated by a ROLLBACK WORK.

The following code can be used with PostgreSQL:

```sql
MAIN
DEFINE rec RECORD
    pkey INT,
    name VARCHAR(50)
END RECORD

CONNECT TO "test1+driver='dbmpgs'" USER "pgsuser" USING "fourjs"

WHENEVER ERROR CONTINUE
DROP TABLE tab1
WHENEVER ERROR STOP
CREATE TABLE tab1 ( pkey INT, name VARCHAR(50) )
FOR rec.pkey=1 TO 10
    LET rec.name = SFMT("Item %1", rec.pkey)
    INSERT INTO tab1 VALUES ( rec.pkey, rec.name )
END FOR

DECLARE c1 CURSOR WITH HOLD FOR SELECT * FROM tab1 ORDER BY pkey

OPEN c1 -- Outside TX block!
FETCH c1 INTO rec.* DISPLAY rec.*
BEGIN WORK
FETCH c1 INTO rec.* DISPLAY rec.*
ROLLBACK WORK
BEGIN WORK
FETCH c1 INTO rec.* DISPLAY rec.*
COMMIT WORK
FETCH c1 INTO rec.* DISPLAY rec.*
BEGIN WORK
FETCH c1 INTO rec.* DISPLAY rec.*
ROLLBACK WORK
FETCH c1 INTO rec.* DISPLAY rec.*
CLOSE c1

FOREACH c1 INTO rec.*
    BEGIN WORK
        DISPLAY rec.* -- Do some real SQL here...
        IF rec.pkey MOD 2 == 0 THEN
            COMMIT WORK
        ELSE
            ROLLBACK WORK
        END IF
    END WORK
END FOREACH
```
Since PostgreSQL automatically closes `FOR UPDATE` cursors when the transaction ends, opening cursors declared `FOR UPDATE` and the `WITH HOLD` option results in an SQL error that does not normally appear with Informix® under the same conditions.

Review the program logic in order to find another way to set locks.

**Related concepts**

Cursors WITH HOLD on page 631

Programming WITH HOLD cursors using SELECT with and without `FOR UPDATE` clause.

**SELECT ... FOR UPDATE**

**Informix®**

Legacy BDL programs typically use a cursor with `SELECT FOR UPDATE` to implement pessimistic locking and avoid several users editing the same rows:

```sql
DECLARE cc CURSOR FOR
SELECT ... FROM tab WHERE ...
OPEN cc
FETCH cc <-- lock is acquired
...
CLOSE cc <-- lock is released
```

The row must be fetched in order to set the lock.

If the cursor is local to a transaction, the lock is released when the transaction ends. If the cursor is declared `WITH HOLD`, the lock is released when the cursor is closed.

Informix® provides the `SET LOCK MODE` instruction to define the lock wait timeout:

```sql
SET LOCK MODE TO { WAIT | NOT WAIT | WAIT seconds }
```

The default mode is NOT WAIT.

**PostgreSQL**

With PostgreSQL, locks are released when closing the cursor or when the transaction ends.

PostgreSQL locking granularity is at the row level.

PostgreSQL has no equivalent for `SET LOCK MODE TO NOT WAIT`.

**Solution**

The database interface is based on an emulation of an Informix® engine using transaction logging. Therefore, opening a `SELECT ... FOR UPDATE` cursor declared outside a transaction will raise an SQL error -255 (not in transaction).

You must review the program logic if you use pessimistic locking because it is based on the NOT WAIT mode which is not supported by PostgreSQL.

**Related concepts**

Cursors WITH HOLD on page 631
Programming WITH HOLD cursors using SELECT with and without FOR UPDATE clause.

**UPDATE/DELETE ... WHERE CURRENT OF**

**Informix®**

Informix® allows positioned UPDATEs and DELETEs with the "WHERE CURRENT OF cursor" clause, if the cursor has been DECLARED with a SELECT ... FOR UPDATE statement.

**PostgreSQL**

UPDATE/DELETE ... WHERE CURRENT OF is supported by PostgreSQL with server-side cursors created by a DECLARE statement.

**Solution**

With PostgreSQL, UPDATE/DELETE ... WHERE CURRENT OF instructions are executed as is without any SQL translation: Since SELECT FOR UPDATE statements are implemented with a DECLARE statement to get a server cursor, native positioned update/delete can take place in PostgreSQL.

As a replacement of WHERE CURRENT OF, if the database table is defined with a primary key column, use the value fetched from the SELECT [FOR UPDATE] cursor in the WHERE clause of the UPDATE/DELETE statement.

**Related concepts**

Positioned UPDATE/DELETE on page 631

Using positioned updates/deletes with named database cursors.

**LOAD and UNLOAD**

**Informix®**

Informix® provides two SQL instructions to export / import data from / into a database table:

The UNLOAD instruction copies rows from a database table into a text file:

```sql
UNLOAD TO "filename.unl" SELECT * FROM tab1 WHERE ..
```

The LOAD instructions insert rows from a text file into a database table:

```sql
LOAD FROM "filename.unl" INSERT INTO tab1
```

**PostgreSQL**

PostgreSQL does not provide LOAD and UNLOAD instructions.

**Solution**

LOAD and UNLOAD instruction are implemented in the Genero BDL runtime system with basic INSERT (for LOAD) or SELECT (for UNLOAD) SQL commands. The LOAD and UNLOAD instruction can be supported with various database servers.

However, LOAD and UNLOAD require the description of the column types in order to work, that can lead to some differences in the data formatting.

**Note:** If no transaction is started, the LOAD instruction will automatically execute a BEGIN WORK and COMMIT WORK when finished, or ROLLBACK WORK if a row insertion failed while loading. Terminating a transaction will automatically close cursors not defined WITH HOLD option. To workaround this situation, see more details in the LOAD on page 736 reference topic.

The LOAD and UNLOAD BDL instructions are supported with PostgreSQL.
Related concepts
LOAD and UNLOAD instructions on page 639
The LOAD and UNLOAD instructions can produce different data formats depending on the database server type.

SQL Interruption

Informix®
With Informix®, it is possible to interrupt a long running query if the SQL INTERRUPT ON option.

PostgreSQL
PostgreSQL supports SQL Interruption: The db client must issue an `PQcancel()` libPQ call to interrupt a query.

Solution
The PostgreSQL database driver supports SQL interruption and converts the SQLSTATE code 57014 to the Informix® error code -213.

Related concepts
Using SQL interruption on page 601
Interrupt long running SQL queries, or interrupt queries waiting for locked data.

Scrollable cursors

Informix®
Informix® SQL and Genero BDL support scrollable cursors when you specify the SCROLL clause in the DECLARE cursor instruction:

```sql
DECLARE c1 SCROLL CURSOR FOR SELECT ...
```

Important: Informix does not allow to fetch TEXT/BYTE columns with scrollable cursors. If you declare a scroll cursor with a SELECT containing TEXT/BYTE columns, Informix will produce the SQL error -611 when executing the OPEN instruction.

PostgreSQL
PostgreSQL supports native scrollable cursors: The client application must use the SCROLL option when declaring a server cursor.

Solution
The PostgreSQL database driver uses native scrollable cursors by declaring server cursors with the SCROLL option.

Related concepts
Scrollable cursors on page 623
How scrollable cursors can be supported on different databases.

Stored procedure calls

PostgreSQL supports stored procedures and stored functions.

Important: Stored procedures are supported since PostgreSQL version 11: Only stored functions are supported in releases prior to PostgreSQL 11.

To create a stored procedure in a PostgresQL database, use the CREATE PROCEDURE statement. Stored procedures can take IN parameter and INOUT parameters that will be returned as a result set row.

To create a stored function in a PostgresQL database, use the CREATE FUNCTION statement. Stored functions can take IN parameters and can also return more that one value when specify the returning values as function parameters.
with the \texttt{OUT} keyword. However, to be a scalar valued function that can be used in SQL expressions, a stored function must return a single value with the \texttt{RETURNS} clause. To return a result set with multiple rows, define a stored function with the \texttt{RETURNS SETOF} clause.

\textbf{Note:} Pay attention to the procedure/function signature; PostgreSQL allows overloading. For example, \texttt{func(int)} and \texttt{func(char)} are two different functions. To drop a procedure or function, you must specify the parameter type to identify signature properly.

\textit{Stored procedure with output parameters}

To execute a stored procedure with PostgreSQL, you must use \texttt{CALL procname}, as shown in this line:

\begin{verbatim}
PREPARE stmt FROM "call proc1(?,?,?)"
\end{verbatim}

In order to retrieve returning values into program variables, you must use an \texttt{INTO} clause in the \texttt{EXECUTE} instruction.

The following example shows how to call a stored procedure with PostgreSQL:

\begin{verbatim}
MAIN
  DEFINE n INTEGER
  DEFINE d DECIMAL(6,2)
  DEFINE c VARCHAR(200)
  DATABASE test1
  EXECUTE IMMEDIATE "create procedure proc1("
    "          p1 integer,"
    "          inout p2 numeric(6,2),"
    "          inout p3 varchar(200)"
  "    )"
  " as $$
  begin
    p2:= p1 + 0.23;
    p3:= 'Value = ' || cast(p1 as text);
  end;"
  " $$ language plpgsql"
PREPARE stmt FROM "call proc1(?,?,?)"
LET n = 111
EXECUTE stmt USING n, d, c INTO d, c
DISPLAY d
DISPLAY c
END MAIN
\end{verbatim}

\textit{Stored functions with result set}

With PostgreSQL, you can execute stored function returning a result set. To do so, you must declare a cursor and fetch the rows:

\begin{verbatim}
MAIN
  DEFINE i, n INTEGER
  DEFINE d DECIMAL(6,2)
  DEFINE c VARCHAR(200)
  DATABASE test1
  CREATE TABLE tab1 ( c1 INTEGER, c2 DECIMAL(6,2), c3 VARCHAR(200) )
  INSERT INTO tab1 VALUES ( 1, 123.45, 'aaaaaa' )
  INSERT INTO tab1 VALUES ( 2, 123.66, 'bbbbbbbb' )
  INSERT INTO tab1 VALUES ( 3, 444.77, 'cccccc' )
  EXECUTE IMMEDIATE "create function func2(integer)"
    " returns setof tab1"
    " as $$
    begin
      for r in (select * from tab1 where c1 > $1) loop
        p1 := r.c1;
        p2 := r.c2;
        p3 := r.c3;
      end loop;
    end;"
    " $$ language sql"
  DECLARE curs CURSOR FROM "select * from func2(?)"
\end{verbatim}
Stored function with output parameters

To execute a stored function with PostgreSQL that returns several values, you must use `SELECT * FROM function`, as shown in this line:

```
PREPARE stmt FROM "select * from procl(?)"
```

In order to retrieve returning values into program variables, you must use an `INTO` clause in the `EXECUTE` instruction.

The following example shows how to call a stored function with PostgreSQL:

```
MAIN
  DEFINE n INTEGER
  DEFINE d DECIMAL(6,2)
  DEFINE c VARCHAR(200)
  DATABASE test1
  EXECUTE IMMEDIATE "create function func1(" || "          p1 integer," || "          out p2 numeric(6,2)," || "          out p3 varchar(200)"
  " as $$
  "    p2:= p1 + 0.23;
  "    p3:= 'Value = ' || cast(p1 as text);
  "    end;
  "$ language plpgsql"
PREPARE stmt FROM "select * from func1(?)"
LET n = 111
EXECUTE stmt USING n INTO d, c
DISPLAY d
DISPLAY c
END MAIN
```

SQLite

Supported versions

Genero BDL supports the following SQLite versions:

- SQLite 3.x

Installation (Runtime Configuration)

SQLite related installation topics.

Install SQLite and create a database - database configuration/design tasks

If you are tasked with installing and configuring the database, here is a list of steps to be taken:

1. If the `dbmsqtl` database driver is linked dynamically with the SQLite library, you must install the SQLite software on your computer. However, on most platforms, the driver has an embedded version of the SQLite library, and on platforms such as Linux® and Mac OS X™, the SQLite library is usually present.

The minimum required version is SQLite 3.6.

2. Create a new SQLite database.
To create a new database with tables, start the sqlite3 command line tool and execute SQL statements:

```
$ sqlite3 /var/data/stores.db
sqlite> CREATE TABLE customer ( cust_id INT PRIMARY KEY, ... );
$ .exit
```

To create an empty database, you can also issue the following command:

```
$ sqlite3 /var/data/stores.db ""
```

or create an empty file with operating system command:

```
$ touch /var/data/stores.db
```

And empty file can also be created from a program by using a base.Channel object:

```
DEFINE ch base.Channel
LET ch = base.Channel.create()
CALL ch.openFile("/var/data/stores.db","w")
CALL ch.close
```

Related concepts

SQL programming on page 590
Covers topics about interacting with a database server using SQL.

Prepare the runtime environment - connecting to the database

1. In order to connect to SQLite, you must have a database driver "dbmsqt" in $FGLDIR/dbdrivers. On most platforms, the SQLite driver is linked statically with the SQLite library, in other word SQLite is embedded in the ODI driver. However, on some platforms such as Linux® and Max OS X, where the SQLite library is usually present.

2. Make sure that the SQLite environment variables are properly set. You may want to define an environment variable such as SQLITEDIR to hold the installation directory of SQLite, which can then be used to set PATH and LD_LIBRARY_PATH. See SQLite documentation for more details.

3. If the SQLite library is not embedded in the dbmsqt* driver, the environment must be set to find the SQLite library. Verify the environment variable defining the search path for the SQLite shared library.

4. Make sure that all operating system users running the application have read/write access to the database file.

5. SQLite databases use UTF-8 encoding. If the locale used by the runtime system (LANG/LC_ALL) is not UTF-8 (for example, fr_FR.iso88591), Genero will do the appropriate character set conversions.

6. Set up the FGLPROFILE entries for database connections.
   a) Define the SQLite database driver:

```
   dbi.database.dbname.driver = "dbmsqt"
```
b) The "source" parameter defines the path to the SQLite database file. Note that the database file must reside
on the local disk (SQLite does not support network file systems). SQLite also supports in-memory database
creation with the :memory: db specification. See SQLite documentation (sqlite3_open) for more details.

```
  dbi.database.dbname.source = "'/opt/myapp/stock.dbs"
```

c) If the "source" parameter defines a relative path or a simple file name and the SQLite database file does not
reside in that location based on the current directory of the fglrun process, define the DBPATH environment
variable to find the database file. See DBPATH documentation for more details about this environment
variable.

```
  DBPATH="/opt/myapp"
```

Related concepts

The FGLPROFILE file(s) on page 255
FGLPROFILE environment variable defines Genero BDL configuration files

Database concepts
SQLite related database concepts topics.

Database concepts
Informix® servers can handle multiple database entities, while SQLite can manage several database files.

Tip: If you have several Informix database entities, migrating from the Informix database to another database it is a
good opportunity to centralize all tables in a single database. To avoid conflicts with table names, use a prefix when
needed.

Solution

Map each Informix® database to a SQLite database file.
Consider creating the SQLite database file before using the connection instruction. The database file can be created as
an empty file, with a OS shell command (touch) or by program by using the file utility classes.

It is possible to specify an SQLite database file name in the database specification in CONNECT TO or DATABASE
instructions:

```
  DATABASE "/opt/myapp/database/stock1.dbs"
```

However, it is recommended to use an indirection by providing an abstract name identifier in the program, and by
defining the real database file with the "source" connection parameter. The file defined by "source" is then found
directly (can be a relative or absolute path), or by using DBPATH settings, if not found from the current directory of
fglrun (when it's not an absolute path).

In the program:

```
  DATABASE stock
```

In the FGLPROFILE configuration file, define the SQLite driver and the database file:

```
  dbi.database.stock.driver = "dbmsqt"
  dbi.database.stock.source = "/opt/myapp/database/stock1.dbf"
```

Or if in FGLPROFILE, you define the file name only:

```
  dbi.database.stock.source = "stock1.dbf"
```
The file is found by using DBPATH:

```
DBPATH="/opt/myapp/database"
```

When specifying :memory: as database file name, an empty SQLite database is created in memory. This can be useful if the persistence of the data is not required after the program has terminated:

```
DATABASE ":memory:"
```

**Concurrency management**

Informix® is a multiuser database engine, while SQLite is typically used for a single-user application. SQLite 3 supports multiuser access to the same database file, but it is not designed for large multiuser applications.

SQLite 3 supports two isolation levels: **SERIALIZABLE** (the default), and **READ UNCOMMITTED**. The isolation level can be changed with the **PRAGMA** command.

By default in the **SERIALIZABLE** isolation level, SQLite will raise an SQL error if a program tries to access a database resource in use by another program. To avoid the SQL error and force programs to wait for each other, programs define the behavior when the SQLite database is busy (SQLITE_BUSY), with a specific API call. No SQL command exists for this.

**Solution**

We recommend that you use SQLite for single-user DB applications. If several programs must access the same SQLite database, each program must perform a **SET LOCK MODE TO WAIT** instruction after the connection: **SET LOCK MODE** will be mapped to a call to the sqlite3_busy_timeout() SQLite API function to get the same behavior as Informix®, while **SET ISOLATION** instructions will be ignored.

**Related concepts**

- Concurrent data access on page 599
- Understanding concurrent data access and data consistency.
- Optimistic locking on page 624
  Implementing optimistic locking to handle access concurrently to the same database records.
- Cursors WITH HOLD on page 631
  Programming WITH HOLD cursors using **SELECT** with and without **FOR UPDATE** clause.

**Transactions handling**

**Informix®**

With the Informix® native mode (non ANSI):

- Transactions blocks start with **BEGIN WORK** and terminate with **COMMIT WORK** or **ROLLBACK WORK**.
- Statements executed outside a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.

```
UPDATE tab1 SET ... -- auto-committed
BEGIN WORK        -- start of TX block
UPDATE tab1 SET ...  
UPDATE tab2 SET ...  
...                
COMMIT WORK        -- end of TX block
```

Informix® version 11.50 introduces savepoints:

```
SAVEPOINT name [UNIQUE]
ROLLBACK [WORK] TO SAVEPOINT [name] 
RELEASE SAVEPOINT name
```
SQLite

With SQLite:

- Individual SQL statements are auto-committed.
- Transactions start with `BEGIN TRANSACTION` and end with `COMMIT TRANSACTION` or `ROLLBACK TRANSACTION`.
- DDL statements can be executed (and canceled) in transaction blocks.

SQLite supports savepoints with some differences compared to Informix®:

- `SAVEPOINT` can be used instead of `BEGIN TRANSACTION`. In this case, `RELEASE` is like a `COMMIT`.
- The syntax of a rollback to the savepoint is `ROLLBACK [TRANSACTION] TO [SAVEPOINT] name`.
- The syntax of a release of the savepoint is `RELEASE [SAVEPOINT] name`.
- Rollback must always specify the savepoint name.
- You cannot rollback to a savepoint if cursors are opened.
- In SQLite versions prior to 3.7, you cannot rollback are transaction if a cursor is open.

Solution

Regarding transaction control instructions, BDL applications do not have to be modified in order to work with SQLite. The `BEGIN WORK`, `COMMIT WORK` and `ROLLBACK WORK` commands are translated the native commands of SQLite.

Note: If you want to use savepoints, always specify the savepoint name in `ROLLBACK TO SAVEPOINT` and do not open cursors during transactions using savepoints. If you are using an SQLite versions prior to 3.7, it is not possible to perform a `ROLLBACK WORK` if a cursor (with hold) is currently open.

See also `SELECT FOR UPDATE`

Related concepts

Database transactions on page 600
Database transactions define a set of SQL instructions to be executed as a whole, or rolled back as a whole.

Database users

Informix®

Until version 11.70.xC2, Informix® database users must be created at the operating system level and must be members of the 'informix' group.

Starting with 11.70.xC2, Informix® supports database-only users with the `CREATE USER` instruction, as in most other db servers.

Any database user must have sufficient privileges to connect and use resources of the database; user rights are defined with the `GRANT` command.

SQLite

SQLite does not have the database users concept.

However, the operating system user must have read/write access to the database file.

Solution

SQLite is mainly designed for single-user applications.

Related concepts

Database users and security on page 608
Properly identifying database users allows to use database security and audit features.

**Data dictionary**  
SQLite related data dictionary topics.  

**Data type conversion table: Informix to SQLite**

Table 274: Data type conversion table between Informix® and SQLite

<table>
<thead>
<tr>
<th>Informix® data types</th>
<th>SQLite data types</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR(n)</td>
<td>CHAR(n) COLLATE RTRIM</td>
</tr>
<tr>
<td>VARCHAR(n[,m])</td>
<td>VARCHAR(n) COLLATE RTRIM</td>
</tr>
<tr>
<td>LVARCHAR(n)</td>
<td>VARCHAR(n) COLLATE RTRIM</td>
</tr>
<tr>
<td>NCHAR(n)</td>
<td>NCHAR(n)</td>
</tr>
<tr>
<td>NVARCHAR(n)</td>
<td>NVARCHAR(n)</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>BIGINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>INT8</td>
<td>BIGINT</td>
</tr>
<tr>
<td>SERIAL[(start)]</td>
<td>INTEGER (see note 1)</td>
</tr>
<tr>
<td>BIGSERIAL[(start)]</td>
<td>N/A (see note 1)</td>
</tr>
<tr>
<td>INT8[(start)]</td>
<td>N/A (see note 1)</td>
</tr>
<tr>
<td>DOUBLE PRECISION / FLOAT[(n)]</td>
<td>FLOAT</td>
</tr>
<tr>
<td>REAL / SMALLFLOAT</td>
<td>SMALLFLOAT</td>
</tr>
<tr>
<td>DECIMAL(p,s)</td>
<td>DECIMAL (p, s)</td>
</tr>
<tr>
<td>DECIMAL(p)</td>
<td>DECIMAL (p, s)</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>MONEY(p,s)</td>
<td>DECIMAL (p, s)</td>
</tr>
<tr>
<td>MONEY(p)</td>
<td>DECIMAL (p, 2)</td>
</tr>
<tr>
<td>MONEY</td>
<td>DECIMAL (16, 2)</td>
</tr>
<tr>
<td>TEXT</td>
<td>TEXT</td>
</tr>
<tr>
<td>BYTE</td>
<td>BLOB</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME HOUR TO HOUR</td>
<td>SMALLTIME</td>
</tr>
<tr>
<td>DATETIME HOUR TO MINUTE</td>
<td>SMALLTIME</td>
</tr>
<tr>
<td>DATETIME HOUR TO SECOND</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME HOUR TO FRACTION(n)</td>
<td>TIME(n)</td>
</tr>
<tr>
<td>DATETIME MINUTE TO MINUTE</td>
<td>SMALLTIME</td>
</tr>
<tr>
<td>DATETIME MINUTE TO SECOND</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME MINUTE TO FRACTION(n)</td>
<td>TIME(n)</td>
</tr>
</tbody>
</table>
### Informix® data types vs. SQLite data types

<table>
<thead>
<tr>
<th>Informix® data types</th>
<th>SQLite data types</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATETIME SECOND TO SECOND</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME SECOND TO FRACTION(n)</td>
<td>TIME(n)</td>
</tr>
<tr>
<td>DATETIME FRACTION TO FRACTION(n)</td>
<td>TIME(n)</td>
</tr>
<tr>
<td>DATETIME YEAR TO YEAR</td>
<td>TINYDATETIME</td>
</tr>
<tr>
<td>DATETIME YEAR TO MONTH</td>
<td>TINYDATETIME</td>
</tr>
<tr>
<td>DATETIME YEAR TO DAY</td>
<td>TINYDATETIME</td>
</tr>
<tr>
<td>DATETIME YEAR TO HOUR</td>
<td>SMALLDATETIME</td>
</tr>
<tr>
<td>DATETIME YEAR TO MINUTE</td>
<td>SMALLDATETIME</td>
</tr>
<tr>
<td>DATETIME YEAR TO SECOND</td>
<td>DATETIME</td>
</tr>
<tr>
<td>DATETIME YEAR TO FRACTION(n)</td>
<td>DATETIME(n)</td>
</tr>
<tr>
<td>DATETIME MONTH TO MONTH</td>
<td>TINYDATETIME</td>
</tr>
<tr>
<td>DATETIME MONTH TO DAY</td>
<td>TINYDATETIME</td>
</tr>
<tr>
<td>DATETIME MONTH TO HOUR</td>
<td>SMALLDATETIME</td>
</tr>
<tr>
<td>DATETIME MONTH TO MINUTE</td>
<td>SMALLDATETIME</td>
</tr>
<tr>
<td>DATETIME MONTH TO SECOND</td>
<td>DATETIME</td>
</tr>
<tr>
<td>DATETIME MONTH TO FRACTION(n)</td>
<td>DATETIME(n)</td>
</tr>
<tr>
<td>DATETIME DAY TO DAY</td>
<td>TINYDATETIME</td>
</tr>
<tr>
<td>DATETIME DAY TO HOUR</td>
<td>SMALLDATETIME</td>
</tr>
<tr>
<td>DATETIME DAY TO MINUTE</td>
<td>SMALLDATETIME</td>
</tr>
<tr>
<td>DATETIME DAY TO SECOND</td>
<td>DATETIME</td>
</tr>
<tr>
<td>DATETIME DAY TO FRACTION(n)</td>
<td>DATETIME(n)</td>
</tr>
<tr>
<td>INTERVAL q1 TO q2</td>
<td>CHAR(50)</td>
</tr>
</tbody>
</table>

#### Notes:

1. For more details about serial emulation, see [SERIAL and BIGSERIAL data types](#) on page 1016.

#### BOOLEAN data type

**Informix®**

Informix® supports the BOOLEAN data type, which can store 't' or 'f' values.

Genero BDL implements the BOOLEAN data type in a different way: A BOOLEAN variable stores integer values 1 or 0 (for TRUE or FALSE). This type is designed to hold the result of a boolean expression.

---

**SQLite**

SQLite does not implement a native BOOLEAN type, but accepts BOOLEAN in the SQL syntax and uses integer values 1/0 for true/false booleans.

---

**Solution**

The SQLite database interface supports the BOOLEAN data type and stores 1 or 0 values in the column.
The BOOLEAN type translation can be controlled with the following FGLPROFILE entry:

```
dbi.database dsname ifxemul datatype boolean = \ l true \ l false \ l
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

*Using portable data types* on page 614

Only a limited set of data types are really portable across several database engines.

### CHAR and VARCHAR data types

**Informix**

Informix® supports the following character data types:

**Table 275: Informix® character data types**

<table>
<thead>
<tr>
<th>Informix® data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR (n)</td>
<td>SBCS and MBCS character data (max is 32767 bytes)</td>
</tr>
<tr>
<td>VARCHAR (n,m)</td>
<td>SBCS and MBCS character data (max is 255 bytes)</td>
</tr>
<tr>
<td>NCHAR (n)</td>
<td>Same as CHAR, with specific collation order</td>
</tr>
<tr>
<td>NVARCHAR (n,m)</td>
<td>Same as VARCHAR, with specific collation order</td>
</tr>
<tr>
<td>LVARCHAR (n)</td>
<td>max size varies depending on the IDS version</td>
</tr>
</tbody>
</table>

With Informix®, both CHAR/VARCHAR and NCHAR/NVARCHAR data types can be used to store single-byte or multibyte encoded character strings. The only difference between CHAR/VARCHAR and NCHAR/NVARCHAR is in how they use sorting: N[VAR]CHAR types use the collation order, while [VAR]CHAR types use the byte order.

The character set used to store strings in CHAR/VARCHAR/NCHAR/NVARCHAR columns is defined by the DB_LOCALE environment variable.

The character set used by applications is defined by the CLIENT_LOCALE environment variable.

Informix® uses Byte Length Semantics (the size N that you specify in [VAR]CHAR (N) is expressed in bytes, not characters as in some other databases)

**SQLite**

SQLite 3 provides the TEXT native data type with no strict size limitation. SQLite allows the CHAR (n), VARCHAR (n), NCHAR (n) and NVARCHAR (n) type names to be used, but actually stores the data in a TEXT native type.

SQLite treats empty strings as NOT NULL values like Informix®.

**Note:** With the default BINARY collation, SQLite compares VARCHAR and CHAR values by taking trailing blanks into account. Informix® always ignores trailing blanks when comparing CHAR/VARCHAR values.

SQLite supports only the UTF-8 character encoding. Thus, client applications must provide UTF-8 encoded strings.

**Solution**

The database interface supports character string variables in SQL statements for input (BDL USING) and output (BDL INTO).

**Important:** With the default BINARY collation, CHAR and VARCHAR comparison in SQLite takes trailing blanks into account. As result, some queries returning rows with Informix® may not return the same result set with SQLite. When creating a table in SQLite, you can change the default collation rule to force the database engine to trim trailing
blanks before comparing CHAR/VARCHAR values, by specifying COLLATION RTRIM in the column definitions. When creating a table from a Genero program, if Informix® emulation is enabled for the CHAR/VARCHAR types, the SQLite database driver adds automatically COLLATE RTRIM after the CHAR(N) or VARCHAR(N) type, to get the same comparison semantics as Informix®.

Regarding character sets, the SQLite database driver automatically converts character strings used in the programs to/from UTF-8 for SQLite.

SQLite uses character length semantics: When you define a CHAR(20) and the database character set is multibyte, the column can hold more bytes/characters than the Informix® CHAR(20) type, when using byte length semantics.

When using a multibyte character set (such as UTF-8), define database columns with the size in character units, and use character length semantics in BDL programs with FGL_LENGTH_SEMANTICS=CHAR.

When extracting a database schema from a SQLite database, the fgldbsch schema extractor uses the size of the column in characters, not the octet length. If you have created a CHAR(10 (characters)) column a in SQLite database using the UTF-8 character set, the .sch file will get a size of 10, that will be interpreted following FGL_LENGTH_SEMANTICS, as a number of bytes or characters.

See also the section about Localization.

The CHAR/VARCHAR type translation can be controlled with the following FGLPROFILE entries:

```
dbi.database.dsname.ifxemul.datatype.char = { true | false }
dbi.database.dsname.ifxemul.datatype.varchar = { true | false }
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

Related concepts
CHAR and VARCHAR types on page 616
Using the CHAR and VARCHAR data types with different databases.

Numeric data types

Informix®

Informix® supports several data types to store numbers:

Table 276: Informix® numeric data types

<table>
<thead>
<tr>
<th>Informix® data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>16 bit signed integer</td>
</tr>
<tr>
<td>INTEGER</td>
<td>32 bit signed integer</td>
</tr>
<tr>
<td>BIGINT</td>
<td>64 bit signed integer</td>
</tr>
<tr>
<td>INT8</td>
<td>64 bit signed integer (replaced by BIGINT)</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>Equivalent to DECIMAL(16)</td>
</tr>
<tr>
<td>DECIMAL(p)</td>
<td>Floating-point decimal number (max precision is 32)</td>
</tr>
<tr>
<td>DECIMAL(p,s)</td>
<td>Fixed-point decimal number (max precision is 32)</td>
</tr>
<tr>
<td>MONEY</td>
<td>Equivalent to DECIMAL(16,2)</td>
</tr>
<tr>
<td>MONEY(p)</td>
<td>Equivalent to DECIMAL(p,2) (max precision is 32)</td>
</tr>
<tr>
<td>MONEY(p,s)</td>
<td>Equivalent to DECIMAL(p,s) (max precision is 32)</td>
</tr>
<tr>
<td>REAL / SMALLFLOAT</td>
<td>32-bit floating point decimal (C float)</td>
</tr>
<tr>
<td>DOUBLE PRECISION / FLOAT[(n)]</td>
<td>64-bit floating point decimal (C double)</td>
</tr>
</tbody>
</table>
SQLite

SQLite 3 supports INTEGER (8 byte integer) and REAL (8 byte floating point) as native types to store numbers, but allows also synonyms:

Table 277: SQLite numeric data types and supported synonyms

<table>
<thead>
<tr>
<th>Supported synonyms</th>
<th>SQLite type affinity</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT, INTEGER, TINYINT, SMALLINT, MEDIUMINT, BIGINT, UNSIGNED BIG INT, INT2, INT8</td>
<td>INTEGER (8 bytes!)</td>
</tr>
<tr>
<td>REAL, DOUBLE, DOUBLE PRECISION, FLOAT</td>
<td>REAL (8 bytes!)</td>
</tr>
<tr>
<td>DECIMAL(p,s), NUMERIC</td>
<td>NUMERIC (based on REAL)</td>
</tr>
</tbody>
</table>

Important: Exact decimal types like DECIMAL(p,s) may be stored as floating point numbers (REAL, INTEGER or TEXT types, according to the type affinity selected by SQLite. When converted to floating point type, data loss and rounding rule differences are possible with SQLite.

Solution

Informix® numeric types are not translated by the SQLite database driver: The numeric types are used as is when creating tables, since SQLite supports a wide range of type synonyms.

Since SQLite 3 does not have exact decimal types like DECIMAL(p,s), you must pay attention to the rounding rules and data loss when using numbers with many significant digits. Arithmetic operations like division have different results than with Informix®. It is better to fetch the original column value into a DECIMAL variable, and do arithmetic operations in the application program.

The numeric types translation can be controlled with the following FGLPROFILE entries:

```sql
  dbi.database.dsnname.ifxemul.datatype.smallint = 1 true false 1
  dbi.database.dsnname.ifxemul.datatype.integer = 1 true false 1
  dbi.database.dsnname.ifxemul.datatype.bigint = 1 true false 1
  dbi.database.dsnname.ifxemul.datatype.int8 = 1 true false 1
  dbi.database.dsnname.ifxemul.datatype.decimal = 1 true false 1
  dbi.database.dsnname.ifxemul.datatype.money = 1 true false 1
  dbi.database.dsnname.ifxemul.datatype.float = 1 true false 1
  dbi.database.dsnname.ifxemul.datatype.smallfloat = 1 true false 1
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

Related concepts

Using portable data types on page 614

Only a limited set of data types are really portable across several database engines.

DATE and DATETIME data types

Informix®

Informix® provides two data types to store date and time information:

- **DATE** = for year, month and day storage.
- **DATETIME** = for year to fraction (1-5) storage.

The **DATE** type is stored as an **INTEGER** with the number of days since 1899/12/31.

The **DATETIME** type can be defined with various time units, by specifying a start and end qualifier. For example, you can define a datetime to store an hour-to-second value with **DATETIME HOUR TO SECOND**.
The values of Informix® DATETIME can be represented with a character string literal, or as DATETIME() literals:

'2017-12-24 15:45:12.345'  -- a DATETIME YEAR TO FRACTION(3)
'15:45'   -- a DATETIME HOUR TO MINUTE
DATETIME(2017-12-24 12:45) YEAR TO MINUTE
DATETIME(12:45:56.333) HOUR TO FRACTION(3)

Informix® is able to convert quoted strings to DATE / DATETIME data, if the string contains matching environment parameters. The string to date conversion rules for DATE are defined by the DBDATE environment variable. The string to datetime format for DATETIME is defined by the GL_DATETIME environment variable.

**Note:** Within Genero programs, the string representation for DATETIME values is always ISO (YYYY-MM-DD hh:mm:ss.ffffff)

Informix® supports date arithmetic on DATE and DATETIME values. The result of an arithmetic expression involving dates/times is an INTEGER number of days when only DATE values are used, and an INTERVAL value if a DATETIME is used in the expression.

Informix® automatically converts an INTEGER to a DATE when the integer is used to set a value of a date column.

Informix® provides the CURRENT q1 TO q2 operator, to get the system date/time on the server where the current database is located. When no qualifiers are specified, CURRENT returns a DATETIME YEAR TO FRACTION(3). Informix also supports the SYSDATE operator, which returns the current system time as a DATETIME YEAR TO FRACTION(5).

**Note:** The USEOSTIME configuration parameter must be set to 1 in order to get the subsecond precision in CURRENT and SYSDATE operators. See Informix documentation for more details.

**SQLite**

SQLite 3 does not have a native type for date/time storage, but you can use data/time type names and functions based on the string representation of dates and times. The date/time values are stored in the TEXT native type.

The date/time functions of SQLite are based on standard DATE (YYYY-MM-DD), TIME (hh:mm:ss) and TIMESTAMP (YYYY-MM-DD hh:mm:ss) concepts.

For maximum flexibility with other RDBMS SQL languages, SQLite allows you to define table columns with your own type names. You can for example use the SMALLDATETIME, SMALLTIME, TIME(N), DATETIME(N) type names.

**Solution**

All Informix® - BDL date/time types can be stored in SQLite date/time columns.

Since SQLite allows various data type names, the date/time type conversion rules define specific type names such as SMALLTIME, TINYDATETIME, to map original Informix® date/time types. This allows the SQLite ODI driver and the fglombsch tool detect the exact date/time type of a column. When a CREATE TABLE statement in a BDL program uses DATETIME HOUR TO MINUTE, it is mapped to a SMALLTIME by the ODI driver, and when extracting the database schema, fglombsch can recognized SMALLTIME as a BDL / Informix® DATETIME HOUR TO MINUTE column.

The storage format must follow the ISO date/time formatting style (YYYY-MM-DD hh:mm:ss.ffffff). Depending on the BDL date/time precision, some parts will be omitted. For example a DATETIME HOUR TO MINUTE is stored as hh:mm (see conversion table below for more details).

Use the following conversion rules to map Informix® date/time types to SQLite date/time (pseudo) types:

**Table 278: Informix® data types and SQLite equivalents**

<table>
<thead>
<tr>
<th>Informix® data type</th>
<th>SQLite (pseudo data type)</th>
<th>Storage format</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>DATE</td>
<td>YYYY-MM-DD</td>
</tr>
<tr>
<td>Informix® data type</td>
<td>SQLite (pseudo data type)</td>
<td>Storage format</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>---------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>DATETIME HOUR TO HOUR</td>
<td>SMALLTIME</td>
<td>hh:00</td>
</tr>
<tr>
<td>DATETIME HOUR TO MINUTE</td>
<td>SMALLTIME</td>
<td>hh:mm</td>
</tr>
<tr>
<td>DATETIME HOUR TO SECOND</td>
<td>TIME</td>
<td>hh:mm:ss</td>
</tr>
<tr>
<td>DATETIME HOUR TO FRACTION(n)</td>
<td>TIME(n)</td>
<td>hh:mm:ssffffff</td>
</tr>
<tr>
<td>DATETIME MINUTE TO MINUTE</td>
<td>SMALLTIME</td>
<td>00:mm</td>
</tr>
<tr>
<td>DATETIME MINUTE TO SECOND</td>
<td>TIME</td>
<td>00:mm:ss</td>
</tr>
<tr>
<td>DATETIME MINUTE TO FRACTION(n)</td>
<td>TIME(n)</td>
<td>00:mm:ssffffff</td>
</tr>
<tr>
<td>DATETIME SECOND TO SECOND</td>
<td>TIME</td>
<td>00:00:ss</td>
</tr>
<tr>
<td>DATETIME SECOND TO FRACTION(n)</td>
<td>TIME(n)</td>
<td>00:00:ssffffff</td>
</tr>
<tr>
<td>DATETIME FRACTION TO FRACTION(n)</td>
<td>TIME(n)</td>
<td>00:00:00:ffffff</td>
</tr>
<tr>
<td>DATETIME YEAR TO YEAR</td>
<td>TINYDATETIME</td>
<td>YYYY-01-01</td>
</tr>
<tr>
<td>DATETIME YEAR TO MONTH</td>
<td>TINYDATETIME</td>
<td>YYYY-MM-01</td>
</tr>
<tr>
<td>DATETIME YEAR TO DAY</td>
<td>TINYDATETIME</td>
<td>YYYY-MM-DD</td>
</tr>
<tr>
<td>DATETIME YEAR TO HOUR</td>
<td>SMALLDATETIME</td>
<td>YYYY-MM-DD hh:00</td>
</tr>
<tr>
<td>DATETIME YEAR TO MINUTE</td>
<td>SMALLDATETIME</td>
<td>YYYY-MM-DD hh:mm</td>
</tr>
<tr>
<td>DATETIME YEAR TO SECOND</td>
<td>DATETIME</td>
<td>YYYY-MM-DD hh:mm:ss</td>
</tr>
<tr>
<td>DATETIME YEAR TO FRACTION(n)</td>
<td>DATETIME(n)</td>
<td>YYYY-MM-DD hh:mm:ssffffff</td>
</tr>
<tr>
<td>DATETIME MONTH TO MONTH</td>
<td>TINYDATETIME</td>
<td>1900-MM-01</td>
</tr>
<tr>
<td>DATETIME MONTH TO DAY</td>
<td>TINYDATETIME</td>
<td>1900-MM-DD</td>
</tr>
<tr>
<td>DATETIME MONTH TO HOUR</td>
<td>SMALLDATETIME</td>
<td>1900-MM-DD hh:00</td>
</tr>
<tr>
<td>DATETIME MONTH TO MINUTE</td>
<td>SMALLDATETIME</td>
<td>1900-MM-DD hh:mm</td>
</tr>
<tr>
<td>DATETIME MONTH TO SECOND</td>
<td>DATETIME</td>
<td>1900-MM-DD hh:mm:ss</td>
</tr>
<tr>
<td>DATETIME MONTH TO FRACTION(n)</td>
<td>DATETIME(n)</td>
<td>1900-MM-DD hh:mm:ssffffff</td>
</tr>
<tr>
<td>DATETIME DAY TO DAY</td>
<td>TINYDATETIME</td>
<td>1900-01-DD</td>
</tr>
<tr>
<td>DATETIME DAY TO HOUR</td>
<td>SMALLDATETIME</td>
<td>1900-01-DD hh:00</td>
</tr>
<tr>
<td>DATETIME DAY TO MINUTE</td>
<td>SMALLDATETIME</td>
<td>1900-01-DD hh:mm</td>
</tr>
<tr>
<td>DATETIME DAY TO SECOND</td>
<td>DATETIME</td>
<td>1900-01-DD hh:mm:ss</td>
</tr>
<tr>
<td>DATETIME DAY TO FRACTION(n)</td>
<td>DATETIME(n)</td>
<td>1900-01-DD hh:mm:ssffffff</td>
</tr>
</tbody>
</table>

The `DATE` and `DATETIME` types translation can be controlled with the following FGLPROFILE entries:

```sql
dbi.database.dsnname.ifxemul.datatype.date = [true | false ]
dbi.database.dsnname.ifxemul.datatype.datetime = [true | false ]
```
For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

In SQL statements, CURRENT expressions are converted to SQLite `strftime('%Y-%m-%d %H:%M:%S', 'now')`. The SQLite `now` option returns the current date/time in UTC, while the FGL runtime system CURRENT instruction returns the current local time. Both values can be different. Always consider using SQL parameters with program variables assigned by the CURRENT instruction of Genero BDL, instead of using CURRENT instructions in SQL statements.

**Related concepts**

- [Date/time literals in SQL statements](#) on page 635
  Good practices for date and time handling in SQL.
- [SQL LOAD and UNLOAD](#) on page 736
  Describes the instructions to export/import information from/to a database.

**INTERVAL data type**

**Informix®**

Informix® provides the INTERVAL data type to store a value that represents a span of time.

INTERVAL types are divided into two classes:

- **year-month** intervals. For example: `INTERVAL YEAR(5) TO MONTH`
- **day-time** intervals. For example: `INTERVAL DAY(9) TO SECOND`

INTERVAL columns can be defined with various time units, by specifying a start and end qualifier. For example, you can define an interval to store a number of hours and minutes with `INTERVAL HOUR(n) TO MINUTE`, where `n` defines the maximum number of digits for the hours unit.

The values of Informix® INTERVAL can be represented with a character string literal, or as `INTERVAL()` literals:

```
'1983-12-12:12:12.12' -- an INTERVAL DAY(6) TO FRACTION(3)
'7623-11'    -- an INTERVAL YEAR(9) TO MONTH
INTERVAL(18734:45) HOUR(5) TO MINUTE
INTERVAL(-7634-11) YEAR(5) TO MONTH
```

**SQLite**

SQLite 3 does not provide a data type similar to Informix® INTERVAL.

**Solution**

The INTERVAL data type and values are converted `CHAR(50)` column with SQLite.

INTERVAL values can be stored and retrieved from the database. However, since SQLite does not support a native interval type, arithmetics cannot be performed on the database side in SQL statements.

**Related concepts**

- [Using portable data types](#) on page 614
  Only a limited set of data types are really portable across several database engines.

**SERIAL and BIGSERIAL data types**

**Informix®**

Informix® supports the SERIAL, BIGSERIAL data types to produce automatic integer sequences:

- SERIAL can produce 32 bit integers (INTEGER)
- BIGSERIAL can produced 64 bit integers (BIGINT)
- SERIAL8 is a synonym for BIGSERIAL
Steps to use serials with Informix®:

1. Create the table with a column using SERIAL, or BIGSERIAL.
2. To generate a new serial, no value or a zero value is specified in the INSERT statement:
   
   ```sql
   INSERT INTO tab1 ( c ) VALUES ( 'aa' )
   INSERT INTO tab1 ( k, c ) VALUES ( 0, 'aa' )
   ```

3. After INSERT, the new value of a SERIAL column is provided in SQLCA.SQLERRD[2], while the new value of a BIGSERIAL value must be fetched with a SELECT dbinfo('bigserial') query.

Informix® allows you to insert rows with a value different from zero for a serial column. Using an explicit value will automatically increment the internal serial counter, to avoid conflicts with future INSERT statements that are using a zero value:

   ```sql
   CREATE TABLE tab ( k SERIAL); -- internal counter = 0
   INSERT INTO tab VALUES ( 0 ); -- internal counter = 1
   INSERT INTO tab VALUES ( 10 ); -- internal counter = 10
   INSERT INTO tab VALUES ( 0 ); -- internal counter = 11
   DELETE FROM tab; -- internal counter = 11
   INSERT INTO tab VALUES ( 0 ); -- internal counter = 12
   ```

SQLite

SQLite supports the AUTOINCREMENT attribute for columns:

- Only one column must be declared as INTEGER PRIMARY KEY AUTOINCREMENT.
- SQLite (version 3.8.3.1) does not support SEQUENCE objects.
- SQLite (version 3.8.3.1) does not allow AUTOINCREMENT on BIGINT columns.
- To get the last generated number, SQLite provides the sqlite_sequence table:

   ```sql
   SELECT seq FROM sqlite_sequence WHERE name='tabname'
   ```

   At INSERT, for the auto-incremented column:

   - When specifying a zero, SQLite will not generate a new sequence like Informix® does.
   - When specifying a NULL, SQLite generates a new sequence; Informix® denies NULLs in serials.
   - When specifying a value different from zero and NULL, SQLite will use that value. The next INSERT statement not providing a value > 0 will produce a new auto-incremented value that is greater as the last inserted value.

Solution

Note: For best SQL portability when using different types of databases, consider using sequences as described in Solution 3: Use native SEQUENCE database objects on page 628.

When using SQLite, the SERIAL data type is converted to INTEGER PRIMARY KEY AUTOINCREMENT.

The serial types emulation can be enabled or disabled with the following FGLPROFILE entries:

```ini
dbi.database.dbname.ifxemul.datatype.serial = [true|false]
dbi.database.dbname.ifxemul.datatype.serial8 = [true|false]
dbi.database.dbname.ifxemul.datatype.bigserial = [true|false]
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.
Disabling automatic serial retrieval for SQLCA.SQLERRD[2]

SERIAL emulation can be totally disabled by setting the ifxemul.datatype.serial FGLPROFILE entry to false:

```
dbi.database.dbname.ifxemul.datatype.serial = false
```

For Informix® compatibility, after an INSERT statement, the ODI drivers automatically execute another SQL query (or do a DB client API call when possible), to get the last generated serial, and fill the SQLCA.SQLERRD[2] register. This results in some overhead that can be avoided, if the SQLCA.SQLERRD[2] register is not used by the program.

When SERIAL emulation is required (to create temp tables with a serial column during program execution), and the SQLCA.SQLERRD[2] register does not need to be filled, (typically because you use your own method to retrieve the last generated serial), you can set the ifxemul.datatype.serial.sqlerrd2 FGLPROFILE entry to false. This will avoid the automatic retrieval of last serial value to fill SQLCA.SQLERRD[2]:

```
dbi.database.dbname.ifxemul.datatype.serial.sqlerrd2 = false
```

See also db_get_last_serial() on page 2308.

Using the native serial emulation (only option)

The SQLCA.SQLERRD[2] register is filled automatically after each INSERT with the last generated number, by fetching the value from the sqlite_sequence table.

**Important:** SQLite (V 3.8) does not support auto-incremented BIGINT columns. Therefore, BIGSERIAL or SERIAL8 cannot be supported. These Informix SQL types are converted to BIGINT PRIMARY KEY AUTOINCREMENT, but that will produce an SQL error "AUTOINCREMENT is only allowed on an INTEGER PRIMARY KEY".

Because SQLite does not behave like Informix® regarding zero and NULL value specification for auto-incremented columns, all INSERT statements must be reviewed to remove the SERIAL column from the list.

For example, the following statement:

```
INSERT INTO tab (col1,col2) VALUES ( 0 , p_value)
```

Can be converted to:

```
INSERT INTO tab (col2) VALUES (p_value)
```

Static SQL INSERT using records defined from the schema file must also be reviewed:

```
DEFINE rec LIKE tab.*
INSERT INTO tab VALUES (rec.*) -- will use the serial column
```

Can be converted to:

```
INSERT INTO tab VALUES rec.* -- without parentheses, serial column is removed
```

Related concepts

Auto-incremented columns (serials) on page 626
How to implement automatic record keys.

**ROWID columns**

**Informix®**

When creating a table, Informix® automatically adds a ROWID integer column (applies to non-fragmented tables only).

The **ROWID** column is auto-filled with a unique number and can be used like a primary key to access a given row.

**Note:** Informix® **ROWID** usage was a common practice in the early days of Informix® 4GL programming. Today it is recommended to define all your database tables with a **PRIMARY KEY** to uniquely identify rows.

With Informix®, the SQLCA.SQLERRD[6] register contains the **ROWID** of the last row affected by an INSERT, UPDATE or DELETE statement.

**SQLite**

SQLite supports **ROWID** columns as 64-bit integers. Informix® rows are 16-bit integers.

**Solution**

If your Genero BDL application uses rowid columns, review the program logic to use primary keys instead. If the database table does not define a primary key, it should be added. All references to SQLCA.SQLERRD[6] must be removed, because this variable will not hold the **ROWID** of the last modified row.

If you cannot avoid the use of rowids, you must change the type of the variables which hold **ROWID** values. Instead of using **INTEGER**, use **DECIMAL(20)**.

For databases where the keyword of the rowid pseudo-column is different than "**ROWID**", the translation can be controlled with the following FGLPROFILE entry:

```plaintext
dbi.database.dsname.ifxemul.rowid = \true \false
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

Using **ROWID columns** on page 644
Automatic **ROWID** columns is not a common database feature.

**Foreign key support**

Foreign keys are an important feature in modern database design, to enforce database integrity:

```sql
CREATE TABLE orders ( ... ,
   FOREIGN KEY(ord_customer) REFERENCES customer(cust_num) )
```

SQLite (3.6.19 and +) implements foreign key support, but this feature is not enabled by default. In fact, it is possible to define foreign keys on tables, but when doing database operations, the constraints are not enforced until you enable it explicitly with a PRAGMA command.

**Solution**

In order to turn on foreign key constraint checking, you must issue a PRAGMA command, which can for example be executed with a EXECUTE IMMEDIATE instruction:

```sql
EXECUTE IMMEDIATE "PRAGMA foreign_keys = ON"
```

Future releases of SQLite might change this, so that foreign key constraints enabled by default.
Related concepts

Data definition statements on page 613
It is recommended to avoid use of DDL in programs.

TEXT and BYTE (LOB) types

Informix®

Informix® provides the TEXT, BYTE, CLOB and BLOB data types to store very large texts or binary data.

Legacy Informix® 4GL applications typically use the TEXT and BYTE types.

Genero BDL does not support the Informix® CLOB and BLOB types.

SQLite

SQLite 3 provides the TEXT and BLOB native data types for large objects storage.

Solution

The SQLite database interface can convert BDL TEXT data to SQLite TEXT and BYTE data to SQLite BLOB.

The TEXT and BYTE types translation can be controlled with the following FGLPROFILE entries:

\[
\begin{align*}
\text{dbi.database.dsname.ifxemul.text} & = \{ \text{true} \mid \text{false} \} \\
\text{dbi.database.dsname.ifxemul.byte} & = \{ \text{true} \mid \text{false} \}
\end{align*}
\]

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

Related concepts

Using portable data types on page 614
Only a limited set of data types are really portable across several database engines.

Name resolution of SQL objects

Informix®

Informix® uses the following form to identify an SQL object:

\[
\text{database[@dbname]:}[\{\text{owner}|"owner"\}].\text{identifier}
\]

The ANSI convention is to use double quotes for identifier delimiters (For example: "customer"."cust_name").

Informix® database object names are not case-sensitive in non-ANSI databases. When using double-quoted identifiers, Informix® becomes case sensitive.

With non-ANSI Informix® databases, you do not have to give a schema name before the tables when executing an SQL statement:

\[
\text{SELECT ... FROM customer WHERE ...}
\]

In Informix® ANSI compliant databases:

- The table name must include "owner", unless the connected user is the owner of the database object.
- The database server shifts the owner name to uppercase letters before the statement executes, unless the owner name is enclosed in double quotes.
SQLite

SQLite database object names are case-insensitive. Using double quotes to surround table names is possible. However, the letter case is kept even without double quotes:

```sql
sqlite> CREATE TABLE tab1 ( pk INT );
sqlite> CREATE TABLE "TAB2" ( pk INT );
sqlite> CREATE TABLE Tab3 ( pk INT );
sqlite> .tables
TAB2  Tab3  tab1
sqlite> .schema "TAB3"
CREATE TABLE Tab3 ( pk INT );
```

In an SQLite, if a prefix is specified as part of an object reference, it must be either "main", or "temp" or the schema-name of an attached database. There is no such concept as user schema in SQLite.

Solution

To write portable SQL, regarding database object names:

1. Use simple database object names (without any owner/schema prefix)
2. Do not use double quotes to surround database object identifiers.
3. If needed, define public synonyms to reference database objects in others databases/schema.
4. Specify database object identifiers in lowercase.

See also Naming database objects on page 637.

Data manipulation

SQLite related data manipulation topics.

Outer joins

Informix® OUTER() syntax

In Informix® SQL, outer joins can be defined in the FROM clause with the OUTER keyword:

```sql
SELECT ... FROM a, OUTER (b)
    WHERE a.key = b.akey

SELECT ... FROM a, OUTER(b,OUTER(c))
    WHERE a.key = b.akey
    AND b.key1 = c.bkey1 AND b.key2 = c.bkey2
```

Informix® also supports the ANSI OUTER join syntax, which is the recommended way to specify outer joins with recent SQL database engines:

```sql
SELECT ... FROM cust LEFT OUTER JOIN order
    ON cust.key = order.custno
    WHERE ... 
```

SQLite

SQLite supports the ANSI outer join syntax:

```sql
SELECT ... 
FROM cust LEFT OUTER JOIN order 
    LEFT OUTER JOIN item 
    ON order.key = item.ordno 
    ON cust.key = order.custno 
WHERE order.cdate > current date
```
**Solution**

The Genero database drivers can convert Informix® OUTER specifications to ANSI outer joins.

**Note:** For better SQL portability, use the ANSI outer join syntax instead of the old Informix® OUTER syntax.

The outer join translation can be controlled with the following FGLPROFILE entry:

```plaintext
dbi.database.dsnname.ifxemulouters = \true \false
```

For more details see [IBM Informix emulation parameters in FGLPROFILE](#) on page 668.

- **Prerequisites:**
  1. In the FROM clause, the main table must be the first item and the outer tables must be listed from left to right in the order of outer levels.

   Example which does not work:

   ```plaintext
   ... FROM OUTER(tab2), tab1
   ```

  2. The outer join in the WHERE clause must use the table name as prefix:

   ```plaintext
   ... WHERE tab1.col1 = tab2.col2
   ```

- **Restrictions:**
  1. Statements composed by 2 or more SELECT instructions are not supported:

   ```plaintext
   SELECT ... UNION SELECT ...
   ```

   or:

   ```plaintext
   SELECT ... WHERE col IN (SELECT...)
   ```

  2. Additional conditions on outer table columns cannot be detected and therefore are not supported:

   ```plaintext
   ... FROM tab1, OUTER(tab2)
   WHERE tab1.col1 = tab2.col2
   AND tab2.colx > 10
   ```

  3. Using subscript in outer conditions:

   ```plaintext
   ... FROM tab1, OUTER(tab2)
   WHERE tab1.col1[1,3] = tab2.col2[1,3]
   ```

- **Notes:**
  1. Table aliases are detected in OUTER expressions.

   OUTER example with table alias:

   ```plaintext
   ... OUTER(tab1 alias1) ...
   ```

  2. In the outer join, `outertab.col` can be placed on both right or left sides of the equal sign:

   ```plaintext
   ... WHERE outertab.col1 = maintab.col2
   ```

  3. Table names detection is not case-sensitive:

   ```plaintext
   SELECT ... FROM tab1, TAB2
   WHERE tab1.col1 = tab2.col2
   ```
4. **Temporary tables** are supported in OUTER specifications:

```
CREATE TEMP TABLE tt1 ( ... )
SELECT ... FROM tab1, OUTER(tt1) ... 
```

**Transactions handling**

**Informix®**

With the Informix® native mode (non ANSI):

- Transactions blocks start with `BEGIN WORK` and terminate with `COMMIT WORK` or `ROLLBACK WORK`.
- Statements executed outside a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.

```
UPDATE tab1 SET ...  -- auto-committed
BEGIN WORK            -- start of TX block
UPDATE tab1 SET ...
UPDATE tab2 SET ...
...
COMMIT WORK           -- end of TX block 
```

Informix® version 11.50 introduces savepoints:

```
SAVEPOINT name [UNIQUE]
ROLLBACK [WORK] TO SAVEPOINT [name] 
RELEASE SAVEPOINT name 
```

**SQLite**

With SQLite:

- Individual SQL statements are auto-committed.
- Transactions start with `BEGIN TRANSACTION` and end with `COMMIT TRANSACTION` or `ROLLBACK TRANSACTION`.
- DDL statements can be executed (and canceled) in transaction blocks.

SQLite supports savepoints with some differences compared to Informix®:

- `SAVEPOINT` can be used instead of `BEGIN TRANSACTION`. In this case, `RELEASE` is like a `COMMIT`.
- The syntax of a rollback to the savepoint is `ROLLBACK [TRANSACTION] TO [SAVEPOINT] name`.
- The syntax of a release of the savepoint is `RELEASE [SAVEPOINT] name`.
- Rollback must always specify the savepoint name.
- You cannot rollback to a savepoint if cursors are opened.
- In SQLite versions prior to 3.7, you cannot rollback are transaction if a cursor is open.

**Solution**

Regarding transaction control instructions, BDL applications do not have to be modified in order to work with SQLite. The `BEGIN WORK`, `COMMIT WORK` and `ROLLBACK WORK` commands are translated the native commands of SQLite.

**Note:** If you want to use savepoints, always specify the savepoint name in `ROLLBACK TO SAVEPOINT` and do not open cursors during transactions using savepoints. If you are using an SQLite versions prior to 3.7, it is not possible to perform a `ROLLBACK WORK` if a cursor (with hold) is currently open.

See also **SELECT FOR UPDATE**

**Related concepts**

Database transactions on page 600
Database transactions define a set of SQL instructions to be executed as a whole, or rolled back as a whole.

**Temporary tables**

**Informix®**

Informix® temporary tables are created with the CREATE TEMP TABLE DDL instruction or with SELECT ... INTO TEMP statement:

```sql
CREATE TEMP TABLE tt1 ( pkey INT, name VARCHAR(50) )
CREATE TEMP TABLE tt2 ( pkey INT, name VARCHAR(50) ) WITH NO LOG
SELECT * FROM tab1 WHERE pkey > 100 INTO TEMP tt2
```

Temporary tables are automatically dropped when the SQL session ends, but they can also be dropped with the DROP TABLE command. There is no name conflict when several users create temporary tables with the same name.

BDL REPORTs can create a temporary table when the rows are not sorted externally (by the source SQL statement).

Informix® allows you to create indexes on temporary tables. No name conflict occurs when several users create an index on a temporary table by using the same index identifier.

When creating temporary tables in Informix®, the WITH NO LOG clause can be used to avoid the overhead of recording DML operations in transaction logs.

**SQLite**

SQLite supports temporary tables with the CREATE TEMP TABLE statement:

```sql
CREATE TEMP TABLE mytt1 ( pkey INT, name VARCHAR(50) )
CREATE TEMP TABLE mytt2 AS SELECT * FROM source
```

**Solution**

Informix® CREATE TEMP TABLE statements are kept as is, while SELECT INTO TEMP statements are converted to SQLite native SQL CREATE TEMP TABLE AS SELECT ... .

**Important:**

Simple Informix-style SQL statement creating temporary tables can be converted to a native SQL equivalent instruction. However, complex SQL statements such as SELECT ... INTO TEMP with subqueries may fail. In such cases, create a view from the complex query and then create the temp table from the view. Or, disable Informix emulation and use the native SQL syntax to create the temporary table (EXECUTE IMMEDIATE "/* fglhint_no_ifxemul */ ...")

With Informix SQL, if the source table has a column defined as SERIAL or BIGSERIAL, a SELECT ... INTO TEMP will produce a new temp table with an auto-incremented serial column. With the SELECT ... INTO TEMP emulation for non-Informix databases, not using the native sequence generators (such as IDENTITY columns in SQL Server), the resulting temporary table will get a simple INTEGER or BIGINT column, instead of an auto-incremented column.

The general FGLPROFILE entry to control temporary table emulation is:

```sql
dbi.database.dsname.ifxemul.temptables = ↓ true ↓ false ↓
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

Temporary tables on page 641
Syntax for temporary table creation is not unique across all database engines.

**Substrings in SQL**

**Informix®**

Informix® SQL statements can use subscripts on columns defined with the character data type:

```sql
SELECT ... FROM tab1 WHERE col1[2,3] = 'RO'
SELECT ... FROM tab1 WHERE col1[10] = 'R' -- Same as col1[10,10]
UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...
SELECT ... FROM tab1 ORDER BY col1[1,3]
```

**Important:** With other database servers as Informix®, when the subscript notation is used to modify column values in UPDATE statement, or as ORDER BY element, you will get and SQL error:

```sql
UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...
SELECT ... FROM tab1 ORDER BY col1[1,3]
```

**SQLite**

SQLite provides the `SUBSTR(expr, start, length)` function, to extract a substring from a string expression:

```sql
SELECT SUBSTR(col,10,5) ...
```

**Solution**

Replace all Informix® `col[x,y]` right-value expressions by `SUBSTR(col,x,y-x+1)`.

Rewrite UPDATE and ORDER BY clauses using `col[x,y]` expressions.

The translation of `col[x,y]` expressions can be controlled with the following FGLPROFILE entry:

```sql
dbi.database.dsnname.ifxemul.colsubs = 1 true 0 false
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

**Substring expressions** on page 644

Handle substrings expressions with different database engines.

**MATCHES and LIKE**

**Informix®**

Informix® supports MATCHES and LIKE operators in SQL statements.

MATCHES expects * and ? wild-card characters, while LIKE uses the % and _ wild-cards as equivalents.

```sql
( col MATCHES 'Smi*' AND col NOT MATCHES 'R?x' )
( col LIKE 'Smi%' AND col NOT LIKE 'R_x' )
```

MATCHES accepts also brackets notation, to specify a set of matching characters at a given position:

```sql
( col MATCHES '[Pp]aris' )
( col MATCHES '[0-9][a-z]*' )
```
SQLite
The SQLite does not provide an equivalent of the Informix® MATCHES operator.
The LIKE operator is supported.

Solution
The database driver is able to translate Informix® MATCHES expressions to LIKE expressions, when no [ ] bracket character ranges are used in the MATCHES operand.
The MATCHES to LIKE expression translation is controlled by the following FGLPROFILE entry:

```
dbi.database.dbname.ifxemul.matches = \{ true \ | \ false \}
```

Important: Only [NOT] MATCHES followed by a search pattern provided as a string literal can be converted by ODI drivers. A [NOT] MATCHES followed by a ? question mark parameter place holder is not translated!
For maximum portability, consider replacing the MATCHES expressions with LIKE expressions in all SQL statements.
Avoid using CHAR(N) types for variable length character data (such as name, address).

Related concepts
MATCHES and LIKE operators on page 645
Use the standard LIKE operator instead of the MATCHES operator.

The LENGTH() function

Informix®
Informix® provides the LENGTH() function to count the number of bytes of a character string expression:

```
SELECT LENGTH("aaa"), LENGTH(col1) FROM table
```

Informix® LENGTH() does not count the trailing blanks for CHAR or VARCHAR expressions, while SQLite counts the trailing blanks.
Informix® LENGTH() returns 0 when the given string is empty. That means, LENGTH('') = 0.

SQLite
SQLite supports the LENGTH() function, but there are some differences with Informix® LENGTH().
The SQLite LENGTH() function counts trailing blanks. When using a CHAR column, values are blank padded, and the function returns the size of the CHAR column. When using a VARCHAR column, trailing blanks are significant, and the function returns the number of characters, including trailing blanks.
When passing NULL as parameter, the SQLite LENGTH() function returns NULL.

Solution
The SQL LENGTH() function name can be used with SQLite.
Check if the trailing blanks are significant when using the LENGTH() SQL function in your application.
To count the number of characters by ignoring the trailing blanks, you must use the RTRIM() function:

```
SELECT LENGTH(RTRIM(col1)) FROM table
```

Related concepts
The LENGTH() function in SQL on page 647
The semantics of the \texttt{LENGTH()} SQL function differs according to the database engine.

**Row limiting clause**

**Informix®**

Informix® SQL supports the \texttt{SKIP} and \texttt{FIRST/LIMIT} keywords to limit the number of rows of a result set:

\begin{verbatim}
SELECT SKIP 10 FIRST 20 customer.* FROM customer ... ORDER BY cust_name
\end{verbatim}

This Informix SQL syntax is not portable.

Recent database engines support the row limiting clause syntax defined by the SQL standard:

\begin{verbatim}
SELECT ... OFFSET n ROWS FETCH FIRST m ROWS ONLY
\end{verbatim}

This should be the preferred syntax to be used, if all target database types support this \texttt{SELECT} clause.

The ODI database drivers can convert the Informix SQL \texttt{SKIP/FIRST} row limiting clause to a native SQL equivalent, if the row limiting clause parameters are simple integer literals (the clause is not translated when using SQL parameters / program variables).

**Important:** In addition to the \texttt{SKIP/FIRST} clause of the projection clause, Informix SQL supports also a \texttt{LIMIT} clause after the \texttt{ORDER BY} clause:

\begin{verbatim}
SELECT customer.* FROM customer ... ORDER BY cust_name LIMIT 10
\end{verbatim}

This Informix SQL syntax construction is not converted by the ODI drivers. To benefit from the conversion, review the code to use the Informix SQL \texttt{SKIP/FIRST} clause instead.

**SQLite**

SQLite supports the following row limiting clause:

\begin{verbatim}
SELECT ... ORDER BY ... LIMIT m OFFSET n
\end{verbatim}

**Solution**

The Informix SQL row limiting clause can be converted by the SQLite driver to the native SQL equivalent clause, when the parameters are simple integer literals.

**Note:** The row limiting clause must not use SQL parameters. Only row limiting clauses using integer constants will be converted.

The translation of the Informix SQL row limiting clause can be controlled with the following FGLPROFILE entry:

\begin{verbatim}
dbi.database.dsname.ifxemul.rowlimiting = [true] [false]
\end{verbatim}

For more details see [IBM Informix emulation parameters in FGLPROFILE](#) on page 668.

**Related concepts**

- [Row limiting clause (SELECT)](#) on page 650
How to use the right clause to limit the number of rows produced by a SELECT statement?

**BDL programming**

SQLite related programming topics.

**INSERT cursors**

**Informix®**

Informix® provides *insert cursors* to optimize row creation in a database. An insert cursor is declared as a cursor, and rows as added with the PUT instruction. The rows are buffered and sent to the database server when executing a FLUSH instruction, or when the cursor is closed with CLOSE. When using transactions in Informix®, the OPEN, PUT and FLUSH instructions must be executed within a transaction block.

```
DECLARE c1 CURSOR FOR INSERT INTO tab1 ...
BEGIN WORK
OPEN c1
WHILE ...
  PUT c1 USING var-list
END WHILE
CLOSE c1
COMMIT WORK
```

**SQLite**

SQLite does not support insert cursors.

**Solution**

Insert cursors are emulated by the database interface, using basic INSERT SQL instructions. The performances might be not as good as with Informix®, but the feature is fully supported.

**Related concepts**

*Insert cursors* on page 633

Using insert cursors with non-Informix databases.

**SELECT ... FOR UPDATE**

**Informix®**

Legacy BDL programs typically use a cursor with SELECT FOR UPDATE to implement pessimistic locking and avoid several users editing the same rows:

```
DECLARE cc CURSOR FOR
SELECT ... FROM tab WHERE ... FOR UPDATE
OPEN cc
FETCH cc <-- lock is acquired
...
CLOSE cc <-- lock is released
```

The row must be fetched in order to set the lock.

If the cursor is local to a transaction, the lock is released when the transaction ends. If the cursor is declared WITH HOLD, the lock is released when the cursor is closed.

Informix® provides the SET LOCK MODE instruction to define the lock wait timeout:

```
SET LOCK MODE TO \| WAIT \| NOT WAIT \| WAIT seconds \|
```

The default mode is NOT WAIT.
SQLite
SQLite does not support the `FOR UPDATE` close in `SELECT` syntax.

Solution
Review the program logic when using `SELECT ... FOR UPDATE` statements.

Related concepts
Cursors WITH HOLD on page 631
Programming WITH HOLD cursors using `SELECT` with and without `FOR UPDATE` clause.

Informix®
Informix® closes opened cursors automatically when a transaction ends, unless the `WITH HOLD` option is used in the `DECLARE` instruction:

```sql
DECLARE c1 CURSOR WITH HOLD FOR SELECT ...
OPEN c1
BEGIN WORK
FETCH c1 ...
COMMIT WORK
FETCH c1 ...
CLOSE c1
```

SQLite
SQLite does not close cursors when a transaction ends.

SQLite does not support the `FOR UPDATE` close in `SELECT` syntax: Therefore, there cannot be a combination of `WITH HOLD + SELECT ... FOR UPDATE`.

Solution
BDL cursors declared `WITH HOLD` remain open even after terminating a transaction with a `COMMIT WORK` or `ROLLBACK WORK`.

For consistency with other database brands, database cursors that are not declared `WITH HOLD` are automatically closed, when a `COMMIT WORK` or `ROLLBACK WORK` is performed.

Important: Opening a `WITH HOLD` cursor declared with a `SELECT FOR UPDATE` results in an SQL error; in the same conditions, this does not normally appear with Informix®. Review the program logic in order to find another way to set locks.

Related concepts
Cursors WITH HOLD on page 631
Programming WITH HOLD cursors using `SELECT` with and without `FOR UPDATE` clause.

UPDATE/DELETE … WHERE CURRENT OF

Informix®
Informix® allows positioned `UPDATE`s and `DELETE`s with the "WHERE CURRENT OF cursor" clause, if the cursor has been DECLARED with a `SELECT ... FOR UPDATE` statement.

SQLite
SQLite does not support the `SELECT ... FOR UPDATE` and `UPDATE/DELETE WHERE CURRENT OF` SQL instruction.
Solution

UPDATE/DELETE ... WHERE CURRENT OF is not supported with SQLite.

As a replacement of WHERE CURRENT OF, if the database table is defined with a primary key column, use the value fetched from the SELECT [FOR UPDATE] cursor in the WHERE clause of the UPDATE/DELETE statement.

Related concepts

Positioned UPDATE/DELETE on page 631

Using positioned updates/deletes with named database cursors.

LOAD and UNLOAD

Informix®

Informix® provides two SQL instructions to export / import data from / into a database table:

The UNLOAD instruction copies rows from a database table into a text file:

```
UNLOAD TO "filename.unl" SELECT * FROM tab1 WHERE ..
```

The LOAD instructions insert rows from a text file into a database table:

```
LOAD FROM "filename.unl" INSERT INTO tab1
```

SQLite

SQLite does not natively provide LOAD and UNLOAD instructions.

Solution

LOAD and UNLOAD instruction are implemented in the Genero BDL runtime system with basic INSERT (for LOAD) or SELECT (for UNLOAD) SQL commands. The LOAD and UNLOAD instruction can be supported with various database servers.

However, LOAD and UNLOAD require the description of the column types in order to work, that can lead to some differences in the data formatting.

Note: If no transaction is started, the LOAD instruction will automatically execute a BEGIN WORK and COMMIT WORK when finished, or ROLLBACK WORK if a row insertion failed while loading. Terminating a transaction will automatically close cursors not defined WITH HOLD option. To workaround this situation, see more details in the LOAD on page 736 reference topic.

The LOAD and UNLOAD BDL instructions are supported with SQLite.

Related concepts

LOAD and UNLOAD instructions on page 639

The LOAD and UNLOAD instructions can produce different data formats depending on the database server type.

SQL Interruption

Informix®

With Informix®, it is possible to interrupt a long running query if the SQL INTERRUPT ON option.

SQLite

SQLite supports SQL Interruption: The db client must issue an sqlite3_interrupt() ODBC call to interrupt a query.
Solution
The SQLite database driver supports SQL interruption and converts the native SQL execution status SQLITE_ABORT to the Informix® error code -213.

Related concepts
Using SQL interruption on page 601
Interrupt long running SQL queries, or interrupt queries waiting for locked data.

Scrollable cursors

Informix®
Informix® SQL and Genero BDL support scrollable cursors when you specify the SCROLL clause in the DECLARE cursor instruction:

```sql
DECLARE c1 SCROLL CURSOR FOR SELECT ...
```

**Important:** Informix does not allow to fetch TEXT/BYTE columns with scrollable cursors. If you declare a scroll cursor with a SELECT containing TEXT/BYTE columns, Informix will produce the SQL error -611 when executing the OPEN instruction.

SQLite
SQLite does not support scrollable cursors.

Solution
The SQLite database driver emulates scrollable cursors by fetching rows in a temporary file.

**Important:** With SQLite is it NOT possible to use LOB columns in a scrollable cursor. If TEXT/BYTE columns are used with a scrollable cursor, the OPEN instruction will produce the SQL error -611 (as with Informix).

See Scrollable cursors on page 623 for more details about scroll cursor emulation.

Modifying many rows in a table

SQLite
SQLite is very slow when doing commits, because of the technique used to ensure data integrity (see SQLite documentation for details).

When a program executes a DML statement like INSERT, it will be automatically committed by SQLite. As result, if you do not enclose the SQL instruction between BEGIN WORK / COMMIT WORK, there will be as many commits as data manipulation statements.

For example, it takes about 10 seconds to insert 1000 rows on an Intel core i7 2.60GHz CPU / 5400.0 RPM HDD computer.

Solution
If a program must modify many rows in a table, execute the SQL statement within a transaction block delimited by BEGIN WORK / COMMIT WORK instructions. This will dramatically speed up the program with SQLite, and even with other non-Informix database servers.

See Performance with transactions on page 652.
### Optimizing database file usage

**SQLite**

By default, when deleting a large amount of data in an SQLite database, it leaves behind empty space, causing the database file to be larger than strictly necessary.

This might be an issue with some mobile applications, when the disk space of the mobile device is limited.

**Solution**

Execute the `VACUUM` SQL command, to truncate the database file and reduce the disk usage.

According to the application, the `VACUUM` command can be executed:

- when starting the application,
- after doing a large db operation (like a synchronization with a central db),
- as a manual option that the user can trigger.

Note that SQLite also supports "PRAGMA auto_vacuum", but it appears that it's not as efficient as the `VACUUM` command, regarding page fragmentation.

Pay attention to the fact that `VACUUM` needs twice the disk space of the actual database file, because it rebuilds totally the db file.

`VACUUM` is not Informix SQL syntax, use `EXECUTE IMMEDIATE` to perform this SQL statement:

```
EXECUTE IMMEDIATE "VACUUM"
```

### SAP® ASE

**Supported versions**

Genero BDL supports the following SAP® Adaptive Server® Enterprise (ASE) database version:

- SAP® ASE 16.x

**Note:** SAP® ASE was originally known as Sybase ASE and references to its original title may still be found in this documentation.

**Installation (Runtime Configuration)**

SAP® ASE related installation topics.

#### Install SAP ASE and create a database - database configuration/design tasks

If you are tasked with installing and configuring the database, here is a list of steps to be taken:

1. Install SAP® ASE software on your computer, with the SAP® client software.

   Make sure that the server is started and environment variables are properly set (On UNIX™, you will find SYBASE.* shell scripts to source in the installation directory).

2. Try to connect to the server with the `isql` command line tool.

   If needed, change the password of the "sa" database administrator:

   ```
   $ isql -S server_name -U sa
   1> sp_password old_password, new_password
   2> go
   Password correctly set.
   (return status = 0)
   ```

   Starting with SAP® ASE 15.7, the password of the sa user is defined at installation time.
3. Define server's default character set: You must identify what server character set you want to use (typically, utf8) and re-configure the server if needed.

With SAP® ASE, the db character set cannot be specified at the database level, it is defined at the server level, typically during the installation. It is also possible to change the server character set with the `charset` utility and with the `sp_configure` stored procedure. You have to shutdown the server, start a first time to have the server take the new character set into account and then restart a second time for use. See SAP® documentation for more details or more recent versions of SAP® ASE.

Make sure that you select a case-sensitive character set / sort order.

**Note:** Check the `$SYBROOT/locales/locales.dat` file, to make sure that your current locale (LANG/LC_ALL) is listed in the file. You may want to add the following lines for UTF-8 support, under the section of your operating system:

```
locale = POSIX, us_english, utf8
locale = en_US.utf8, us_english, utf8
; Windows only:
locale = .fglutf8, us_english, utf8
```

Example:

```
$ export DSQUERY=servername
$ charset -Usa -P binary.srt utf8
Please enter sa's Password:
Loading file 'binary.srt'.
Found a [sortorder] section.
This is Class-1 sort order.
Finished loading the Character Set Definition.
Finished loading file 'binary.srt'.
1 sort order loaded successfully
$ isql -Usa -P
1> sp_configure 'default sortorder id', 50, 'utf8'
2> go
3> shutdown
4> go
Server SHUTDOWN by request.
ASE is terminating this process.
........
$ $SYBROOT/ASE_*/install/RUN_servername
... (server makes some initialization / setup and stops) ...
$ $SYBROOT/ASE_*/install/RUN_servername
...
```

4. Create a new SAP® ASE database entity, with sufficient storage devices for data and transaction log.

Use either the SAP® Central, the SAP® Control Center GUI tool or use isql with SQL commands.

Connect to the server with the sa user.

First create database devices for data and transaction log. Define a transaction that can hold the biggest transaction your application can do to avoid administrative tasks to dump the log when the server hangs. When creating the database, use the new created database devices as database segments:

```
use master
go
```
5. Leave the default transaction mode ("unchained" mode), to force explicit transaction start and end commands.
   See the `set chained` command for more details.

6. The database allows NULLs by default when creating columns.
   This is controlled by the 'allow nulls by default' option. If this option is set to OFF, columns created without NULL or NOT NULL keywords are NOT NULL by default:

   ```sql
   master..sp_dboption dbname, 'allow nulls by default', true
   go
   ```

7. The database must allow Data Definition Language (DDL) statements in transaction blocks.
   To turn this on, use following commands:

   ```sql
   master..sp_dboption dbname, 'ddl in tran', true
   go
   checkpoint
   go
   ```

8. For development purpose, consider setting the database option to truncate the transaction log when a checkpoint occurs, otherwise you will have to dump the transaction log when it is full.
   Command to automatically truncate the transaction log on checkpoint:

   ```sql
   master..sp_dboption dbname, 'trunc log on chkpt', true
   go
   ```

9. Create a new login dedicated to your application: the application administrator.
   Assign the new created database as default database for this user:

   ```sql
   use dbname
   go
   sp_addlogin 'username', 'password', dbname, ... options ...
   go
   ```

10. Create a new database user linked to the new application administrator login:
    In SAP ASE Central, open to the "Databases" node, select "Users" and right-click "New" ...

    ```sql
    use dbname
    go
    sp_adduser 'username', 'group', ... options ...
    go
    ```

    See documentation for more details about database users and privileges. You must create groups to make tables visible to all users.

11. If you plan to use SERIAL emulation based on triggers using a registration table, create the SERIALREG table.
    Create the triggers for each table using a SERIAL. See issue SERIAL data types for more details.

12. Create the application tables.
    Convert Informix® data types to SAP® ASE data types. See topic data type Conversion Table for more details.
    In order to make application tables visible to all users, make sure that all users are members of the group of the
owner of the application tables. For more details, see SAP® ASE documentation ("Database object names and prefixes").

**Prepare the runtime environment - connecting to the database**

1. In order to connect to SAP® ASE, you must have a SAP® ASE database driver "dbmase" in $FGLDIR/dbdrivers.
2. If you want to connect to a remote database server, you must have the SAP® ASE Client Software installed on the computer running BDL applications.
   The SAP® Open Client Library is required.
3. Make sure that the SAP ASE client environment variables are properly set.
   Check for example SYBASE (the path to the installation directory), SYBASE_ASE (the name of the server sub-directory), SYBASE_OCS (the name of the client sub-directory), etc. See SAP® ASE documentation for more details.
4. Verify the environment variable defining the search path for SAP® ASE OCS database client shared libraries (libsybct.so, libsybcs.so UNIX™, LIBSYBCT.DLL and LIBSYBCS.DLL on Windows®).

**Table 279: Shared library environment setting for SAP ASE**

<table>
<thead>
<tr>
<th>SAP® ASE version</th>
<th>Shared library environment setting</th>
</tr>
</thead>
</table>
| SAP® ASE 16.0 and higher | UNIX: Add $SYBASE_OCS/lib to LD_LIBRARY_PATH (or its equivalent).  
Windows: Add %SYBASE_OCS%\dll to PATH.  
Where SYBASE_OCS is the directory of the SAP ASE Open Client Software. |

5. The name of the SAP® ASE server must be registered in a configuration file.
   On UNIX™, the server name must be defined in the "interfaces" file located in $SYBASE. On Windows®, the server name must be defined in the "sql.ini" file located in %SYBASE\ini. You may want to define the DSQUERY environment variable to the name of the server. See SAP® ASE documentation for more details.

When connecting from a Genero program, both database and server names can be specified with:

```
database@server
```

For more details see the description for the connection data source parameter in DATABASE and CONNECT instructions.

6. Check the SAP® ASE database client locale settings.
   The SAP® ASE client locale must match the locale used by the runtime system (LC_ALL, LANG on UNIX™, ANSI code page on Windows®).

   By default, SAP® ASE OCS uses the character set defined by the operating system. On Windows®, this is the ANSI code page, on UNIX™ it is defined by LC_CTYPE, LC_ALL or LANG environment variables. Note that Genero BDL allows you to define the LANG environment variable also on Windows®. The value of the LANG environment variable must be listed in the "locales.dat" file under the $SYBASE/locales directory, otherwise you will get an error when connecting to the database.

   **Note:** Check the $SYBROOT/locales/locales.dat file, to make sure that your current locale (LANG/LC_ALL) is listed in the file. You may want to add the following lines for UTF-8 support, under the section of your operating system:

```
locale = POSIX, us_english, utf8  
locale = en_US.utf8, us_english, utf8  
; Windows only:  
locale = .fglutf8, us_english, utf8
```
locale = .utf8, us_english, utf8

See also SAP ASE OCS documentation regarding localization and character set definition.

7. Test the SAP ASE Client Software: Make sure the server is started and try to connect to a database by using the SAP ASE command interpreter:

$ isql -S server -U appadmin -P password

8. Set up the FGLPROFILE entries for database connections:
   a) Define the SAP ASE database driver:

   dbi.database.dbname.driver = "dbmase"

   b) Define the connection timeout with the following FGLPROFILE entry:

   dbi.database.dbname.ase.logintime = integer

   This entry defines the number of seconds to wait for a connection.
   Default is 5 seconds.

c) Define the number of rows to be pre-fetched for result sets:

   dbi.database.dbname.ase.prefetch.rows = integer

   Default is 10 rows.

Database concepts
SAP® ASE related database concepts topics.

Database concepts
As in Informix®, a SAP® ASE engine can manage multiple database entities. When creating a database object such as a table, SAP®ASE allows you to use the same object name in different databases.

Tip: If you have several Informix database entities, migrating from the Informix database to another database it is a good opportunity to centralize all tables in a single database. To avoid conflicts with table names, use a prefix when needed.

Concurrency management

Data consistency and concurrency concepts

• Data Consistency applies to situations when readers want to access data currently being modified by writers.
• Concurrent Data Access applies to situations when several writers are accessing the same data for modification.
• Locking Granularity defines the amount of data concerned when a lock is set (for example, row, page, table).

Informix®

Informix® uses a locking mechanism to handle data consistency and concurrency. When a process changes database information with UPDATE, INSERT or DELETE, an exclusive lock is set on the touched rows. The lock remains active until the end of the transaction. Statements performed outside a transaction are treated as a transaction containing a single operation and therefore release the locks immediately after execution. SELECT statements can set shared locks, depending on isolation level. In case of locking conflicts (for example, when two processes want to acquire an exclusive lock on the same row for modification, or when a writer is trying to modify data protected by a shared lock), the behavior of a process can be changed by setting the lock wait mode.

Control:

• Lock wait mode: SET LOCK MODE TO ... 
• Isolation level: SET ISOLATION TO ... 
• Locking granularity: CREATE TABLE ... LOCK MODE {PAGE|ROW}
• Explicit exclusive lock: SELECT ... FOR UPDATE

Defaults:
• The default isolation level is READ COMMITTED.
• The default lock wait mode is NOT WAIT.
• The default locking granularity is PAGE.

SAP ASE

As in Informix®, SAP® ASE uses locks to manage data consistency and concurrency. The database manager sets exclusive locks on the modified rows and shared locks when data is read, based on the isolation level. The locks are held until the end of the transaction. When multiple processes want to access the same data, the latest processes must wait until the first finishes its transaction or the lock timeout occurred. The lock granularity is at the row, page or table level. For more details, see SAP® ASE's Documentation.

Control:
• The lock wait mode can be controlled with: SET LOCK {WAIT seconds | NOWAIT}
• Isolation level: Can be set with: SET TRANSACTION ISOLATION LEVEL = {0|1|2|3}
• Locking granularity: Row, page or table level (controlled by locking schemes).
• Explicit locking: SELECT ... FOR UPDATE

Defaults:
• The default isolation level is Read Committed (readers cannot see uncommitted data; no shared lock is set when reading data).

Solution

The SET ISOLATION TO ... Informix® syntax is replaced by SET TRANSACTION ISOLATION LEVEL ... in SAP® ASE. The following table shows the isolation level mappings done by the SAP® ASE database driver:

| Table 280: Isolation level mappings done by the SAP ASE database driver |
|---|---|
| SET ISOLATION instruction in program | Native SQL command |
| SET ISOLATION TO DIRTY READ | SET TRANSACTION ISOLATION LEVEL = 0 |
| SET ISOLATION TO COMMITTED READ [READ COMMITTED] [RETAIN UPDATE LOCKS] | SET TRANSACTION ISOLATION LEVEL = 1 |
| SET ISOLATION TO CURSOR STABILITY | SET TRANSACTION ISOLATION LEVEL = 2 |
| SET ISOLATION TO REPEATABLE READ | SET TRANSACTION ISOLATION LEVEL = 3 |

For portability, it is recommended that you work with Informix® in the read committed isolation level, to make processes wait for each other (lock mode wait) and to create tables with the "lock mode row" option.

The SET LOCK MODE TO ... Informix® syntax is replaced by SET LOCK ... in SAP® ASE. If SET LOCK MODE TO WAIT is used in programs (i.e. wait forever), the driver will simulate this with a SET LOCK WAIT 5000 in SAP® ASE:

| Table 281: SET LOCK MODE instruction for SAP ASE |
|---|---|
| SET LOCK MODE instruction in program | Native SQL command |
| SET LOCK MODE TO NOT WAIT | SET LOCK NOWAIT |
| SET LOCK MODE TO WAIT n | SET LOCK WAIT n |
SET LOCK MODE instruction in program | Native SQL command
--- | ---
SET LOCK MODE TO WAIT | SET LOCK WAIT 5000

See the Informix® and SAP® ASE documentation for more details about data consistency, concurrency and locking mechanisms.

**Related concepts**

**Concurrent data access** on page 599
Understanding concurrent data access and data consistency.

**Optimistic locking** on page 624
Implementing optimistic locking to handle access concurrently to the same database records.

**Cursors WITH HOLD** on page 631
Programming WITH HOLD cursors using SELECT with and without FOR UPDATE clause.

**Transactions handling**

**Informix®**
With the Informix® native mode (non ANSI):
- Transactions blocks start with BEGIN WORK and terminate with COMMIT WORK or ROLLBACK WORK.
- Statements executed outside a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.

```
UPDATE tab1 SET ...   -- auto-committed
BEGIN WORK            -- start of TX block
UPDATE tab1 SET ...
UPDATE tab2 SET ...
...
COMMIT WORK           -- end of TX block
```

Informix® version 11.50 introduces savepoints:

```
SAVEPOINT name [UNIQUE]
ROLLBACK [WORK] TO SAVEPOINT [name]
RELEASE SAVEPOINT name
```

**SAP ASE**
- SAP® ASE supports two transaction modes:
  1. The SQL standards-compatible mode, called chained mode, to get implicit transaction.
  2. The default mode, called unchained mode, where transactions have to be started/ended explicitly.
- Transactions are started with "BEGIN TRANSACTION [name]."
- Transactions are validated with "COMMIT TRANSACTION [name]."
- Transactions are canceled with "ROLLBACK TRANSACTION [name]."
- Transactions save points can be placed with "SAVEPOINT [name]."
- SAP® ASE supports named and nested transactions.
- DDL statements can be executed in transactions blocks when the 'ddl in tran' option is set to true with:

```
master..sp_dboption dbname, 'ddl in tran', true
go
checkpoint
go
```
Solution
Informix® transaction handling commands are automatically converted to SAP® ASE instructions to start, commit or rollback transactions.

Make sure that the database uses the default *unchained* mode (set chained off) and allows DDLs in transactions ('ddl in tran' option is true).

Regarding the transaction control instructions, the BDL applications do not have to be modified in order to work with SAP® ASE.

Related concepts
Database transactions on page 600
Database transactions define a set of SQL instructions to be executed as a whole, or rolled back as a whole.

Database users

Informix®
Until version 11.70.xC2, Informix® database users must be created at the operating system level and must be members of the ‘informix’ group.

Starting with 11.70.xC2, Informix® supports database-only users with the CREATE USER instruction, as in most other db servers.

Any database user must have sufficient privileges to connect and use resources of the database; user rights are defined with the GRANT command.

SAP ASE
The system administrator (DBA) must declare the application users in the database with the GRANT statement.

You may also need to define groups in order to make tables visible to other users.

Solution
See SAP® ASE documentation for more details on database logins and users.

Related concepts
Database users and security on page 608
Properly identifying database users allows to use database security and audit features.

Setting privileges

Informix®
Informix® users must have at least the CONNECT privilege to access the database:

```sql
GRANT CONNECT TO username
```

Application administration users need the RESOURCE privilege to create tables:

```sql
GRANT RESOURCE TO username
```

Since version 7.20, Informix® supports database roles:

```sql
GRANT rolename TO username
```

SAP ASE
SAP® ASE supports the concept of roles to grant or revoke permissions to a group of users.
See SAP® ASE documentation for more details.

**Solution**

Informix® and SAP® ASE user privileges management are quite similar.

**Data dictionary**

SAP® ASE related data dictionary topics.

**Data type conversion table: Informix to SAP ASE**

<table>
<thead>
<tr>
<th>Informix® data types</th>
<th>SAP® ASE data types</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR (n)</td>
<td>CHAR (n) (limit = page size, ex:16384 bytes)</td>
</tr>
<tr>
<td>VARCHAR (n[,m])</td>
<td>VARCHAR (n) (limit = page size, ex:16384 bytes)</td>
</tr>
<tr>
<td>LVARCHAR (n)</td>
<td>VARCHAR (n) (limit = page size, ex:16384 bytes)</td>
</tr>
<tr>
<td>NCHAR (n)</td>
<td>NCHAR (n) (length in characters)</td>
</tr>
<tr>
<td>NVARCHAR (n[,m])</td>
<td>NVARCHAR (n) (length in characters)</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>BIT (must be NOT NULL!)</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>BIGINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>INT8</td>
<td>BIGINT</td>
</tr>
<tr>
<td>SERIAL (without start value!)</td>
<td>INTEGER (see note 1)</td>
</tr>
<tr>
<td>BIGSERIAL (without start value!)</td>
<td>BIGINT (see note 1)</td>
</tr>
<tr>
<td>SERIAL8 (without start value!)</td>
<td>BIGINT (see note 1)</td>
</tr>
<tr>
<td>DOUBLE PRECISION / FLOAT[(n)]</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>REAL / SMALLFLOAT</td>
<td>REAL</td>
</tr>
<tr>
<td>DECIMAL(p, s)</td>
<td>DECIMAL(p, s)</td>
</tr>
<tr>
<td>DECIMAL(p) with p&lt;=19</td>
<td>DECIMAL(2*p,p)</td>
</tr>
<tr>
<td>DECIMAL(p) with p&gt;19</td>
<td>N/A</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL(32,16)</td>
</tr>
<tr>
<td>MONEY(p, s)</td>
<td>DECIMAL(p, s)</td>
</tr>
<tr>
<td>MONEY(p)</td>
<td>DECIMAL(p, 2)</td>
</tr>
<tr>
<td>MONEY</td>
<td>DECIMAL(16,2)</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME HOUR TO MINUTE</td>
<td>BIGTIME</td>
</tr>
<tr>
<td>DATETIME HOUR TO SECOND</td>
<td>BIGTIME</td>
</tr>
<tr>
<td>DATETIME HOUR TO FRACTION(n)</td>
<td>BIGTIME</td>
</tr>
<tr>
<td>DATETIME YEAR TO MONTH</td>
<td>BIGTIME</td>
</tr>
<tr>
<td>DATETIME YEAR TO DAY</td>
<td>BIGDATETIME</td>
</tr>
<tr>
<td>DATETIME YEAR TO HOUR</td>
<td>BIGDATETIME</td>
</tr>
</tbody>
</table>
### Informix® data types | SAP® ASE data types
--- | ---
| DATETIME YEAR TO MINUTE | BIGDATETIME |
| DATETIME YEAR TO SECOND | BIGDATETIME |
| DATETIME YEAR TO FRACTION(n) | BIGDATETIME |
| INTERVAL q1 TO q2 | CHAR(50) |
| TEXT | TEXT |
| BYTE | IMAGE |

Notes:

1. For more details about serial emulation, see SERIAL and BIGSERIAL data type on page 1048.

**BOOLEAN data type**

**Informix®**

Informix® supports the BOOLEAN data type, which can store 't' or 'f' values.

Genero BDL implements the BOOLEAN data type in a different way: A BOOLEAN variable stores integer values 1 or 0 (for TRUE or FALSE). This type is designed to hold the result of a boolean expression.

**SAP ASE**

SAP® ASE provides the BIT data type to store boolean values.

However, unlike Informix® types, BIT columns cannot be NULL and thus you must specify the NOT NULL constraint when creating the table.

**Solution**

The SAP® ASE database interface converts BOOLEAN type to BIT columns and stores 1 or 0 values in the column.

You must explicitly specify the NOT NULL constraint in the CREATE TABLE statement.

The BOOLEAN type translation can be controlled with the following FGLPROFILE entry:

```plaintext
dbi.database.dsname.ifxemul.datatype.boolean = {true | false}
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

- Using portable data types on page 614

Only a limited set of data types are really portable across several database engines.

**CHAR and VARCHAR data types**

**Informix®**

Informix® supports the following character data types:

**Table 283: Informix® character data types**

<table>
<thead>
<tr>
<th>Informix® data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR (n)</td>
<td>SBCS and MBCS character data (max is 32767 bytes)</td>
</tr>
</tbody>
</table>
| VARCHAR (n,
m)      | SBCS and MBCS character data (max is 255 bytes) |
Informix® data type | Description
---|---
NCHAR (n) | Same as CHAR, with specific collation order
NVARCHAR (n[m]) | Same as VARCHAR, with specific collation order
LVARCHAR (n) | max size varies depending on the IDS version

With Informix®, both CHAR/VARCHAR and NCHAR/NVARCHAR data types can be used to store single-byte or multibyte encoded character strings. The only difference between CHAR/VARCHAR and NCHAR/NVARCHAR is in how they use sorting: N[VAR]CHAR types use the collation order, while [VAR]CHAR types use the byte order.

The character set used to store strings in CHAR/VARCHAR/NCHAR/NVARCHAR columns is defined by the DB_LOCALE environment variable.

The character set used by applications is defined by the CLIENT_LOCALE environment variable.

Informix® uses Byte Length Semantics (the size N that you specify in [VAR]CHAR (N) is expressed in bytes, not characters as in some other databases)

**SAP ASE**

SAP® ASE supports following data types to store character data:

**Table 284: SAP ASE character data types**

<table>
<thead>
<tr>
<th>SAP® ASE data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR (n)</td>
<td>SBCS or MBCS character data using the database character set, where n is specified in bytes (max is 16384 bytes)</td>
</tr>
<tr>
<td>VARCHAR (n)</td>
<td>SBCS or MBCS character data using the database character set, where n is specified in bytes (max is 16384 bytes)</td>
</tr>
<tr>
<td>NCHAR (n)</td>
<td>MBCS character data using the database character set, where n is specified in characters (max is 16384 bytes)</td>
</tr>
<tr>
<td>NVARCHAR (n)</td>
<td>MBCS character data using the database character set, where n is specified in characters (max is 16384 bytes)</td>
</tr>
<tr>
<td>UNICHAR (n)</td>
<td>UNICODE/UCS-2 character data, where n is specified in characters (max is 16384 characters)</td>
</tr>
<tr>
<td>UNIVARCHAR (n)</td>
<td>UNICODE/UCS-2 character data, where n is specified in characters (max is 16384 characters)</td>
</tr>
</tbody>
</table>

SAP® ASE implements the following character data types:

- **CHAR (N)** with N <= 16384 bytes
- **VARCHAR (N)** with N <= 16384 bytes
- **NCHAR (N)** with N <= 16384 characters
- **NVARCHAR (N)** with N <= 16384 characters
- **UNICHAR (N)** with N <= 16384 characters
- **UNIVARCHAR (N)** with N <= 16384 characters

Like Informix®, SAP® ASE can store multibyte characters in CHAR/VARCHAR columns, depending on the database character set. For example, SAP® can store UTF-8 strings in CHAR/VARCHAR columns. For multibyte character sets, you could also use the NCHAR/NVARCHAR or UNICHAR/UNIVARCHAR SAP® ASE types, the only difference with CHAR/VARCHAR is that the length is specified in characters instead of bytes. The UNICHAR/UNIVARCHAR types store characters in 16 bit UCS-2 charset only, but this is transparent to the database client.

SAP® ASE supports automatic character set conversion between the client application and the server. By default, the SAP® ASE database client character set is defined by the operating system locale where the database client runs.
On Windows®, it is the ANSI code page of the login session (can be overwritten by setting the LANG environment variable), on UNIX™ it is defined by the LC_CTYPE, LC_ALL or LANG environment variable. You may need to edit the $SYBASE/locales/locales.dat file to map the OS locale name to a known SAP® ASE character set.

Unlike most other database engines, SAP® ASE trims trailing blanks when inserting character strings in a VARCHAR column.

For example:

```sql
CREATE TABLE t1 ( k INT, vc VARCHAR(5))
INSERT INTO t1 VALUES ( 1, 'abc ')
SELECT '['||vc||']' FROM t1 WHERE k = 1
```

With other database servers you would get 1 blank after abc:

```sql
[abc ]
```

### Solution

If your application must support multibyte character sets like BIG5 or UTF-8, it is recommended that you use CHAR/VARCHAR SAP® ASE data types, where the length is specified in bytes like with Informix®.

Check that your database schema does not use CHAR, VARCHAR or LVARCHAR types with a length exceeding the SAP® ASE limit.

If your application creates tables with NCHAR/NVARCHAR types, the same type name will be used in SAP® ASE. Keep in mind that the size of NCHAR/NVARCHAR in SAP® is specified in characters, while Informix® uses a number of bytes.

When using a multibyte character set (such as UTF-8), define database columns with the size in character units, and use character length semantics in BDL programs with FGL_LENGTH_SEMANTICS=CHAR.

When extracting a database schema from a SAP® ASE database, the `fgldbseq` schema extractor uses the size of the column in characters, not the octet length. If you have created a CHAR(10 characters) column in a SAP® ASE database using the UTF-8 character set, the .sch file will get a size of 10, that will be interpreted by FGL_LENGTH_SEMANTICS as a number of bytes or characters.

Do not forget to properly define the database client character set, which must correspond to the runtime system character set.

Since trailing blanks are trimmed for VARCHAR columns, make sure that your application does not rely on this non-standard behavior.

See also the section about Localization

The CHAR/VARCHAR type translation can be controlled with the following FGLPROFILE entries:

```ini
[dbi.database.dsnname.ifxemul.datatype.char]
true  false
[dbi.database.dsnname.ifxemul.datatype.varchar]
true  false
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

CHAR and VARCHAR types on page 616
Using the CHAR and VARCHAR data types with different databases.

**Numeric data types**

**Informix®**

Informix® supports several data types to store numbers:

**Table 285: Informix® numeric data types**

<table>
<thead>
<tr>
<th>Informix® data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>16 bit signed integer</td>
</tr>
<tr>
<td>INTEGER</td>
<td>32 bit signed integer</td>
</tr>
<tr>
<td>BIGINT</td>
<td>64 bit signed integer</td>
</tr>
<tr>
<td>INT8</td>
<td>64 bit signed integer (replaced by BIGINT)</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>Equivalent to DECIMAL(16)</td>
</tr>
<tr>
<td>DECIMAL(p)</td>
<td>Floating-point decimal number (max precision is 32)</td>
</tr>
<tr>
<td>DECIMAL(p, s)</td>
<td>Fixed-point decimal number (max precision is 32)</td>
</tr>
<tr>
<td>MONEY</td>
<td>Equivalent to DECIMAL(16, 2)</td>
</tr>
<tr>
<td>MONEY(p)</td>
<td>Equivalent to DECIMAL(p, 2)   (max precision is 32)</td>
</tr>
<tr>
<td>MONEY(p, s)</td>
<td>Equivalent to DECIMAL(p, s)   (max precision is 32)</td>
</tr>
<tr>
<td>REAL / SMALLFLOAT</td>
<td>32-bit floating point decimal (C float)</td>
</tr>
<tr>
<td>DOUBLE PRECISION / FLOAT[(n)]</td>
<td>64-bit floating point decimal (C double)</td>
</tr>
</tbody>
</table>

**SAP ASE**

SAP® ASE supports the following numeric data types:

**Table 286: SAP ASE numeric data types**

<table>
<thead>
<tr>
<th>SAP® ASE data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>16 bit signed integer</td>
</tr>
<tr>
<td>INTEGER</td>
<td>32 bit signed integer</td>
</tr>
<tr>
<td>BIGINT</td>
<td>64 bit signed integer</td>
</tr>
<tr>
<td>DECIMAL(p, s)</td>
<td>Fixed point decimal</td>
</tr>
<tr>
<td>SMALLMONEY</td>
<td>32-bit floating point decimal with currency</td>
</tr>
<tr>
<td>MONEY</td>
<td>64-bit floating point decimal with currency</td>
</tr>
<tr>
<td>REAL</td>
<td>32-bit floating point decimal (C float)</td>
</tr>
<tr>
<td>FLOAT[(n)] (DOUBLE)</td>
<td>64-bit floating point decimal (C double)</td>
</tr>
</tbody>
</table>

Notes about SAP® ASE DECIMAL type:

- Without any decimal storage specification, the precision defaults to 18 and the scale defaults to zero:
  - DECIMAL in SAP ASE = \( \text{DECIMAL}(18, 0) \) in Genero BDL.
  - DECIMAL(p) in SAP ASE = \( \text{DECIMAL}(p, 0) \) in Genero BDL.
• The maximum precision is 38.

Notes about the SAP® ASE MONEY and SMALLMONEY types:

• SAP® ASE provides the MONEY and SMALLMONEY data types, but the currency symbol handling is quite different. Therefore, it is recommended to implement Informix® MONEY columns as DECIMAL columns in SAP® ASE.

Solution

Use the following conversion rules to map Informix® numeric types to SAP® ASE numeric types:

Table 287: Informix® data types and SAP ASE equivalents

<table>
<thead>
<tr>
<th>Informix®</th>
<th>SAP® ASE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>BIGINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>INT8</td>
<td>BIGINT</td>
</tr>
<tr>
<td>DECIMAL(p, s)</td>
<td>DECIMAL(p, s)</td>
</tr>
<tr>
<td>DECIMAL(p&lt;=19)</td>
<td>DECIMAL(2*p, p)</td>
</tr>
<tr>
<td>DECIMAL(p&gt;19)</td>
<td>N/A</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL(32,16)</td>
</tr>
<tr>
<td>MONEY(p, s)</td>
<td>DECIMAL(p, s)</td>
</tr>
<tr>
<td>MONEY(p)</td>
<td>DECIMAL(p, 2)</td>
</tr>
<tr>
<td>MONEY</td>
<td>DECIMAL(16, 2)</td>
</tr>
<tr>
<td>REAL / SMALLFLOAT</td>
<td>REAL</td>
</tr>
<tr>
<td>FLOAT[(n)] / DOUBLE PRECISION</td>
<td>FLOAT(n) (Where n must be from 1 to 15)</td>
</tr>
</tbody>
</table>

When creating tables from BDL programs, the database interface automatically converts Informix® numeric data types to corresponding SAP® ASE data types. In database creation scripts, apply the conversion rules as described in the above table.

Important: There is no SAP ASE equivalent for the Informix® DECIMAL(p) floating point decimal (i.e. without a scale). If your application is using such data types, you must review the database schema in order to use SQL Server compatible types. To workaround the SAP ASE limitation, the SAP ASE database driver converts DECIMAL(p) types to a DECIMAL(2*p, p), to store all possible numbers an Informix® DECIMAL(p) can store. However, the original Informix® precision cannot exceed 19, since SAP® ASE maximum DECIMAL precision is 38(2*19). If the original precision is bigger than 19, a CREATE TABLE statement executed from a Genero program will fail with an SAP ASE error 2756.

SAP® ASE does not support implicit character string to numeric conversions. For example, if you compare an integer column to '123' in a WHERE clause, SAP® ASE will raise a conversion error. The problem exists also when using CHAR or VARCHAR SQL parameters. Check that your programs do not use string literals or CHAR/VARCHAR SQL parameters in integer expressions, as in this example:

```sql
DEFINE pv CHAR(1)
CREATE TABLE mytable ( v1 INT, v2 INT )
LET pv = '1'
SELECT * FROM mytable WHERE v1 = '1' AND v2 = pv
```
The numeric types translation can be controlled with the following FGLPROFILE entries:

```
dbi.database.dsname.ifxemul.datatype.smallint = true | false
dbi.database.dsname.ifxemul.datatype.integer = true | false
dbi.database.dsname.ifxemul.datatype.bigt = true | false
dbi.database.dsname.ifxemul.datatype.int8 = true | false
dbi.database.dsname.ifxemul.datatype.decimal = true | false
dbi.database.dsname.ifxemul.datatype.money = true | false
dbi.database.dsname.ifxemul.datatype.float = true | false
dbi.database.dsname.ifxemul.datatype.smallfloat = true | false
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

**Using portable data types** on page 614

Only a limited set of data types are really portable across several database engines.

**DATE and DATETIME data types**

**Informix®**

Informix® provides two data types to store date and time information:

- **DATE** = for year, month and day storage.
- **DATETIME** = for year to fraction (1-5) storage.

The **DATE** type is stored as an **INTEGER** with the number of days since 1899/12/31.

The **DATETIME** type can be defined with various time units, by specifying a start and end qualifier. For example, you can define a datetime to store an hour-to-second time value with **DATETIME HOUR TO SECOND**.

The values of Informix® **DATETIME** can be represented with a character string literal, or as **DATETIME()** literals:

```
'2017-12-24 15:45:12.345'  -- a DATETIME YEAR TO FRACTION(3)
'15:45'  -- a DATETIME HOUR TO MINUTE
DATETIME(2017-12-24 12:45) YEAR TO MINUTE
DATETIME(12:45:56.333) HOUR TO FRACTION(3)
```

Informix® is able to convert quoted strings to **DATE**/**DATETIME** data, if the string contains matching environment parameters. The string to date conversion rules for **DATE** is defined by the **DBDATE** environment variable. The string to datetime format for **DATETIME** is defined by the **GL_DATETIME** environment variable.

**Note:** Within Genero programs, the string representation for **DATETIME** values is always ISO (**YYYY-MM-DD hh:mm:ss.fffff**)  

Informix® supports date arithmetic on **DATE** and **DATETIME** values. The result of an arithmetic expression involving dates/times is an **INTEGER** number of days when only **DATE** values are used, and an **INTERVAL** value if a **DATETIME** is used in the expression.

Informix® automatically converts an **INTEGER** to a **DATE** when the integer is used to set a value of a date column.

Informix® provides the **CURRENT [q1 TO q2]** operator, to get the system date/time on the server where the current database is located. When no qualifiers are specified, **CURRENT** returns a **DATETIME YEAR TO FRACTION(3)**. Informix also supports the **SYSDATE** operator, which returns the current system time as a **DATETIME YEAR TO FRACTION(5)**.

**Note:** The **USEOSTIME** configuration parameter must be set to 1 in order to get the subsecond precision in **CURRENT** and **SYSDATE** operators. See Informix documentation for more details.

**SAP ASE**

SAP® ASE provides the following data type to store date and time data:
Table 288: SAP ASE date/time data types

<table>
<thead>
<tr>
<th>SAP® ASE SERVER data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>for year, month, day storage</td>
</tr>
<tr>
<td>TIME</td>
<td>for hour, minutes, seconds, fraction(3) storage</td>
</tr>
<tr>
<td>SMALLDATETIME</td>
<td>for hour, minutes, seconds, fraction(3) storage</td>
</tr>
<tr>
<td>DATETIME</td>
<td>for hour, minutes, seconds, fraction(3) storage</td>
</tr>
<tr>
<td>BIGTIME</td>
<td>for hour, minutes, seconds, fraction(6) storage</td>
</tr>
<tr>
<td>BIGDATETIME</td>
<td>for year, month, day, hour, minutes, seconds, fraction(6) storage</td>
</tr>
</tbody>
</table>

SAP® ASE can convert quoted strings representing datetime data in the ANSI format. The `CONVERT()` SQL function allows you to convert strings to dates.

With SAP® ASE, you must use built-in functions to do date/time computing (for example, see `dateadd()` function).

**Solution**

Use the following conversion rules to map Informix® date/time types to SAP® ASE date/time types:

Table 289: Informix® data types and SAP ASE equivalents

<table>
<thead>
<tr>
<th>Informix® data type</th>
<th>SAP ASE data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME HOUR TO MINUTE</td>
<td>BIGTIME</td>
</tr>
<tr>
<td>DATETIME HOUR TO SECOND</td>
<td>BIGTIME</td>
</tr>
<tr>
<td>DATETIME HOUR TO FRACTION(n)</td>
<td>BIGTIME</td>
</tr>
<tr>
<td>DATETIME YEAR TO MONTH</td>
<td>BIGTIME</td>
</tr>
<tr>
<td>DATETIME YEAR TO DAY</td>
<td>BIGDATETIME</td>
</tr>
<tr>
<td>DATETIME YEAR TO HOUR</td>
<td>BIGDATETIME</td>
</tr>
<tr>
<td>DATETIME YEAR TO MINUTE</td>
<td>BIGDATETIME</td>
</tr>
<tr>
<td>DATETIME YEAR TO SECOND</td>
<td>BIGDATETIME</td>
</tr>
<tr>
<td>DATETIME YEAR TO FRACTION(n)</td>
<td>BIGDATETIME</td>
</tr>
</tbody>
</table>

The `DATE` and `DATETIME` types translation can be controlled with the following `FGLPROFILE` entries:

```plaintext
dbi.database.dsname.ifxemul.datatype.date = \{ true \ false \}
dbi.database.dsname.ifxemul.datatype.datetime = \{ true \ false \}
```

For more details see IBM Informix emulation parameters in `FGLPROFILE` on page 668.

SAP ASE has the same `DATE` data type as Informix® (year, month, day). So you can use SAP ASE `DATE` data type for Informix® `DATE` columns.

SAP ASE `BIGTIME` data type can be used to store Informix® `DATETIME HOUR TO MINUTE`, `DATETIME HOUR TO SECOND` and `DATETIME HOUR TO FRACTION(5)` values, and any other `DATETIME` type with qualifiers `HOUR, MINUTE, SECOND` and `FRACTION(n)`. Missing time parts default to 00:00:00.0. For example, when using a `DATETIME MINUTE TO FRACTION(3)` with the value of "45:23.999", the SAP ASE `BIGTIME` value will be "00:45:23.999".
Informix® DATETIME values with any precision from YEAR to FRACTION(5) can be stored in SAP ASE BIGDATETIME columns. Missing date or time parts default to 1900-01-01 00:00:00.0. For example, when using a DATETIME DAY TO MINUTE with the value of "23 11:45", the SAP ASE BIGDATETIME value will be "1900-01-23 11:45:00.0".

**Note:** SAP ASE does not support INTEGER to DATE automatic conversion.

**Related concepts**
- Date/time literals in SQL statements on page 635
- Good practices for date and time handling in SQL.

**INTERVAL data type**

**Informix®**

Informix® provides the INTERVAL data type to store a value that represents a span of time.

INTERVAL types are divided into two classes:
- **year-month** intervals. For example: INTERVAL YEAR(5) TO MONTH
- **day-time** intervals. For example: INTERVAL DAY(9) TO SECOND

INTERVAL columns can be defined with various time units, by specifying a start and end qualifier. For example, you can define an interval to store a number of hours and minutes with INTERVAL HOUR(n) TO MINUTE, where n defines the maximum number of digits for the hours unit.

The values of Informix® INTERVAL can be represented with a character string literal, or as INTERVAL() literals:

```
'19834 15:45:12.345'  -- an INTERVAL DAY(6) TO FRACTION(3)
'7623-11'   -- an INTERVAL YEAR(9) TO MONTH
INTERVAL(18734:45) HOUR(5) TO MINUTE
INTERVAL(-7634-11) YEAR(5) TO MONTH
```

**SAP ASE**

SAP® ASE does not provide a data type corresponding to the Informix® INTERVAL data type.

**Solution**

The INTERVAL data type and values are converted CHAR(50) column with SAP® ASE.

INTERVAL values can be stored and retrieved from the database. However, since SAP® ASE does not support a native interval type, arithmetics cannot be performed on the database side in SQL statements.

The INTERVAL types translation can be controlled with the following FGLPROFILE entry:

```
dbi.database.dsname.ifxemul.datatype.interval = \{true \| false \}
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**
- Using portable data types on page 614
  Only a limited set of data types are really portable across several database engines.

**SERIAL and BIGSERIAL data type**

**Informix®**

Informix® supports the SERIAL, BIGSERIAL data types to produce automatic integer sequences:

- SERIAL can produce 32 bit integers (INTEGER)
- BIGSERIAL can produced 64 bit integers (BIGINT)
• **SERIAL8** is a synonym for **BIGSERIAL**

Steps to use serials with Informix®:

1. Create the table with a column using **SERIAL**, or **BIGSERIAL**.
2. To generate a new serial, no value or a zero value is specified in the **INSERT** statement:

   ```sql
   INSERT INTO tab1 ( c ) VALUES ( 'aa' )
   INSERT INTO tab1 ( k, c ) VALUES ( 0, 'aa' )
   ```

3. After **INSERT**, the new value of a **SERIAL** column is provided in `SQLCA.SQLERRD[2]`, while the new value of a **BIGSERIAL** value must be fetched with a **SELECT** `dbinfo('bigserial')` query.

Informix® allows you to insert rows with a value different from zero for a serial column. Using an explicit value will automatically increment the internal serial counter, to avoid conflicts with future **INSERT** statements that are using a zero value:

```sql
CREATE TABLE tab ( k SERIAL) ; -- internal counter = 0
INSERT INTO tab VALUES ( 0 ) ; -- internal counter = 1
INSERT INTO tab VALUES ( 10 ) ; -- internal counter = 10
INSERT INTO tab VALUES ( 0 ) ; -- internal counter = 11
DELETE FROM tab ; -- internal counter = 11
INSERT INTO tab VALUES ( 0 ) ; -- internal counter = 12
```

**SAP ASE**

**SAP ASE** **IDENTITY** columns:

• When creating a table, the **IDENTITY** keyword must be specified after the column data type:

  ```sql
  CREATE TABLE tab1 ( k integer identity, c char(10) )
  ```

• You cannot specify a start value
• A new number is automatically created when inserting a new row:

  ```sql
  INSERT INTO tab1 ( c ) VALUES ( 'aaa' )
  ```

• To get the last generated number, **SAP ASE** provides a global variable:

  ```sql
  SELECT @@IDENTITY
  ```

• When **IDENTITY_INSERT=ON**, you can set a specific value into a **IDENTITY** column, but zero does not generate a new serial:

  ```sql
  SET IDENTITY_INSERT tab1 ON
  INSERT INTO tab1 ( k, c ) VALUES ( 100, 'aaa' )
  ```

Informix® **SERIALs** and **SAP ASE** **IDENTITY** columns are quite similar; the main difference is that **SAP ASE** does not generate a new serial when you specify a zero value for the identity column.

**Solution**

**Note:** For best SQL portability when using different types of databases, consider using sequences as described in [Solution 3: Use native SEQUENCE database objects](#) on page 628.

With **SAP ASE**, the **SERIAL** emulation can use **IDENTITY** columns (1) or insert triggers based on the **SERIALREG** table (2). The first solution is faster, but does not allow explicit serial value specification in insert statements; the second solution is slower but allows explicit serial value specification. You can initially use the second solution to have unmodified BDL programs working on **SAP ASE**, but it is recommended that you update your code to use native **IDENTITY** columns for performance.
The method used to emulate SERIAL types is defined by the ifxemul.datatype.serial.emulation FGLPROFILE parameter:

```sql
<table>
<thead>
<tr>
<th>dbi.database.dbname.ifxemul.datatype.serial.emulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;native&quot;</td>
</tr>
</tbody>
</table>
```

- **native**: uses IDENTITY columns.
- **regtable**: uses insert triggers with the SERIALREG table.

The default emulation technique is "native".

The serial types emulation can be enabled or disabled with the following FGLPROFILE entries:

```sql
|dbi.database.dbname.ifxemul.datatype.serial = \{true|false\}|
|dbi.database.dbname.ifxemul.datatype.serial8 = \{true|false\}|
|dbi.database.dbname.ifxemul.datatype.bigserial = \{true|false\}|
```

### Disabling automatic serial retrieval for SQLCA.SQLERRD[2]

SERIAL emulation can be totally disabled by setting the ifxemul.datatype.serial FGLPROFILE entry to false:

```sql
|dbi.database.dbname.ifxemul.datatype.serial = false|
```

For Informix® compatibility, after an INSERT statement, the ODI drivers automatically execute another SQL query (or do a DB client API call when possible), to get the last generated serial, and fill the SQLCA.SQLERRD[2] register. This results in some overhead that can be avoided, if the SQLCA.SQLERRD[2] register is not used by the program.

When SERIAL emulation is required (to create temp tables with a serial column during program execution), and the SQLCA.SQLERRD[2] register does not need to be filled, (typically because you use your own method to retrieve the last generated serial), you can set the ifxemul.datatype.serial.sqlerrd2 FGLPROFILE entry to false. This will avoid the automatic retrieval of last serial value to fill SQLCA.SQLERRD[2]:

```sql
|dbi.database.dbname.ifxemul.datatype.serial.sqlerrd2 = false|
```

See also `db_get_last_serial()` on page 2308.

### Using the native serial emulation

In database creation scripts, all SERIAL data types must be converted by hand to INTEGER IDENTITY data types, while BIGSERIAL must be converted to BIGINT IDENTITY.

Start values SERIAL(n) / BIGSERIAL(n) cannot be converted, there is no INTEGER IDENTITY(n) in SAP® ASE.

Tables created from the BDL programs can use the SERIAL data type: When a BDL program executes a CREATE [TEMP] TABLE with a SERIAL column, the database interface automatically converts the "SERIAL[(n)]" data type to "INTEGER IDENTITY[(n,1)]".

In BDL, the new generated SERIAL value is available from the SQLCA.SQLERRD[2] variable. This is supported by the database interface which performs a "SELECT @@IDENTITY". However, SQLCA.SQLERRD[2] is defined as an INTEGER, it cannot hold values from BIGINT identity columns. If you are using BIGINT IDENTITY columns, you must use `@@IDENTITY`.

When you insert a row with zero as serial value, the serial column gets the value zero. You must review all INSERT statements using zero for the serial column. For example, the following statement:

```sql
INSERT INTO tab (col1, col2)VALUES (0, p_value)
```
must be converted to:

```sql
INSERT INTO tab (col2) VALUES (p_value)
```

Static SQL INSERT using records defined from the schema file must also be reviewed:

```sql
DEFINE rec LIKE tab.*
INSERT INTO tab VALUES ( rec.*) -- will use the serial column
```

can be converted to:

```sql
INSERT INTO tab VALUES rec.* -- without parentheses, serial column is removed
```

**Using the regtable serial emulation**

First, you must prepare the database and create the SERIALREG table as follows:

```sql
CREATE TABLE serialreg (
    tablename VARCHAR(50) NOT NULL,
    lastserial BIGINT NOT NULL,
    PRIMARY KEY ( tablename )
)
```

In database creation scripts, all SERIAL[(n)] data types must be converted to INTEGER data types, BIGSERIAL column types must be changed to BIGINT, and you must create one trigger for each table. To know how to write those triggers, you can create a small Genero program that creates a table with a SERIAL column. Set the FGLSQLDEBUG environment variable and run the program. The debug output will show you the native trigger creation command.

Tables created from the BDL programs can use the SERIAL data type. When a BDL program executes a CREATE [TEMP] TABLE with a SERIAL column, the database interface automatically converts the "SERIAL[(n)]" data type to "INTEGER" and creates the insert triggers.

SAP® ASE does not allow you to create triggers on temporary tables. Therefore, you cannot create temp tables with a SERIAL column when using this solution.

**Note:**

- SELECT ... INTO TEMP statements using a table created with a SERIAL column do not automatically create the SERIAL triggers in the temporary table. The type of the column in the new table is INTEGER.
- SAP® ASE triggers are not automatically dropped when the corresponding table is dropped. Database administrators must be aware of this behavior when managing schemas.
- INSERT statements using NULL for the SERIAL column will produce a new serial value:
  ```sql
  INSERT INTO tab ( col1, col2 ) VALUES ( NULL, 'data' )
  ```

This behavior is mandatory in order to support INSERT statements which do not use the serial column:

```sql
INSERT INTO tab (col2) VALUES('data')
```

Check if your application uses tables with a SERIAL column that can contain a NULL value.

- The serial production is based on the SERIALREG table which registers the last generated number for each table. If you delete rows of this table, sequences will restart at 1 and you will get unexpected data.

**Related concepts**

- Auto-incremented columns (serials) on page 626
- How to implement automatic record keys.
- The SQLCA diagnostic record on page 594
The SQLCA variable is a predefined record containing SQL statement execution information.

**ROWID columns**

**Informix®**

When creating a table, Informix® automatically adds a ROWID integer column (applies to non-fragmented tables only).

The ROWID column is auto-filled with a unique number and can be used like a primary key to access a given row.

**Note:** Informix® ROWID usage was a common practice in the early days of Informix® 4GL programming. Today it is recommended to define all your database tables with a PRIMARY KEY to uniquely identify rows.

With Informix®, the SQLCA.SQLERRD[6] register contains the ROWID of the last row affected by an INSERT, UPDATE or DELETE statement.

**SAP ASE**

SAP® ASE does not support ROWIDs.

**Solution**

If your Genero BDL application uses rowid columns, review the program logic to use primary keys instead. If the database table does not define a primary key, it should be added. All references to SQLCA.SQLERRD[6] must be removed, because this variable will not hold the ROWID of the last modified row.

For databases where the keyword of the rowid pseudo-column is different than "ROWID", the translation can be controlled with the following FGLPROFILE entry:

```
dbi.database.dsname.ifxemul.rowid = ↓ true ↓ false ↓
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

- Using ROWID columns on page 644
- Automatic ROWID columns is not a common database feature.

**Case sensitivity**

**Informix®**

With Informix®, database object names like table and column names are not case sensitive:

```
CREATE TABLE Customer ( Custno INTEGER, ... )
SELECT CustNo FROM cuSTomer ...
```

**SAP ASE**

With SAP® ASE, database object names and character data are case-insensitive by default:

```
CREATE TABLE Customer ( Custno INTEGER, CustName CHAR(20) )
INSERT INTO CUSTOMER VALUES ( 1, 'TECHNOSOFT' )
SELECT CustNo FROM cuSTomer WHERE custname = 'techNOSoft'
```

**Solution**

When you create a SAP® ASE database with dbinit, you can use the -c option to make the database case-sensitive.
TEXT and BYTE (LOB) types

Informix®
Informix® provides the TEXT, BYTE, CLOB and BLOB data types to store very large texts or binary data.
Legacy Informix® 4GL applications typically use the TEXT and BYTE types.
Genero BDL does not support the Informix® CLOB and BLOB types.

SAP ASE
SAP® ASE provides the TEXT and IMAGE data types for large objects storage.
SAP® ASE 16.0 does not support TEXT/IMAGE expressions in WHERE clauses.

Solution
TEXT and BYTE data types are supported by the SAP® ASE database interface, with some limitation.

Important: The SAP® ASE ODI driver is implemented with the SAP Open Client Library C API. In SAP® ASE version 16.0, this API has limited support for LOBs, especially when it comes to update LOB data in the database. You cannot directly INSERT large LOB data, you must first INSERT nulls and then UPDATE the row with the real data. Additionally, UPDATE can only take one LOB parameter at a time. Fetching LOB data is supported, with the following limitation: LOB columns must appear at the end of the SELECT list.

When inserting TEXT/BYTE data in a table, you must first insert with nulls, the update the new row, and only with one TEXT/BYTE parameter at a time:

```
DEFINE ptext TEXT, pbyte BYTE
...
LOCATE ptext IN ...
LOCATE pbyte IN ...
CREATE TABLE tab (k INT, t TEXT, b BYTE)
-- First INSERT a new row with NULLs
INSERT INTO tab VALUES (123,null,null)
-- Then UPDATE first TEXT column
UPDATE tab SET t = ptext WHERE k = 123
-- Then UPDATE second BYTE column
UPDATE tab SET b = pbyte WHERE k = 123
```

Fetching TEXT/BYTE columns is possible as long as the columns appear at the end of the SELECT list. For example, if you have a statement such as (where pdata is a TEXT or BYTE column):

```
SELECT pid, pdata, ptimestamp FROM pic WHERE ...
```

Put the TEXT/BYTE column at the end of the SELECT list:

```
SELECT pid, ptimestamp, pdata FROM pic WHERE ...
```

The TEXT and BYTE types translation can be controlled with the following FGLPROFILE entries:

```
  dbi.database.dsname.ifxemul.text = { true | false }
  dbi.database.dsname.ifxemul.byte = { true | false }
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

Related concepts
Using portable data types on page 614
Only a limited set of data types are really portable across several database engines.

**Table constraints**

**Informix**

Informix supports primary key, unique, foreign key, default and check constraints. The constraint naming syntax is different in Informix and most other databases: Informix expects the constraint name after the constraint definition:

```sql
CREATE TABLE emp (  
    ...  
    emp_code CHAR(10) UNIQUE CONSTRAINT pk_emp,  
    ...  
)
```

While other SQL database brands require to specify the constraint name before the constraint definition:

```sql
CREATE TABLE emp (  
    ...  
    emp_code CHAR(10) CONSTRAINT pk_emp UNIQUE,  
    ...  
)
```

**SAP ASE**

SAP ASE supports primary key, unique, foreign key, default and check constraints.

**Constraint naming**

However, SAP ASE does not support a constraint naming clause.

**Solution**

The database interface does not convert constraint naming expressions when creating tables from BDL programs. Review the database creation scripts to adapt the constraint naming clauses for SAP ASE.

**Related concepts**

Data definition statements on page 613

It is recommended to avoid use of DDL in programs.

**Name resolution of SQL objects**

**Informix**

Informix uses the following form to identify an SQL object:

```sql
database[@dbservername]:][{owner|"owner"}.]identifier
```

The ANSI convention is to use double quotes for identifier delimiters (For example: "customer"."cust_name").

Informix database object names are not case-sensitive in non-ANSI databases. When using double-quoted identifiers, Informix becomes case sensitive.

With non-ANSI Informix databases, you do not have to give a schema name before the tables when executing an SQL statement:

```sql
SELECT ... FROM customer WHERE ...
```
In Informix® ANSI compliant databases:

- The table name must include "owner", unless the connected user is the owner of the database object.
- The database server shifts the owner name to uppercase letters before the statement executes, unless the owner name is enclosed in double quotes.

**SAP ASE**

With SAP® ASE, an object name takes the following form:

```
[database][database].[owner][owner].[identifier][identifier]
```

Informix® database object names are not case sensitive in non-ANSI databases.

SAP® ASE database objects names are case sensitive by default.

**Solution**

To write portable SQL, regarding database object names:

1. Use simple database object names (without any owner/schema prefix)
2. Do not use double quotes to surround database object identifiers.
3. If needed, define public synonyms to reference database objects in others databases/schema.
4. Specify database object identifiers in lowercase.

See also Naming database objects on page 637.

**Data manipulation**

SAP® ASE related data manipulation topics.

**Reserved words**

**Informix®**

With Informix®, it is possible to create database objects with reserved words.

For example:

```
CREATE TABLE table ( char CHAR(10) );
```

Indeed this is not good practice, but Informix® SQL allows this to be backward compatible when introducing a new keyword in the SQL syntax.

Most other database systems do not allow reserved words as database identifiers. If your legacy code is using SQL reserved words of the target database SQL syntax, an error will be thrown at CREATE TABLE execution.

**SAP ASE**

Even if SAP® ASE allows SQL reserved keywords as SQL object names even if enclosed in square brakets (create table [table] ( coll int )), verify this with your existing database schema and make sure that you do not use SAP® ASE SQL words.

**Solution**

Database objects having names which are SAP® ASE SQL reserved words must be renamed.

All BDL application sources must be verified. To check if a given keyword is used in a source, you can use UNIX™ `grep` or `awk` tools. Most modifications can be automatically done with UNIX™ tools like `sed` or `awk`. 
Outer joins

**Informix® OUTER() syntax**

In Informix® SQL, outer joins can be defined in the FROM clause with the OUTER keyword:

```sql
SELECT ... FROM a, OUTER (b)
    WHERE a.key = b.akey

SELECT ... FROM a, OUTER(b,OUTER(c))
    WHERE a.key = b.akey
    AND b.key1 = c.bkey1 AND b.key2 = c.bkey2
```

Informix® also supports the ANSI OUTER join syntax, which is the recommended way to specify outer joins with recent SQL database engines:

```sql
SELECT ... FROM cust LEFT OUTER JOIN order
    ON cust.key = order.custno
```

**SAP ASE**

SAP® ASE supports the ANSI outer join syntax:

```sql
SELECT ...
FROM cust LEFT OUTER JOIN order
    LEFT OUTER JOIN item
    ON order.key = item.ordno
    ON cust.key = order.custno
WHERE order.cdate > current date
```

Note:

The legacy syntax to specify outer join in SAP® ASE uses the *\= notation:

```sql
SELECT ... FROM a, b WHERE a.key *= b.key
```

**Solution**

The Genero database drivers can convert Informix Informix® OUTER specifications to ANSI outer joins.

**Note:** For better SQL portability, use the ANSI outer join syntax instead of the old Informix® OUTER syntax.

The outer join translation can be controlled with the following FGLPROFILE entry:

```ini
dbi.database.dsname.ifxemul.outers = true | false
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

- **Prerequisites:**

  1. In the FROM clause, the main table must be the first item and the outer tables must be listed from left to right in the order of outer levels.

     Example which does not work:

     ```sql
     ... FROM OUTER(tab2), tab1
     ```

  2. The outer join in the WHERE clause must use the table name as prefix:

     ```sql
     ... WHERE tab1.col1 = tab2.col2
     ```
• **Restrictions:**
  1. Statements composed by 2 or more SELECT instructions are not supported:

     ```sql
     SELECT ... UNION SELECT ...
     ```

     or:

     ```sql
     SELECT ... WHERE col IN (SELECT...)
     ```

  2. Additional conditions on outer table columns cannot be detected and therefore are not supported:

     ```sql
     ... FROM tab1, OUTER(tab2)
     WHERE tab1.col1 = tab2.col2
     AND tab2.colx > 10
     ```

  3. Using subscript in outer conditions:

     ```sql
     ... FROM tab1, OUTER(tab2)
     WHERE tab1.col1[1,3] = tab2.col2[1,3]
     ```

• **Notes:**
  1. Table aliases are detected in OUTER expressions.

     OUTER example with table alias:

     ```sql
     ... OUTER(tab alias1) ...
     ```

  2. In the outer join, `outertab.col` can be placed on both right or left sides of the equal sign:

     ```sql
     ... WHERE outertab.col1 = maintab.col2
     ```

  3. Table names detection is not case-sensitive:

     ```sql
     SELECT ... FROM tab1, TAB2
     WHERE tab1.col1 = tab2.col2
     ```

  4. **Temporary tables** are supported in OUTER specifications:

     ```sql
     CREATE TEMP TABLE tt1 ( ... )
     SELECT ... FROM tab1, OUTER(tt1) ...
     ```

**Related concepts**

- **Outer joins** on page 643

  Use standard ISO outer join syntax instead of the old IBM® Informix® `OUTER()` syntax.

**Transactions handling**

**Informix®**

With the Informix® native mode (non ANSI):

- Transactions blocks start with `BEGIN WORK` and terminate with `COMMIT WORK` or `ROLLBACK WORK`.
- Statements executed outside a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.

```sql
UPDATE tab1 SET ...  -- auto-committed
BEGIN WORK            -- start of TX block
UPDATE tab1 SET ...
UPDATE tab2 SET ...
...```
Informix® version 11.50 introduces savepoints:

```
SAVEPOINT name [UNIQUE]
ROLLBACK [WORK] TO SAVEPOINT _name_  
RELEASE SAVEPOINT name
```

### SAP ASE

- SAP® ASE supports two transaction modes:
  1. The SQL standards-compatible mode, called chained mode, to get implicit transaction.
  2. The default mode, called unchained mode, where transactions have to be started/ended explicitly.
- Transactions are started with "BEGIN TRANSACTION [name]".
- Transactions are validated with "COMMIT TRANSACTION [name]".
- Transactions are canceled with "ROLLBACK TRANSACTION [name]".
- Transactions save points can be placed with "SAVEPOINT [name]".
- SAP® ASE supports named and nested transactions.
- DDL statements can be executed in transactions blocks when the 'ddl in tran' option is set to true with:

```
master..sp_dboption dbname, 'ddl in tran', true
go
checkpoint
go
```

### Solution

Informix® transaction handling commands are automatically converted to SAP® ASE instructions to start, commit or rollback transactions.

Make sure that the database uses the default unchained mode (set chained off) and allows DDLs in transactions ('ddl in tran' option is true).

Regarding the transaction control instructions, the BDL applications do not have to be modified in order to work with SAP® ASE.

### Related concepts

[Database transactions](#) on page 600

Database transactions define a set of SQL instructions to be executed as a whole, or rolled back as a whole.

### Temporary tables

**Informix®**

Informix® temporary tables are created with the `CREATE TEMP TABLE` DDL instruction or with `SELECT ... INTO TEMP` statement:

```
CREATE TEMP TABLE tt1 ( pkey INT, name VARCHAR(50) )
CREATE TEMP TABLE tt2 ( pkey INT, name VARCHAR(50) ) WITH NO LOG
SELECT * FROM tab1 WHERE pkey > 100 INTO TEMP tt2
```

Temporary tables are automatically dropped when the SQL session ends, but they can also be dropped with the DROP TABLE command. There is no name conflict when several users create temporary tables with the same name.

BDL REPORTs can create a temporary table when the rows are not sorted externally (by the source SQL statement).

Informix® allows you to create indexes on temporary tables. No name conflict occurs when several users create an index on a temporary table by using the same index identifier.
When creating temporary tables in Informix®, the WITH NO LOG clause can be used to avoid the overhead of recording DML operations in transaction logs.

**SAP ASE**

SAP® ASE supports temporary tables by using the # pound sign before the table name:

```sql
CREATE TABL #temp1 ( kcol INTEGER, .... )
SELECT * INTO #temp2 FROM customers WHERE ... 
```

**Solution**

In BDL, Informix® temporary tables instructions are converted to generate native SAP® ASE temporary tables.

SELECT INTO TEMP statements cannot be converted, because SAP® ASE does not provide a way to create a temporary table from a result set, such as CREATE TABLE xx AS (SELECT ... ).

The general FGLPROFILE entry to control temporary table emulation is:

```
dbi.database.dsname.ifxemul.temptables = [ true | false ]
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

Temporary tables on page 641

Syntax for temporary table creation is not unique across all database engines.

**Substrings in SQL**

**Informix®**

Informix® SQL statements can use subscripts on columns defined with the character data type:

```sql
SELECT ... FROM tab1 WHERE col1[2,3] = 'RO'
SELECT ... FROM tab1 WHERE col1[10] = 'R' -- Same as col1[10,10]
UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...
SELECT ... FROM tab1 ORDER BY col1[1,3]
```

**Important**: With other database servers as Informix®, when the subscript notation is used to modify column values in UPDATE statement, or as ORDER BY element, you will get and SQL error:

```sql
UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...
SELECT ... FROM tab1 ORDER BY col1[1,3]
```

**SAP ASE**

SAP® ASE provides the SUBSTRING() function, to extract a substring from a string expression:

```sql
SELECT .... FROM tab1 WHERE SUBSTRING(col1,2,2) = 'RO'
SELECT SUBSTRING('Some text',6,3 ) FROM DUAL -- Gives 'tex'
```

**Solution**

Replace all Informix® col[x,y] right-value expressions by SUBSTRING( col from x for (y-x+1) ).

Rewrite UPDATE and ORDER BY clauses using col[x,y] expressions.
The translation of \( \text{col}[x,y] \) expressions can be controlled with the following FGLPROFILE entry:

\[
\text{dbi.database.dsnname.ifxemul.colsubs = \{ true \ false }\]

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

Related concepts
Substring expressions on page 644
Handle substrings expressions with different database engines.

String delimiters

Informix®

The ANSI SQL string delimiter character is the single quote ('string'), while double quotes are used to delimit database object names:

\[
\text{SELECT ... WHERE "tabname"."colname" = 'a string value'}\]

In Informix® databases created in native mode (non-ANSI), you can use double quotes as string delimiters:

\[
\text{SELECT ... WHERE tabname.colname = 'a string value'}\]

This is important, since many BDL programs use that character to delimit the strings in SQL commands.

Note: This problem concerns only double quotes within SQL statements. Double quotes used in pure BDL string expressions are not subject to SQL compatibility problems.

SAP ASE

SAP® ASE allows to use double quotes as string delimiters, if the QUOTED_IDENTIFIER session option is OFF (the default):

\[
\text{SET QUOTED_IDENTIFIER OFF}\]

Solution

When using Static SQL statements, the fglcomp compiler converts string literals using double quotes to string literals with single quotes:

\[
\text{$ cat s.4gl\nMAIN\n \hspace{1cm} \text{DEFINE n INT}\n \hspace{1cm} \text{SELECT COUNT(*) INTO n FROM tab1 WHERE col1 = "abc"}}\]
\text{END MAIN}$
\text{\hspace{1cm} $ fglcomp -S s.4gl\n s.4gl^3^SELECT COUNT(*) FROM tab1 WHERE col1 = 'abc'$}

However, SQL statements created dynamically are not modified by the Genero compiler.

The Genero database interface can automatically replace all double quotes by single quotes in SQL statements. This applies to static and dynamic SQL statements.

The The translation of double quoted expression to single quoted expressions can be controlled with the following FGLPROFILE entry:

\[
\text{dbi.database.dbname.ifxemul.dblquotes = \{ true \ false }\]

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.
However, database object names must not be delimited by double quotes, because the database interface cannot determine the difference between a database object name and a quoted string! For example, if the program executes the SQL statement:

```sql
... WHERE "tabname"."colname" = "a string value"
```

replacing all double quotes by single quotes would produce:

```sql
... WHERE 'tabname'.'colname' = 'a string value'
```

This would produce an error since 'tabname'.colname' is not allowed by ORACLE.

Escaped string delimiters can be used inside strings like the following:

```sql
'This is a single quote: '''
'This is a single quote: \\
"This is a double quote: '"'
"This is a double quote: \"
```

Although double quotes are replaced automatically in SQL statements, it is recommended that you use only single quotes to enforce portability.

**Related concepts**

- [String literals in SQL statements](#) on page 634
  
Single quotes is the standard for delimiting string literals in SQL.

**Single row SELECT**

**Informix®**

With Informix®, you must use the system table with a condition on the table id:

```sql
SELECT user FROM systables WHERE tabid=1
```

**SAP ASE**

With SAP® ASE, you can omit the FROM clause to generate one row only:

```sql
SELECT user
```

**Solution**

Check the BDL sources for "FROM systables WHERE tabid=1" and use dynamic SQL to resolve this problem.

Consider writing a FUNCTION which produces the FROM and WHERE part, depending on the target database type.

**MATCHES and LIKE**

**Informix®**

Informix® supports MATCHES and LIKE operators in SQL statements.

MATCHES expects * and ? wild-card characters, while LIKE uses the % and _ wild-cards as equivalents.

```sql
( col MATCHES 'Smi*' AND col NOT MATCHES 'R?x' )
( col LIKE 'Smi%' AND col NOT LIKE 'R_x' )
```
MATCHES accepts also brackets notation, to specify a set of matching characters at a given position:

```
( col MATCHES '[Pp]aris' )
( col MATCHES '[0-9][a-z]*' )
```

**SAP ASE**

SAP® ASE does not provide an equivalent of the Informix® MATCHES operator. The LIKE operator is supported.

**Solution**

The database driver is able to translate Informix® MATCHES expressions to LIKE expressions, when no [ ] bracket character ranges are used in the MATCHES operand.

The MATCHES to LIKE expression translation is controlled by the following FGLPROFILE entry:

```
 dbi.database.dbname.ifxemul.matches = [true | false ]
```

**Important:** Only [NOT] MATCHES followed by a search pattern provided as a string literal can be converted by ODI drivers. A [NOT] MATCHES followed by a ? question mark parameter place holder is not translated!

For maximum portability, consider replacing the MATCHES expressions with LIKE expressions in all SQL statements.

Avoid using CHAR(N) types for variable length character data (such as name, address).

**Related concepts**

MATCHES and LIKE operators on page 645

Use the standard LIKE operator instead of the MATCHES operator.

**The LENGTH() function**

**Informix®**

Informix® provides the LENGTH() function to count the number of bytes of a character string expression:

```
SELECT LENGTH("aaa"), LENGTH(col1) FROM table
```

Informix® LENGTH() does not count the trailing blanks for CHAR or VARCHAR expressions, while Oracle counts the trailing blanks.

Informix® LENGTH() returns 0 when the given string is empty. That means, LENGTH('') = 0.

**SAP ASE**

SAP® ASE supports the LEN() function, which is similar to Informix® LENGTH() .

The SAP ASE LEN() function ignores trailing blanks.

When passing NULL as parameter, the SAP ASE LEN() function returns NULL.

**Solution**

The database driver is able to replace LENGTH() by the LEN() function name.

The translation of LENGTH() expressions can be controlled with the following FGLPROFILE entry:

```
 dbi.database.dsname.ifxemul.length = [true | false ]
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.
Related concepts

The LENGTH() function in SQL on page 647

The semantics of the LENGTH() SQL function differs according to the database engine.

Row limiting clause

Informix®

Informix® SQL supports the SKIP and FIRST/LIMIT keywords to limit the number of rows of a result set:

```
SELECT SKIP 10 FIRST 20 customer.* FROM customer ... ORDER BY cust_name
```

This Informix SQL syntax is not portable.

Recent database engines support the row limiting clause syntax defined by the SQL standard:

```
SELECT ... OFFSET n ROWS FETCH FIRST m ROWS ONLY
```

This should be the preferred syntax to be used, if all target database types support this SELECT clause.

The ODI database drivers can convert the Informix SQL SKIP/FIRST row limiting clause to a native SQL equivalent, if the row limiting clause parameters are simple integer literals (the clause is not translated when using SQL parameters / program variables).

**Important:** In addition to the SKIP/FIRST clause of the projection clause, Informix SQL supports also a LIMIT clause after the ORDER BY clause:

```
SELECT customer.* FROM customer ... ORDER BY cust_name LIMIT 10
```

This Informix SQL syntax construction is not converted by the ODI drivers. To benefit from the conversion, review the code to use the Informix SQL SKIP/FIRST clause instead.

SAP ASE

SAP® ASE supports the following row limiting clause:

```
SELECT TOP (m) ... FROM ...
```

Solution

**Important:** The Informix SQL row limiting clause is not converted by the SAP ASE driver, because the native SQL syntax does not support an offset option.

Related concepts

Row limiting clause (SELECT) on page 650

How to use the right clause to limit the number of rows produced by a SELECT statement?

BDL programming

SAP® ASE related programming topics.

INSERT cursors

Informix®

Informix® provides insert cursors to optimize row creation in a database. An insert cursor is declared as a cursor, and rows as added with the PUT instruction. The rows are buffered and sent to the database server when executing a FLUSH instruction, or when the cursor is closed with CLOSE. When using transactions in Informix®, the OPEN, PUT and FLUSH instructions must be executed within a transaction block.

```
DECLARE c1 CURSOR FOR INSERT INTO tab1 ...
```
BEGIN WORK
OPEN c1
WHILE ...
    PUT c1 USING var-list
END WHILE
CLOSE c1
COMMIT WORK

SAP ASE

SAP® ASE does not support insert cursors.

Solution

Insert cursors are emulated by the database interface, using basic INSERT SQL instructions.
The performances might be not as good as with Informix®, but the feature is fully supported.

Related concepts
Insert cursors on page 633
Using insert cursors with non-Informix databases.

Cursors WITH HOLD

Informix®

Informix® closes opened cursors automatically when a transaction ends, unless the WITH HOLD option is used in the DECLARE instruction:

DECLARE c1 CURSOR WITH HOLD FOR SELECT ...
OPEN c1
BEGIN WORK
FETCH c1 ...
COMMIT WORK
FETCH c1 ...
CLOSE c1

SAP ASE

By default, SAP® ASE does not close cursors when a transaction ends.

See the SAP ASE manual for more details about cursor behaviors and configuration settings to control the behavior of cursors.

Solution

BDL cursors declared WITH HOLD remain open even after terminating a transaction with a COMMIT WORK or ROLLBACK WORK.

For consistency with other database brands, database cursors that are not declared WITH HOLD are automatically closed, when a COMMIT WORK or ROLLBACK WORK is performed.

Important: Opening a WITH HOLD cursor declared with a SELECT FOR UPDATE results in an SQL error; in the same conditions, this does not normally appear with Informix®. Review the program logic in order to find another way to set locks.

Related concepts
Cursors WITH HOLD on page 631
Programming WITH HOLD cursors using SELECT with and without FOR UPDATE clause.

SELECT ... FOR UPDATE

Informix®

Legacy BDL programs typically use a cursor with SELECT FOR UPDATE to implement pessimistic locking and avoid several users editing the same rows:

```sql
DECLARE cc CURSOR FOR
SELECT ... FROM tab WHERE ... FOR UPDATE
OPEN cc
FETCH cc <-- lock is acquired
...
CLOSE cc <-- lock is released
```

The row must be fetched in order to set the lock.

If the cursor is local to a transaction, the lock is released when the transaction ends. If the cursor is declared WITH HOLD, the lock is released when the cursor is closed.

Informix® provides the SET LOCK MODE instruction to define the lock wait timeout:

```sql
SET LOCK MODE TO \{ WAIT | NOT WAIT | WAIT seconds \}
```

The default mode is NOT WAIT.

SAP® ASE

SAP® ASE supports SELECT ... FOR UPDATE clause in the context of a cursor. This can be achieved by using the CS_FOR_UPDATE option in ct-lib Client-Library `ct_cursor()` function.

**Note:** The "select for update" database configuration parameter is by default set to zero. This parameter defines if a single SELECT ... FOR UPDATE (at the SQL language level) must set locks. This parameter does not need to be 1 when executing SELECT ... FOR UPDATE in the context of a SAP ASE ct-lib cursor (`ct_cursor()` + CS_FOR_UPDATE).

Rows selected with SELECT ... FOR UPDATE, within or outside of a cursor context, retain an exclusive lock until the transaction is complete.

SAP® ASE's locking granularity is at the row level, page level or table level (the level is automatically selected by the engine for optimization, depending on the locking scheme).

**Solution**

SELECT FOR UPDATE statements are supported with SAP® ASE under some conditions:

SAP® ASE requires a PRIMARY KEY or UNIQUE INDEX on the table using in the SELECT ... FOR UPDATE statement.

When the SAP® ASE database driver must execute a SELECT FOR UPDATE, the ct-lib `ct_cursor()` function is called with the CS_FOR_UPDATE option.

Locks are acquired when fetching rows with the cursor. Locks are released when the transaction ends or when the cursor is closed.

The database interface is based on an emulation of an Informix® engine using transaction logging. Therefore, opening a SELECT ... FOR UPDATE cursor declared outside a transaction will raise an SQL error -255 (not in transaction).

**Related concepts**

Cursors WITH HOLD on page 631
Programming WITH HOLD cursors using SELECT with and without FOR UPDATE clause.

**UPDATE/DELETE ... WHERE CURRENT OF**

**Informix**

Informix® allows positioned UPDATEs and DELETEs with the "WHERE CURRENT OF cursor" clause, if the cursor has been DECLARED with a SELECT ... FOR UPDATE statement.

**SAP® ASE**

UPDATE/DELETE ... WHERE CURRENT OF is supported with SAP® ASE by using server side cursors.

**Solution**

With SAP® ASE, UPDATE/DELETE ... WHERE CURRENT OF instructions are executed as is without any SQL translation: Since BDL cursors for the SELECT FOR UPDATE statements are implemented with ODBC server cursors, native positioned update/delete can take place in SQL SERVER.

As a replacement of WHERE CURRENT OF, if the database table is defined with a primary key column, use the value fetched from the SELECT FOR UPDATE cursor in the WHERE clause of the UPDATE/DELETE statement.

**Related concepts**

Positioned UPDATE/DELETE on page 631
Using positioned updates/deletes with named database cursors.

**LOAD and UNLOAD**

**Informix**

Informix® provides two SQL instructions to export / import data from / into a database table:

The UNLOAD instruction copies rows from a database table into a text file:

```
UNLOAD TO "filename.unl" SELECT * FROM tab1 WHERE ..
```

The LOAD instructions insert rows from a text file into a database table:

```
LOAD FROM "filename.unl" INSERT INTO tab1
```

**SAP ASE**

SAP® ASE has LOAD and UNLOAD instructions, but those commands are related to database backup and recovery. Do not confuse with Informix® commands.

**Solution**

LOAD and UNLOAD instruction are implemented in the Genero BDL runtime system with basic INSERT (for LOAD) or SELECT (for UNLOAD) SQL commands. The LOAD and UNLOAD instruction can be supported with various database servers.

However, LOAD and UNLOAD require the description of the column types in order to work, that can lead to some differences in the data formatting.

**Note:** If no transaction is started, the LOAD instruction will automatically execute a BEGIN WORK and COMMIT WORK when finished, or ROLLBACK WORK if a row insertion failed while loading. Terminating a transaction will automatically close cursors not defined WITH HOLD option. To workaround this situation, see more details in the LOAD on page 736 reference topic.

The LOAD and UNLOAD BDL instructions are supported with SAP® ASE with some limitations:
• The `LOAD` instruction does not work with tables using emulated `SERIAL` columns because the generated `INSERT` statement holds the `SERIAL` column which is actually a `IDENTITY` column in SAP® ASE. See the limitations of `INSERT` statements when using `SERIAL` types.

• With SAP® ASE, Informix® `DATETIME` data is stored in `BIGDATETIME` columns, but `BIGDATETIME` columns are similar to Informix® `DATETIME YEAR TO FRACTION(5)` columns. Therefore, when using `LOAD` and `UNLOAD`, those column values are converted to text data with the format "YYYY-MM-DD hh:mm:ss.fffffff".

Related concepts
LOAD and UNLOAD instructions on page 639
The `LOAD` and `UNLOAD` instructions can produce different data formats depending on the database server type.

SQL Interruption

Informix®
With Informix®, it is possible to interrupt a long running query if the SQL INTERRUPT ON option.

SAP ASE
SAP® ASE supports SQL Interruption: The db client must issue an `ct_cancel()` call to interrupt a query.

Solution
The SAP® ASE database driver supports SQL interruption and raises error code -213 if the statement is interrupted.

Related concepts
Using SQL interruption on page 601
Interrupt long running SQL queries, or interrupt queries waiting for locked data.

Scrollable cursors

Informix®
Informix® SQL and Genero BDL support scrollable cursors when you specify the `SCROLL` clause in the `DECLARE` cursor instruction:

```
DECLARE c1 SCROLL CURSOR FOR SELECT ...  
```

Important: Informix does not allow to fetch `TEXT`/`BYTE` columns with scrollable cursors. If you declare a scroll cursor with a `SELECT` containing `TEXT`/`BYTE` columns, Informix will produce the SQL error -611 when executing the `OPEN` instruction.

SAP ASE
SAP® ASE supports native scrollable cursors.

Solution
The SAP® ASE database driver uses the native SAP® ASE Open Client Library scrollable cursors.

Important: With SAP ASE is it NOT possible to use LOB columns in a scrollable cursor. If `TEXT`/`BYTE` columns are used with a scrollable cursor, the `OPEN` instruction will produce the SQL error -611 (as with Informix).

Related concepts
Scrollable cursors on page 623
How scrollable cursors can be supported on different databases.

Stored procedure calls
SAP® ASE supports stored procedures, which can take and return user-supplied parameters.
SAP® ASE stored procedures can also produce one or more result sets.

To create a stored procedure in a SAP® ASE database, use the `CREATE PROCEDURE` statement. See reference documentation for more details.

**Stored procedures with output parameters**

SAP® ASE stored procedures must be called with the input and output parameters specification in the `USING` clause of the `EXECUTE`, `OPEN` or `FOREACH` instruction. As in normal dynamic SQL, parameters must correspond by position and the `IN/OUT/INOUT` options must match the parameter definition of the stored procedure.

To execute the stored procedure, you must use a specific syntax to have the database driver identify the statement as an RPC call. The syntax of an RPC call must be:

```sql
!rpc procedure-name ( [ @param-name [,...] ] )
```

The parameter names must be specified, with the same names as the arguments of the stored procedure, because the ODI driver must bind stored procedure parameters by name.

Example:

```sql
PREPARE stmt FROM "!rpc update_account ( @custid, @old, @new )"
```

Here is a complete example creating and calling a stored procedure with output parameters:

```sql
MAIN
DEFINE n INTEGER
DEFINE d DECIMAL(6,2)
DEFINE c VARCHAR(200)
DATABASE test1
EXECUTE IMMEDIATE "create procedure proc1"
  " @v1 integer,"      
  " @v2 decimal(6,2) output,"  
  " @v3 varchar(20) output"
  " as begin"
  " set @v2 = @v1 + 0.23"
  " set @v3 = 'Value = ' || cast(@v1 as varchar)"
"end"
PREPARE stmt FROM "!rpc proc1( @v1, @v2, @v3 )"
LET n = 111
EXECUTE stmt USING n IN, d OUT, c OUT
DISPLAY d
DISPLAY c
END MAIN
```

**Stored procedures with result set**

With SAP ASE, you can execute stored procedures returning a result set. To do so, you must declare a cursor and fetch the rows.

When the stored procedure generates multiple active statements, you cannot use another SQL statement while processing the results of such stored procedure. You must `CLOSE` the cursor created for the stored procedure before continuing with other SQL statements.

```sql
MAIN
DEFINE i, n INTEGER
DEFINE d DECIMAL(6,2)
DEFINE c VARCHAR(200)
DATABASE test1
CREATE TABLE tab1 ( c1 INTEGER, c2 DECIMAL(6,2), c3 VARCHAR(200) )
INSERT INTO tab1 VALUES ( 1, 123.45, 'aaaaaa' )
INSERT INTO tab1 VALUES ( 2, 123.66, 'bbbbbbbb' )
```
| SQL support | 1069 |

```
INSERT INTO tab1 VALUES ( 3, 444.77, 'cccccc' )
EXECUTE IMMEDIATE "create procedure proc2 @key integer"
  as select * from tab1 where c1 > @key"
DECLARE curs CURSOR FROM "!rpc proc2( @key )"
LET i = 1
FOREACH curs USING i INTO n, d, c
  DISPLAY n, d, c
END FOREACH
END MAIN
```

**Stored procedures with output parameters, return code and result set**

With SAP® ASE stored procedures, you call stored procedures with a return code, output parameters and producing a result set.

Return codes and output parameters are the last items returned to the application by SAP® ASE; they are not returned until the last row of the result set has been fetched.

```
MAIN
  DEFINE r, i, n INTEGER
  DEFINE d DECIMAL(6,2)
  DEFINE c VARCHAR(200)
  DATABASE test1
  CREATE TABLE tab1 ( c1 INTEGER, c2 DECIMAL(6,2), c3 VARCHAR(200) )
  INSERT INTO tab1 VALUES ( 1, 123.45, 'aaaaaa' )
  INSERT INTO tab1 VALUES ( 2, 123.66, 'bbbbbbbb' )
  INSERT INTO tab1 VALUES ( 3, 444.77, 'cccccc' )
  EXECUTE IMMEDIATE "create procedure proc3 @key integer output"
    " as begin"
    "   set @key = @key - 1"
    "   select * from tab1 where c1 > @key"
    "   return (@key * 3)"
    " end"

  DECLARE curs CURSOR FROM "!rpc proc3( @key )"
  LET i = 1
  OPEN curs USING r OUT, i OUT
  DISPLAY r, i
  FETCH curs INTO n, d, c
  FETCH curs INTO n, d, c
  FETCH curs INTO n, d, c
  DISPLAY r, i
  CLOSE curs
  DISPLAY r, i -- Now the returned values are available
END MAIN
```

**SAP HANA®**

**Supported versions**

Genero BDL supports the following SAP HANA® versions:

- SAP HANA 2.0

**Installation (Runtime Configuration)**

SAP HANA® related installation topics.

**Install SAP HANA® and create a database - database configuration/design tasks**

If you are tasked with installing and configuring the database, here is a list of steps to be taken:

1. Install the SAP HANA® database server on your computer.

   The installation program requires you to provide a strong password for system users. Keep this password in mind for future usage.
2. Understand basic SAP HANA® server administration tasks (command line).
   • The configuration files nameserver.ini and the indexserver.ini can be edited by the OS administrator user (root).
   • The default SAP HANA® admin OS user is <sid>adm. For example, with SAP HANA express edition it is hxeadm.
   • To check if the SAP HANA® server is up, use: the HDB info command.
   • To start/stop the SAP HANA® server, use HDB start/HDB stop commands.
   • If needed, configure the host firewall for remote connections. For example, on UNIX/Linux, check TCP ports to be opened with the netstat -tnlp command.

3. Create a SAP HANA® database entity: dbname
   Connect as the SYSTEM user to the SystemDB database and create the application database:
   Note: In next examples the SAP HANA instance number is 90.
   
   $ hdbsql -i 90 -d SystemDB -u SYSTEM -p system-password
   ...> CREATE DATABASE dbname SYSTEM USER PASSWORD password;
   ...> \q

4. Create the DB user to be the owner of application database tables.
   Connect to the new created database with the SYSTEM user, and create the application user, and grant the required privileges to create tables:
   
   $ hdbsql -i 90 -d dbname -u SYSTEM -p system-password
   ...> CREATE USER appadmin PASSWORD password;
   ...> GRANT USER ADMIN TO appadmin WITH ADMIN OPTION;
   ...> \q

5. Create the application tables with CREATE TABLE statements.
   
   Connect to the application database as the application administrator and create the tables:
   
   $ hdbsql -i 90 -d dbname -u appadmin -p password
   ...> CREATE TABLE customer ( ... ) ;
   ...

   Convert Informix® data types to SAP HANA® data types. See issue Data Type Conversion Table for more details.

6. If you plan to use SERIAL column emulation, you must prepare the database.
   See SERIAL and BIGSERIAL data types on page 1084.

Prepare the runtime environment - connecting to the database

1. In order to connect to SAP HANA®, you must have the database driver "dbmhdb" in $FGLDIR/dbdrivers.
2. Make sure that the SAP HANA® client software is installed and check that SAP HANA environment variables are properly set.
   See SAP HANA® documentation for more details.
3. The database client locale does not need to be defined with the SAP HANA® ODI driver: The driver will automatically do the required code set conversions from/to the application client locale and UTF-16LE, with the wide char (SQL_C_WCHAR) ODBC APIs.
   Only define the application locale for the runtime system (LC_ALL, LANG).
4. Verify the environment variable defining the search path for the SAP HANA® database client shared library (libodbcHDB.so on UNIX™, libodbcHDB.DLL on Windows™).
Table 290: Shared library environment setting for SAP HANA® version

<table>
<thead>
<tr>
<th>SAP HANA® version</th>
<th>Shared library environment setting</th>
</tr>
</thead>
</table>
| SAP HANA® 2.x and higher | *UNIX™*: Add $HDBDIR to LD_LIBRARY_PATH (or its equivalent).  
*Windows™*: Add %HDBDIR% to PATH. |

5. To verify if the SAP HANA® client environment is correct, you can start the hdbsql command interpreter and connect to the server:

```
$ hdbsql -i 90 -d SystemDB -u SYSTEM -p password
```

6. Setup the FGLPROFILE entries for database connections.
   a) Define the SAP HANA database driver:

   ```
   dbi.database.dbname.driver = "dbmhdb"
   ```

   b) The "source" parameter defines the name of the SAP HANA® ODBC data source.

   ```
   dbi.database.dbname.source = "test1"
   ```

   c) Define the database schema selection if needed:

   Use the following entry to define the database schema to be used by the application. The database interface will automatically perform a `SET SCHEMA name` instruction to switch to a specific schema:

   ```
   dbi.database.dbname.hdb.schema = 'name'
   ```

   Here `dbname` identifies the database name used in the BDL program (DATABASE `dbname`) and `name` is the schema name to be used in the SET SCHEMA instruction. If this entry is not defined, no "SET SCHEMA" instruction is executed and the current schema defaults to the user's name.

Database concepts
SAP HANA® related database concept topics.

Database concepts
As with Informix®, one SAP HANA® database server can handle more than one database entity, with multitenant database containers. The multitenant database feature has been introduced in SAP HANA SPS09. In a given tenant database, you can create several schemas, owned by database users.

Informix® servers have an ID (INFORMIXSERVER) and databases are identified by name.

Tip: If you have several Informix database entities, migrating from the Informix database to another database it is a good opportunity to centralize all tables in a single database. To avoid conflicts with table names, use a prefix when needed.

Data storage concepts
Because SAP HANA® is an in-memory database engine, the Informix® storage concepts and options cannot be applied with this database.

Concurrency management

Data consistency and concurrency concepts
- **Data Consistency** applies to situations when readers want to access data currently being modified by writers.
- **Concurrent Data Access** applies to situations when several writers are accessing the same data for modification.
- **Locking Granularity** defines the amount of data concerned when a lock is set (for example, row, page, table).
**Informix®**

Informix® uses a locking mechanism to handle data consistency and concurrency. When a process changes database information with UPDATE, INSERT or DELETE, an exclusive lock is set on the touched rows. The lock remains active until the end of the transaction. Statements performed outside a transaction are treated as a transaction containing a single operation and therefore release the locks immediately after execution. SELECT statements can set shared locks, depending on isolation level. In case of locking conflicts (for example, when two processes want to acquire an exclusive lock on the same row for modification, or when a writer is trying to modify data protected by a shared lock), the behavior of a process can be changed by setting the lock wait mode.

Control:

- Lock wait mode: SET LOCK MODE TO ...
- Isolation level: SET ISOLATION TO ...
- Locking granularity: CREATE TABLE ... LOCK MODE {PAGE|ROW}
- Explicit exclusive lock: SELECT ... FOR UPDATE

Defaults:

- The default isolation level is READ COMMITTED.
- The default lock wait mode is NOT WAIT.
- The default locking granularity is PAGE.

**SAP HANA®**

SAP HANA® uses multi-version row and exclusive locks to manage data consistency and concurrency. Concurrent readers will see a consistent view of the database data without blocking concurrent write operations. Updates are implemented by inserting new versions of data and not by overwriting existing records. The database engine sets exclusive locks on the modified rows and shared locks when data is read, based on the isolation level. The locks are held until the end of the transaction. When multiple processes want to access the same data, the latest processes must wait until the first finishes its transaction. The lock granularity is at the row. For more details, see SAP HANA® documentation.

Control:

- Lock wait mode: SET TRANSACTION LOCK WAIT TIMEOUT milliseconds. Default is defined by server settings.
- Isolation level: SET TRANSACTION ISOLATION LEVEL {READ COMMITTED | REPEATABLE READ | SERIALIZABLE}.
- Locking granularity: row (or table level, if demanded by the SQL commant semantics).
- Explicit locking: SELECT ... FOR UPDATE

Defaults:

- The default isolation level is READ COMMITTED.

**Important:** With SAP HANA, when a lock timeout occurs, the complete SQL transaction is rolled back: SAP HANA will cancel the SQL statements executed in the transaction before the SQL statement which produced the lock timeout.

**Solution**

The SET ISOLATION TO ... Informix® syntax is converted to SET TRANSACTION ISOLATION LEVEL ... for SAP HANA®. The table shows the isolation level mappings applied by the database driver:

**Table 291: Isolation level mappings done by the SAP HANA® database driver**

<table>
<thead>
<tr>
<th>SET ISOLATION instruction in program</th>
<th>Native SQL command</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET ISOLATION TO DIRTY READ</td>
<td>SET TRANSACTION ISOLATION LEVEL READ</td>
</tr>
<tr>
<td></td>
<td>COMMITTED</td>
</tr>
</tbody>
</table>
### SET ISOLATION instruction in program vs. Native SQL command

<table>
<thead>
<tr>
<th>SET ISOLATION instruction in program</th>
<th>Native SQL command</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET ISOLATION TO COMMITTED READ [READ COMMITTED] [RETAIN UPDATE LOCKS]</td>
<td>SET TRANSACTION ISOLATION LEVEL READ COMMITTED</td>
</tr>
<tr>
<td>SET ISOLATION TO CURSOR STABILITY</td>
<td>SET TRANSACTION ISOLATION LEVEL REPEATABLE READ</td>
</tr>
<tr>
<td>SET ISOLATION TO REPEATABLE READ</td>
<td>SET TRANSACTION ISOLATION LEVEL SERIALIZABLE</td>
</tr>
</tbody>
</table>

When using the `SET LOCK MODE` instruction in your Genero programs, the SAP HANA® driver will execute the equivalent `SET TRANSACTION LOCK WAIT TIMEOUT` instruction, by converting the number of seconds to milliseconds.

For portability, it is recommended that you work with Informix® in the read committed isolation level, to make processes wait for each other (lock mode wait) and to create tables with the "lock mode row" option.

See Informix® and SAP HANA® documentation for more details about data consistency, concurrency and locking mechanisms.

Regarding transaction rollback when a lock timeout occurs, you may need to review your code, to handle the SAP HANA® behavior, where all SQL statements in the transaction are canceled. However, lock timeouts should not occur, if transactions are quick, and each process uses read committed isolation level, and define a lock timeout that is sufficient to let other processes terminate their transactions.

**Related concepts**
- Concurrent data access on page 599
  Understanding concurrent data access and data consistency.
- Optimistic locking on page 624
  Implementing optimistic locking to handle access concurrently to the same database records.
- Cursors WITH HOLD on page 631
  Programming WITH HOLD cursors using `SELECT` with and without `FOR UPDATE` clause.

**Transactions handling**

**Informix®**

With the Informix® native mode (non ANSI):
- Transactions blocks start with `BEGIN WORK` and terminate with `COMMIT WORK` or `ROLLBACK WORK`.
- Statements executed outside a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.

```
UPDATE tab1 SET ...   -- auto-committed
BEGIN WORK            -- start of TX block
UPDATE tab1 SET ...
UPDATE tab2 SET ...
...                    
COMMIT WORK           -- end of TX block
```

Informix® version 11.50 introduces savepoints:

```
SAVEPOINT name [UNIQUE]
ROLLBACK [WORK] TO SAVEPOINT [name] 
RELEASE SAVEPOINT name
```
SAP HANA®

Transactions in SAP HANA®:

- Beginning of transactions are implicit; two transactions are delimited by COMMIT or ROLLBACK.
- DDL statements can be executed in transactions, but these will persist on rollback.

SAP HANA® (version 2 SPS 04) supports savepoints with similar features as Informix SQL with:

- `SAVEPOINT name`
- `ROLLBACK TO SAVEPOINT name`
- `RELEASE SAVEPOINT name`

Solution

The Informix® behavior is simulated with an autocommit mode in the SAP HANA® database interface. A switch to the explicit commit mode is done when a BEGIN WORK is performed by the BDL program. Regarding the transaction control instructions, the BDL applications do not have to be modified in order to work with SAP HANA®.

**Note:** Savepoints are not supported.

See also [SELECT FOR UPDATE](../sql/support).

Related concepts

- Database transactions on page 600
- Database transactions define a set of SQL instructions to be executed as a whole, or rolled back as a whole.

Database users

**Informix®**

Until version 11.70.xC2, Informix® database users must be created at the operating system level and must be members of the 'informix' group.

Starting with 11.70.xC2, Informix® supports database-only users with the `CREATE USER` instruction, as in most other db servers.

Any database user must have sufficient privileges to connect and use resources of the database; user rights are defined with the `GRANT` command.

**SAP HANA®**

SAP HANA® users are created for each database tenant with the `CREATE USER` instruction. Users can be authenticated with a password that must be provided at each connection (`CREATE USER name PASSWORD password`), or can be identified with an external authentication mechanism (`CREATE USER name IDENTIFIED EXTERNALLY AS ...`).

For each database user a schema with the user's name is automatically created, owned by this user and becomes the default schema.

Solution

Create your database users with the `CREATE USER` SQL instruction.

Related concepts

- Database users and security on page 608
Properly identifying database users allows to use database security and audit features.

**Setting privileges**

**Informix**

Informix® users must have at least the `CONNECT` privilege to access the database:

```sql
GRANT CONNECT TO username
```

Application administration users need the `RESOURCE` privilege to create tables:

```sql
GRANT RESOURCE TO username
```

Since version 7.20, Informix® supports database roles:

```sql
GRANT rolename TO username
```

**SAP HANA®**

Typically, SAP HANA® users must have schema privileges to access the database table of a schema created by the application administrator.

SAP HANA® supports concept of `roles`, to define common user privileges.

**Solution**

Informix® and SAP HANA® user privileges management is quite similar.

**Data dictionary**

SAP HANA® related data dictionary topics.

**Data type conversion table: Informix to SAP HANA**

**Table 292: Data type conversion table (Informix to SAP HANA )**

<table>
<thead>
<tr>
<th>Informix® data types</th>
<th>SAP HANA® data types</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR(n)</td>
<td>NVARCHAR(n) (max is 5000 characters)</td>
</tr>
<tr>
<td>VARCHAR(n[,m])</td>
<td>NVARCHAR(n) (max is 5000 characters)</td>
</tr>
<tr>
<td>LVARCHAR(n)</td>
<td>NVARCHAR(n) (max is 5000 characters)</td>
</tr>
<tr>
<td>NCHAR(n)</td>
<td>N/A</td>
</tr>
<tr>
<td>NVARCHAR(n[,m])</td>
<td>N/A</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>INT / INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>BIGINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>INT8</td>
<td>BIGINT</td>
</tr>
<tr>
<td>SERIAL [(start)]</td>
<td>INTEGER (see note 1)</td>
</tr>
<tr>
<td>BIGSERIAL [(start)]</td>
<td>BIGINT (see note 1)</td>
</tr>
<tr>
<td>SERIAL8 [(start)]</td>
<td>BIGINT (see note 1)</td>
</tr>
<tr>
<td>DOUBLE PRECISION / FLOAT [(n)]</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>Informix® data types</td>
<td>SAP HANA® data types</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>---------------------------------------</td>
</tr>
<tr>
<td>REAL / SMALLFLOAT</td>
<td>REAL</td>
</tr>
<tr>
<td>NUMERIC / DEC / DECIMAL((p, s))</td>
<td>DECIMAL((p, s))</td>
</tr>
<tr>
<td>NUMERIC / DEC / DECIMAL((p&lt;=19))</td>
<td>DECIMAL((p*2, p))</td>
</tr>
<tr>
<td>NUMERIC / DEC / DECIMAL((p&gt;19))</td>
<td>N/A: Max p is 38 for SAP HANA</td>
</tr>
<tr>
<td>NUMERIC / DEC / DECIMAL</td>
<td>DECIMAL((32, 16))</td>
</tr>
<tr>
<td>MONEY((p, s))</td>
<td>DECIMAL((p, s))</td>
</tr>
<tr>
<td>MONEY((p))</td>
<td>DECIMAL((p, 2))</td>
</tr>
<tr>
<td>MONEY</td>
<td>DECIMAL((16, 2))</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME HOUR TO HOUR</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME HOUR TO MINUTE</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME HOUR TO SECOND</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME HOUR TO FRACTION((n))</td>
<td>TIMESTAMP (for fraction part)</td>
</tr>
<tr>
<td>DATETIME MINUTE TO MINUTE</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME MINUTE TO SECOND</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME MINUTE TO FRACTION((n))</td>
<td>TIMESTAMP (for fraction part)</td>
</tr>
<tr>
<td>DATETIME SECOND TO SECOND</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME SECOND TO FRACTION((n))</td>
<td>TIMESTAMP (for fraction part)</td>
</tr>
<tr>
<td>DATETIME FRACTION TO FRACTION((n))</td>
<td>TIMESTAMP (for fraction part)</td>
</tr>
<tr>
<td>DATETIME YEAR TO YEAR</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME YEAR TO MONTH</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME YEAR TO DAY</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME YEAR TO HOUR</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME YEAR TO MINUTE</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME YEAR TO SECOND</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME YEAR TO FRACTION((n))</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>DATETIME MONTH TO MONTH</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME MONTH TO DAY</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME MONTH TO HOUR</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME MONTH TO MINUTE</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME MONTH TO SECOND</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME MONTH TO FRACTION((n))</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>DATETIME DAY TO DAY</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME DAY TO HOUR</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME DAY TO MINUTE</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME DAY TO SECOND</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>Informix® data types</td>
<td>SAP HANA® data types</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>DATETIME DAY TO FRACTION(n)</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>INTERVAL q1 TO q2</td>
<td>VARCHAR(50)</td>
</tr>
<tr>
<td>TEXT</td>
<td>NCLOB</td>
</tr>
<tr>
<td>BYTE</td>
<td>CLOB</td>
</tr>
</tbody>
</table>

Notes:

1. For more details about serial emulation, see SERIAL and BIGSERIAL data types on page 1084.

**BOOLEAN data type**

**Informix®**

Informix® supports the BOOLEAN data type, which can store ’t’ or ’f’ values.

Genero BDL implements the BOOLEAN data type in a different way: A BOOLEAN variable stores integer values 1 or 0 (for TRUE or FALSE). This type is designed to hold the result of a boolean expression.

**SAP HANA®**

SAP HANA® provides a BOOLEAN SQL type, similar to the Genero BDL BOOLEAN type, which is able to store TRUE/FALSE values.

**Solution**

The SAP HANA® database interface supports the BOOLEAN SQL type.

The BOOLEAN type translation can be controlled with the following FGLPROFILE entry:

```
dbi.database.dsnname.ifxemul.datatype.boolean = {true | false}
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

Using portable data types on page 614

Only a limited set of data types are really portable across several database engines.

**CHAR and VARCHAR data types**

**Informix®**

Informix® supports the following character data types:

**Table 293: Informix® character data types**

<table>
<thead>
<tr>
<th>Informix® data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR (n)</td>
<td>SBCS and MBCS character data (max is 32767 bytes)</td>
</tr>
<tr>
<td>VARCHAR (n[1, n[1))</td>
<td>SBCS and MBCS character data (max is 255 bytes)</td>
</tr>
<tr>
<td>NCHAR (n)</td>
<td>Same as CHAR, with specific collation order</td>
</tr>
<tr>
<td>NVARCHAR (n[1, n[1))</td>
<td>Same as VARCHAR, with specific collation order</td>
</tr>
<tr>
<td>LVARCHAR (n)</td>
<td>max size varies depending on the IDS version</td>
</tr>
</tbody>
</table>
With Informix®, both CHAR/VARCHAR and NCHAR/NVARCHAR data types can be used to store single-byte or multibyte encoded character strings. The only difference between CHAR/VARCHAR and NCHAR/NVARCHAR is in how they use sorting: N[VAR]CHAR types use the collation order, while [VAR]CHAR types use the byte order.

The character set used to store strings in CHAR/VARCHAR/NCHAR/NVARCHAR columns is defined by the DB_LOCALE environment variable.

The character set used by applications is defined by the CLIENT_LOCALE environment variable.

Informix® uses Byte Length Semantics (the size N that you specify in [VAR]CHAR(N) is expressed in bytes, not characters as in some other databases)

**SAP HANA®**

SAP HANA® provides the following character data types:

- **VARCHAR(N)** with N <= 5000 bytes, to store ASCII-7 character strings.
- **NVARCHAR(N)** with N <= 5000 characters, to store UNICODE character strings.
- **ALPHANUM(N)** with N <= 127 bytes, to store ASCII-7 alpha-numeric character strings.
- **SHORTTEXT(N)**, supporting text and string search features.
- **CHAR(N) / NCHAR(N)**: Not officially supported!

The SAP HANA® VARCHAR data type must only be used to store ASCII-7 character strings. To store non-ASCII / locale dependent character strings (such as the UTF-8 codeset), you must use the NVARCHAR data type.

**Note:** The SAP HANA® database does not officially support the CHAR and NCHAR datatypes. See SAP HANA® documentation for more details.

**Solution**

Because SAP HANA® VARCHAR(N) can only store ASCII-7 characters and does not recommend to use CHAR/NCHAR types, Informix® CHAR(N), VARCHAR(N) or LVARCHAR(N) types must be mapped to SAP HANA® NVARCHAR(N) when using UTF-8, or a single-byte encoding like ISO-8859-15. In some rare cases where the application character set is ASCII-7 or when only ASCII-7 data is stored in the column, you can use SAP HANA® VARCHAR(N) type.

**Important:** Since SAP HANA® does not officially support the CHAR/NCHAR types, all character columns must be SAP HANA® VARCHAR (if ASCII-7) or NVARCHAR (if non-ASCII). As the semantics of VARCHAR types are different regarding blank padding, you will get different behavior, when comparing with string values containing trailing blanks.

When extracting a database schema from a SAP HANA® database, the fg1dbsch schema extractor uses the size of the column in characters, not the octet length. If you have created a NVARCHAR(10 (characters) ) column a in SAP HANA® database using the UTF-8 character set, the .sch file will get a size of 10, that will be interpreted depending on FGL_LENGTH_SEMANTICS as a number of bytes or characters.

Check that your database schema does not use CHAR or VARCHAR types with a length exceeding the SAP HANA® limits especially as the Informix® CHAR type has a very long size limit compared to SAP HANA® NVARCHAR.

The SAP HANA® database interface of Genero automatically converts from/to the application locale to the database locale. Therefore, no database client locale configuration is required with SAP HANA®.

See also the section about Localization.

**Related concepts**

CHAR and VARCHAR types on page 616
Using the CHAR and VARCHAR data types with different databases.

**Numeric data types**

**Informix®**

Informix® supports several data types to store numbers:

**Table 294: Informix® numeric data types**

<table>
<thead>
<tr>
<th>Informix® data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>16 bit signed integer</td>
</tr>
<tr>
<td>INTEGER</td>
<td>32 bit signed integer</td>
</tr>
<tr>
<td>BIGINT</td>
<td>64 bit signed integer</td>
</tr>
<tr>
<td>INT8</td>
<td>64 bit signed integer (replaced by BIGINT)</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>Equivalent to DECIMAL(16)</td>
</tr>
<tr>
<td>DECIMAL(p)</td>
<td>Floating-point decimal number (max precision is 32)</td>
</tr>
<tr>
<td>DECIMAL(p,s)</td>
<td>Fixed-point decimal number (max precision is 32)</td>
</tr>
<tr>
<td>MONEY</td>
<td>Equivalent to DECIMAL(16,2)</td>
</tr>
<tr>
<td>MONEY(p)</td>
<td>Equivalent to DECIMAL(p,2) (max precision is 32)</td>
</tr>
<tr>
<td>MONEY(p,s)</td>
<td>Equivalent to DECIMAL(p,s) (max precision is 32)</td>
</tr>
<tr>
<td>REAL / SMALLFLOAT</td>
<td>32-bit floating point decimal (C float)</td>
</tr>
<tr>
<td>DOUBLE PRECISION / FLOAT[(n)]</td>
<td>64-bit floating point decimal (C double)</td>
</tr>
</tbody>
</table>

**SAP HANA®**

SAP HANA® supports the following numeric data types:

**Table 295: SAP HANA® numeric data types**

<table>
<thead>
<tr>
<th>SAP HANA® data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TINYINT</td>
<td>8 bit unsigned integer (0 to 255)</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>16 bit signed integer (-32,768 to 32,767)</td>
</tr>
<tr>
<td>INTEGER</td>
<td>32 bit signed integer (-2,147,483,648 to 2,147,483,647)</td>
</tr>
<tr>
<td>BIGINT</td>
<td>64 bit signed integer (-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)</td>
</tr>
<tr>
<td>DECIMAL(p,s)</td>
<td>Fixed point decimal (max p is 38)</td>
</tr>
<tr>
<td>DECIMAL(p)</td>
<td>Floating point decimal (approximative)</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>Floating point decimal (approximative)</td>
</tr>
<tr>
<td>SMALLDECIMAL</td>
<td>Floating point decimal (approximative)</td>
</tr>
<tr>
<td>REAL</td>
<td>32-bit floating point decimal (C float)</td>
</tr>
<tr>
<td>FLOAT[(n)] (DOUBLE)</td>
<td>64-bit floating point decimal (C double)</td>
</tr>
</tbody>
</table>
Solution

Table 296: Informix® numeric data types and SAP HANA® equivalents

<table>
<thead>
<tr>
<th>Informix® data type</th>
<th>SAP HANA® equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>INT8 / BIGINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>DECIMAL(p, s)</td>
<td>DECIMAL(p, s)</td>
</tr>
<tr>
<td>DECIMAL(p&lt;=19)</td>
<td>DECIMAL(p*2, p)</td>
</tr>
<tr>
<td>DECIMAL(p&gt;19)</td>
<td>N/A: Max p is 38 for SAP HANA</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL(32, 16)</td>
</tr>
<tr>
<td>MONEY</td>
<td>DECIMAL(16, 2)</td>
</tr>
<tr>
<td>MONEY(p)</td>
<td>DECIMAL(p, 2)</td>
</tr>
<tr>
<td>MONEY(p, s)</td>
<td>DECIMAL(p, s)</td>
</tr>
<tr>
<td>SMALLFLOAT</td>
<td>REAL</td>
</tr>
<tr>
<td>FLOAT[(n)]</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

SQL scripts to create databases must be converted manually. Tables created from BDL programs do not have to be converted; the database interface detects the MONEY data type and uses the DECIMAL type for SAP HANA®.

**Important:** There is no SAP HANA® equivalent for the Informix® DECIMAL(p) floating point decimal (i.e. without a scale). If your application is using such data types, you must review the database schema in order to use SAP HANA® compatible types. To workaround the SAP HANA® limitation, the SAP HANA® database driver converts DECIMAL(p) types to a DECIMAL(2*p, p), to store all possible numbers an Informix® DECIMAL(p) can store. However, the original Informix® precision cannot exceed 19, because SAP HANA® supports a maximum DECIMAL precision of 38 (2*19). If the original precision is bigger than 19, a CREATE TABLE statement executed from a Genero program will fail with an SQL error.

The numeric types translation can be controlled with the following FGLPROFILE entries:

```
dbi.database.dsname.ifxemul.datatype.smallint = □ true □ false □
dbi.database.dsname.ifxemul.datatype.integer = □ true □ false □
dbi.database.dsname.ifxemul.datatype.bigint = □ true □ false □
dbi.database.dsname.ifxemul.datatype.int8 = □ true □ false □
dbi.database.dsname.ifxemul.datatype.decimal = □ true □ false □
dbi.database.dsname.ifxemul.datatype.money = □ true □ false □
dbi.database.dsname.ifxemul.datatype.float = □ true □ false □
dbi.database.dsname.ifxemul.datatype.smallfloat = □ true □ false □
```

For more details see [IBM Informix emulation parameters in FGLPROFILE](#) on page 668.

**Related concepts**

- [Using portable data types](#) on page 614

Only a limited set of data types are really portable across several database engines.

**DATE and DATETIME data types**

**Informix®**

Informix® provides two data types to store date and time information:

- **DATE** = for year, month and day storage.
• DATETIME = for year to fraction (1-5) storage.

The DATE type is stored as an INTEGER with the number of days since 1899/12/31.

The DATETIME type can be defined with various time units, by specifying a start and end qualifier. For example, you can define a datetime to store an hour-to-second time value with DATETIME HOUR TO SECOND.

The values of Informix® DATETIME can be represented with a character string literal, or as DATETIME () literals:

| '2017-12-24 15:45:12.345' -- a DATETIME YEAR TO FRACTION(3) |
| '15:45' -- a DATETIME HOUR TO MINUTE |
| DATETIME(2017-12-24 12:45) YEAR TO MINUTE |
| DATETIME(12:45:56.333) HOUR TO FRACTION(3) |

Informix® is able to convert quoted strings to DATE / DATETIME data, if the string contains matching environment parameters. The string to date conversion rules for DATE is defined by the DBDATE environment variable. The string to datetime format for DATETIME is defined by the GL_DATETIME environment variable.

**Note:** Within Genero programs, the string representation for DATETIME values is always ISO (YYYY-MM-DD hh:mm:ss.fffff)

Informix® supports date arithmetic on DATE and DATETIME values. The result of an arithmetic expression involving dates/times is an INTEGER number of days when only DATE values are used, and an INTERVAL value if a DATETIME is used in the expression.

Informix® automatically converts an INTEGER to a DATE when the integer is used to set a value of a date column.

Informix® provides the CURRENT [q1 TO q2 l] operator, to get the system date/time on the server where the current database is located. When no qualifiers are specified, CURRENT returns a DATETIME YEAR TO FRACTION(3). Informix also supports the SYSDATE operator, which returns the current system time as a DATETIME YEAR TO FRACTION(5).

**Note:** The USEOSTIME configuration parameter must be set to 1 in order to get the subsecond precision in CURRENT and SYSDATE operators. See Informix documentation for more details.

**SAP HANA®**

SAP HANA® provides following data type to store dates:

**Table 297: SAP HANA® date/time data types**

<table>
<thead>
<tr>
<th>SAP HANA® data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>for year, month, day storage</td>
</tr>
<tr>
<td>SECONDDATE</td>
<td>for year, month, day, hour, minutes, seconds storage</td>
</tr>
<tr>
<td>TIME</td>
<td>for hour, minutes, seconds storage</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>for year, month, day, hour, minutes, seconds, fraction(7) storage</td>
</tr>
</tbody>
</table>

As Informix®, SAP HANA® can convert quoted strings to date-time values.

By default, the date/time formats are:

• 'YYYY-MM-DD' for DATE
• 'HH:MM:SS' for TIME
• 'YYYY-MM-DD HH:MM:SS' for SECONDDATE
• 'YYYY-MM-DD HH:MM:SS.FF7' for TIMESTAMP

The date/time format can be changed at SQL session level with the SET command:

```sql
SET 'DATE_FORMAT' = 'DD/MM/YYYY';
SET 'TIME_FORMAT' = 'HH:MI:SS';
```
Solution

Use the following conversion rules to map Informix date/time types to SAP HANA® date/time types:

Table 298: Informix® data types and SAP HANA® equivalents

<table>
<thead>
<tr>
<th>Informix® data type</th>
<th>SAP HANA® data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME HOUR TO HOUR</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME HOUR TO MINUTE</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME HOUR TO SECOND</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME HOUR TO FRACTION(n)</td>
<td>TIMESTAMP (for fraction part)</td>
</tr>
<tr>
<td>DATETIME MINUTE TO MINUTE</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME MINUTE TO SECOND</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME MINUTE TO FRACTION(n)</td>
<td>TIMESTAMP (for fraction part)</td>
</tr>
<tr>
<td>DATETIME SECOND TO SECOND</td>
<td>TIME</td>
</tr>
<tr>
<td>DATETIME SECOND TO FRACTION(n)</td>
<td>TIMESTAMP (for fraction part)</td>
</tr>
<tr>
<td>DATETIME FRACTION TO FRACTION(n)</td>
<td>TIMESTAMP (for fraction part)</td>
</tr>
<tr>
<td>DATETIME YEAR TO YEAR</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME YEAR TO MONTH</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME YEAR TO DAY</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME YEAR TO HOUR</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME YEAR TO MINUTE</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME YEAR TO SECOND</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME YEAR TO FRACTION(n)</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>DATETIME MONTH TO MONTH</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME MONTH TO DAY</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME MONTH TO HOUR</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME MONTH TO MINUTE</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME MONTH TO SECOND</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME MONTH TO FRACTION(n)</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>DATETIME DAY TO DAY</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME DAY TO HOUR</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME DAY TO MINUTE</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME DAY TO SECOND</td>
<td>SECONDDATE</td>
</tr>
<tr>
<td>DATETIME DAY TO FRACTION(n)</td>
<td>TIMESTAMP</td>
</tr>
</tbody>
</table>

SET 'SECONDDATE_FORMAT' = 'DD/MM/YYYY HH:MI:SS';
SET 'TIMESTAMP_FORMAT' = 'DD/MM/YYYY HH:MI:SS.FF3';
The **DATE** and **DATETIME** types translation can be controlled with the following FGLPROFILE entries:

```
  dbi.database.dsname.ifxemul.datatype.date = true | false
  dbi.database.dsname.ifxemul.datatype.datetime = true | false
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**SAP HANA**® and Informix® **DATE** data type are equivalent and store year, month, day values.

**SAP HANA**® **TIME** data type can be used to store Informix® **DATETIME HOUR TO SECOND, DATETIME HOUR TO MINUTE, SECOND, except FRACTION (n)**. If the time-class **DATETIME** uses fraction of seconds such as **DATETIME HOUR TO FRACTION (n)**, the native **SAP HANA**® type must be **TIMESTAMP**, because **SAP HANA**® **TIME** does not store fractions of seconds. Missing time parts default to 00:00:00. For example, when using a **DATETIME MINUTE TO SECOND** with the value of "45:23", the **SAP HANA**® **TIME** value will be "00:45:23".

Informix® **DATETIME** values with any qualifiers from **YEAR** to **SECOND** can be stored in **SAP HANA**® **SECONDDATE** columns. **DATETIME** types with **FRACTION (n)** can be stored in **SAP HANA**® **TIMESTAMP** columns. Missing date or time parts default to 1900-01-01 00:00:00.0. For example, when using a **DATETIME DAY TO MINUTE** with the value of "23 11:45", the **SAP HANA**® **SECONDDATE** value will be "1900-01-23 11:45:00".

**Date/time SQL functions**

**Table 299: Informix® and SAP HANA® date/time SQL functions**

<table>
<thead>
<tr>
<th>Informix®</th>
<th>SAP HANA®</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>current_date</td>
</tr>
<tr>
<td>current hour to second</td>
<td>current_time</td>
</tr>
<tr>
<td>current year to fraction(5)</td>
<td>current_timestamp</td>
</tr>
</tbody>
</table>

**Related concepts**

Date/time literals in SQL statements on page 635

Good practices for date and time handling in SQL.

**INTERVAL data type**

**Informix®**

Informix® provides the **INTERVAL** data type to store a value that represents a span of time.

**INTERVAL** types are divided into two classes:

- **year-month** intervals. For example: **INTERVAL YEAR(5) TO MONTH**
- **day-time** intervals. For example: **INTERVAL DAY(9) TO SECOND**

**INTERVAL** columns can be defined with various time units, by specifying a start and end qualifier. For example, you can define an interval to store a number of hours and minutes with **INTERVAL HOUR(n) TO MINUTE**, where \( n \) defines the maximum number of digits for the hours unit.

The values of Informix® **INTERVAL** can be represented with a character string literal, or as **INTERVAL()** literals:

- "'9834 15:45:12.345'" -- an **INTERVAL DAY(6) TO FRACTION(3)**
- "'7634-11'" -- an **INTERVAL YEAR(9) TO MONTH**
- **INTERVAL(18734:45) HOUR(5) TO MINUTE**
- **INTERVAL(-7634-11) YEAR(5) TO MONTH**
SAP HANA® does not provide a data type corresponding the Informix® INTERVAL data type.

Solution
The INTERVAL data type and values are converted CHAR(50) column with SAP HANA®.
INTERVAL values can be stored and retrieved from the database. However, since SAP HANA® does not support a native interval type, arithmetics cannot be performed on the database side in SQL statements.
The INTERVAL types translation can be controlled with the following FGLPROFILE entry:

```
dbi.database.dsname.ifxemul.datatype.interval = {true | false}
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**SERIAL and BIGSERIAL data types**

**Informix®**
Informix® supports the SERIAL, BIGSERIAL data types to produce automatic integer sequences:
- SERIAL can produce 32 bit integers (INTEGER)
- BIGSERIAL can produced 64 bit integers (BIGINT)
- SERIAL8 is a synonym for BIGSERIAL

Steps to use serials with Informix®:
1. Create the table with a column using SERIAL, or BIGSERIAL.
2. To generate a new serial, no value or a zero value is specified in the INSERT statement:
   ```
   INSERT INTO tab1 ( c ) VALUES ( 'aa' )
   INSERT INTO tab1 ( k, c ) VALUES ( 0, 'aa' )
   ```
3. After INSERT, the new value of a SERIAL column is provided in SQLCA.SQLERRD[2], while the new value of a BIGSERIAL value must be fetched with a SELECT dbinfo('bigserial') query.

Informix® allows you to insert rows with a value different from zero for a serial column. Using an explicit value will automatically increment the internal serial counter, to avoid conflicts with future INSERT statements that are using a zero value:

```
CREATE TABLE tab ( k SERIAL); -- internal counter = 0
INSERT INTO tab VALUES ( 0 ); -- internal counter = 1
INSERT INTO tab VALUES ( 10 ); -- internal counter = 10
INSERT INTO tab VALUES ( 0 ); -- internal counter = 11
DELETE FROM tab; -- internal counter = 11
INSERT INTO tab VALUES ( 0 ); -- internal counter = 12
```

**SAP HANA®**
SAP HANA® supports IDENTITY columns:

```
CREATE TABLE tab ( 
   k BIGINT GENERATED BY DEFAULT AS IDENTITY 
   (START WITH 101 INCREMENT BY 1) NOT NULL,
   ... 
) 
```
To get the last generated IDENTITY value after an INSERT, SAP HANA® provides the `CURRENT_IDENTITY_VALUE()` function:

```sql
INSERT INTO table_with_identity_column VALUES (...)
SELECT CURRENT_IDENTITY_VALUE() FROM DUMMY
```

SAP HANA® supports SEQUENCES:

```sql
CREATE SEQUENCE seq1 START WITH 100
```

To create a new sequence number, you must use the "`sequence-name.NEXTVAL`" expression:

```sql
INSERT INTO table VALUES ( seq1.NEXTVAL, ... )
```

To get the last generated sequence number, you must use the "`sequence-name.CURRVAL`" expression:

```sql
SELECT seq1.CURRVAL FROM DUMMY
```

**Solution**

**Note:** For best SQL portability when using different types of databases, consider using sequences as described in Solution 3: Use native SEQUENCE database objects on page 628.

To emulate Informix® serials with SAP HANA®, you can use IDENTITY columns (1), or insert triggers using sequences (2). The first solution is faster, but does not allow explicit serial value specification in insert statements; the second solution is slower but allows explicit serial value specification.

**Important:** The trigger-based solution is provided to simplify the conversion from Informix, but it is slower than the solution using identity columns. We strongly recommend that you use native IDENTITY columns instead to get best performances.

The method used to emulate SERIAL types is defined by the `ifxemul.datatype.serial.emulation` FGLPROFILE parameter:

```sql
dbi.database.dbname.ifxemul.datatype.serial.emulation = {"native" | "trigseq"}
```

- native: uses IDENTITY columns.
- trigseq: uses insert triggers with sequences.

The default emulation technique is "native".

The serial types emulation can be enabled or disabled with the following FGLPROFILE entries:

```sql
dbi.database.dbname.ifxemul.datatype.serial = {true | false}
dbi.database.dbname.ifxemul.datatype.serial8 = {true | false}
dbi.database.dbname.ifxemul.datatype.bigserial = {true | false}
```

**Disabling automatic serial retrieval for SQLCA.SQLERRD[2]**

SERIAL emulation can be totally disabled by setting the `ifxemul.datatype.serial` FGLPROFILE entry to false:

```sql
dbi.database.dbname.ifxemul.datatype.serial = false
```

For Informix® compatibility, after an `INSERT` statement, the ODI drivers automatically execute another SQL query (or do a DB client API call when possible), to get the last generated serial, and fill the `SQLCA.SQLERRD[2]` register. This results in some overhead that can be avoided, if the `SQLCA.SQLERRD[2]` register is not used by the program.
When SERIAL emulation is required (to create temp tables with a serial column during program execution), and the SQLCA.SQLERRD[2] register does not need to be filled, (typically because you use your own method to retrieve the last generated serial), you can set the ifxemul.datatype.serial.sqlerrd2 FGLPROFILE entry to false. This will avoid the automatic retrieval of last serial value to fill SQLCA.SQLERRD[2]:

```
dbi.database.dbname.ifxemul.datatype.serial.sqlerrd2 = false
```

See also `db_get_last_serial()` on page 2308.

**Using the native serial emulation**

In database creation scripts, all SERIAL[ (n) ] data types must be converted by hand to:

```
INTEGER GENERATED ALWAYS AS IDENTITY[( START WITH n INCREMENT BY 1)]
```

while the SERIAL8 and BIGSERIAL[(n)] types must be converted to:

```
BIGINT GENERATED ALWAYS AS IDENTITY[( START WITH n INCREMENT BY 1)]
```

**Note:** Unlike other database types, the syntax of SAP HANA® IDENTITY options does not use the comma.

Tables created from the BDL programs can use the SERIAL data type: When a BDL program executes a CREATE [TEMP] TABLE with a SERIAL column, the database interface automatically converts the SERIAL[ (n) ] data type to an IDENTITY specification.

In BDL, the new generated SERIAL value is available from the SQLCA.SQLERRD[2] variable: The database interface which performs a call to the CURRENT_IDENTITY_VALUE() function. However, SQLCA.SQLERRD[2] is defined as an INTEGER, it cannot hold values from BIGINT identity columns. If you are using BIGINT IDENTITY columns, you must use the IDENTITY_VAL_LOCAL() function.

The SAP HANA® GENERATED ALWAYS AS IDENTITY feature does not allow you to specify the value of IDENTITY columns, it is mandatory to convert all INSERT statements to remove the SERIAL column from the list. For example, the following statement:

```
INSERT INTO tab (col1,col2) VALUES (0, p_value)
```

must be converted to:

```
INSERT INTO tab (col2) VALUES (p_value)
```

Static SQL INSERT using records defined from the schema file must also be reviewed:

```
DEFINE rec LIKE tab.*
INSERT INTO tab VALUES ( rec.* ) -- will use the serial column
```

must be converted to:

```
INSERT INTO tab VALUES rec.* -- without parentheses, serial column is removed
```

**Using the trigseq serial emulation**

In database creation scripts, all SERIAL[ (n) ] data types must be converted to INTEGER DEFAULT 0 data types, SERIAL8/BIGSERIAL must be converted to BIGINT DEFAULT 0, and you must create a sequence and a trigger for each table using a SERIAL.

To know how to write those triggers, you can create a small Genero program that creates a table with a SERIAL column. Set the FGLSQLDEBUG environment variable and run the program. The debug output will show you the native SQL commands to create the sequence and the trigger.
Tables created from the BDL programs can use the **SERIAL** data type: When a BDL program executes a `CREATE [TEMP] TABLE` with a **SERIAL** column, the database interface automatically converts the **SERIAL[(n)]** data type to **INTEGER** and creates the sequence and the insert trigger.

**Important:**
- With SAP HANA®, trigger creation is not allowed on temporary tables. Therefore, the "trigseq" method cannot work with temporary tables using serials.

**Related concepts**
- [Auto-incremented columns (serials)](page 626)
- How to implement automatic record keys.

### ROWID columns

**Informix®**

When creating a table, Informix® automatically adds a **ROWID** integer column (applies to non-fragmented tables only).

The **ROWID** column is auto-filled with a unique number and can be used like a primary key to access a given row.

**Note:** Informix® **ROWID** usage was a common practice in the early days of Informix® 4GL programming. Today it is recommended to define all your database tables with a **PRIMARY KEY** to uniquely identify rows.

With Informix®, the `SQLCA.SQLERRD[6]` register contains the **ROWID** of the last row affected by an **INSERT**, **UPDATE** or **DELETE** statement.

**SAP HANA®**

SAP HANA® has an internal rowid column ($rowid$) but this feature is internal and not documented.

**Solution**

If your Genero BDL application uses rowid columns, review the program logic to use primary keys instead. If the database table does not define a primary key, it should be added. All references to `SQLCA.SQLERRD[6]` must be removed, because this variable will not hold the **ROWID** of the last modified row.

For databases where the keyword of the rowid pseudo-column is different than "ROWID", the translation can be controlled with the following FGLPROFILE entry:

```
  dbi.database.dsnname.ifxemul.rowid = \true \false
```

For more details see [IBM Informix emulation parameters in FGLPROFILE](page 668).

**Related concepts**
- [Using ROWID columns](page 644)
  - Automatic ROWID columns is not a common database feature.

### TEXT and BYTE (LOB) types

**Informix®**

Informix® provides the **TEXT**, **BYTE**, **CLOB** and **BLOB** data types to store very large texts or binary data.

Legacy Informix® 4GL applications typically use the **TEXT** and **BYTE** types.

Genero BDL does not support the Informix® **CLOB** and **BLOB** types.

**SAP HANA®**

SAP HANA® supports several data types for large objects storage:
Table 300: SAP HANA® LOB data types

<table>
<thead>
<tr>
<th>SAP HANA® data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB</td>
<td>To store large binary data like images</td>
</tr>
<tr>
<td>CLOB</td>
<td>To store large ASCII text data</td>
</tr>
<tr>
<td>NCLOB</td>
<td>To store large UNICODE text data</td>
</tr>
<tr>
<td>TEXT</td>
<td>Similar to NCLOB, but supports textsearch-features and is only for column-tables</td>
</tr>
</tbody>
</table>

In SAP HANA®, the maximum size of an LOB is 2 GB.

**Solution**

The SAP HANA® database interface can convert BDL TEXT data to SAP HANA NCLOB and BYTE data to SAP HANA BLOB.

If you want to use the TEXT type for SAP HANA® column-tables, the tables must be created outside Genero BDL programs, or the TEXT data type emulation flag must be set to false when creating tables in BDL programs:

```plaintext
dbi.database.dbname.ifxemul.datatype.text = false
```

The TEXT and BYTE types translation can be controlled with the following FGLPROFILE entries:

```plaintext
dbi.database.dsnname.ifxemul.text = { true | false }
dbi.database.dsnname.ifxemul.byte = { true | false }
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

**Related concepts**

- Using portable data types on page 614

Only a limited set of data types are really portable across several database engines.

**Table constraints**

**Informix®**

Informix® supports primary key, unique, foreign key, default and check constraints.

The constraint naming syntax is different in Informix® and most other databases: Informix expects the constraint name after the constraint definition:

```plaintext
CREATE TABLE emp (  ...
   emp_code CHAR(10) UNIQUE CONSTRAINT pk_emp,  ...
)  
```

While other SQL database brands require to specify the constraint name before the constraint definition:

```plaintext
CREATE TABLE emp (  
   emp_code CHAR(10) CONSTRAINT pk_emp UNIQUE,  
)  
```
SAP HANA®

SAP HANA® supports primary key, unique, foreign key, default and check constraints.

Constraint naming

The constraint naming clause must be placed before the constraint specification.

Primary keys

Like Informix®, SAP HANA® creates an index to enforce PRIMARY KEY constraints (some RDBMS do not create indexes for constraints). Using CREATE UNIQUE INDEX to define unique constraints is obsolete (use primary keys or a secondary key instead).

Note: SAP HANA® primary key constraints do not allow NULLs; make sure your tables do not contain NULLs in the primary key columns.

Unique constraints

Like Informix®, SAP HANA® creates an index to enforce UNIQUE constraints (some RDBMS do not create indexes for constraints).

Note: SAP HANA® unique constraints do allow NULLs; this can be forced with NOT NULL.

Foreign keys

Both Informix® and SAP HANA® support the ON DELETE CASCADE option.

Check constraints

Informix® supports CHECK constraints, but this feature is not available in SAP HANA®.

Null constraints

Informix® and SAP HANA® support NOT NULL constraints.

Solution

Constraint naming syntax: The database interface does not convert constraint naming expressions when creating tables from BDL programs. Review the database creation scripts to adapt the constraint naming clauses for SAP HANA®.

Related concepts

Data definition statements on page 613

It is recommended to avoid use of DDL in programs.

Name resolution of SQL objects

Informix®

Informix® uses the following form to identify an SQL object:

database[@dbservername]:][{owner|"owner"}.]identifier

The ANSI convention is to use double quotes for identifier delimiters (For example: "customer"."cust_name").

Informix® database object names are not case-sensitive in non-ANSI databases. When using double-quoted identifiers, Informix® becomes case sensitive.
With non-ANSI Informix® databases, you do not have to give a schema name before the tables when executing an SQL statement:

```
SELECT ... FROM customer WHERE ... 
```

In Informix® ANSI compliant databases:

- The table name must include "owner", unless the connected user is the owner of the database object.
- The database server shifts the owner name to uppercase letters before the statement executes, unless the owner name is enclosed in double quotes.

**SAP HANA®**

SAP HANA® database object names are case-sensitive. When a name is used without double quotes, it is automatically converted to uppercase letters. When using double quotes, the names are not converted:

```
CREATE TABLE tab1 ( Key INT, Col1 CHAR(20) )
-- Table name is "TAB1", column names are "KEY" and "COL1"
CREATE TABLE "Tab1" ( "Key" INT, "Col1" CHAR(20) )
-- Table name is "Tab1", column names are "Key" and "Col1"
```

In an SAP HANA®, tables always belong to a database schema. When executing a SQL statement, a schema name must be used as the high-order part of a two-part object name, unless the current schema corresponds to the table's schema.

The default (implicit) schema is the current user's name but it can be changed with the `SET SCHEMA` instruction.

**Solution**

To write portable SQL, regarding database object names:

1. Use simple database object names (without any owner/schema prefix)
2. Do not use double quotes to surround database object identifiers.
3. If needed, define public synonyms to reference database objects in others databases/schema.
4. Specify database object identifiers in lowercase.

See also Naming database objects on page 637.

The SAP HANA® schema concept:

After a connection, the database interface can automatically execute a `SET SCHEMA name` instruction if the following FGLPROFILE entry is defined:

```
dbi.database.dbname.hdb.schema= "name"
```

Here `dbname` identifies the database name used in the BDL program (DATABASE `dbname`) and `name` is the schema name to be used in the SET SCHEMA instruction. If this entry is not defined, no "SET SCHEMA" instruction is executed and the current schema defaults to the user's name.

Examples:

```
dbi.database.stores.hdb.schema= "STORES1"
dbi.database.accnts.hdb.schema= "ACCSCH"
```
Data manipulation
SAP HANA® related data manipulation topics.

Reserved words

Informix®
With Informix®, it is possible to create database objects with reserved words.
For example:

```sql
CREATE TABLE table ( char CHAR(10) );
```

Indeed this is not good practice, but Informix® SQL allows this to be backward compatible when introducing a new keyword in the SQL syntax.

Most other database systems do not allow reserved words as database identifiers. If your legacy code is using SQL reserved words of the target database SQL syntax, an error will be thrown at CREATE TABLE execution.

SAP HANA®
Even if SAP HANA® allows SQL reserved keywords as SQL object names when using double-quotation marks, it is still recommended that you check your existing database schema for use of SAP HANA® reserved words.

Solution
See SAP HANA® documentation for reserved keywords.

Outer joins

Informix® OUTER() syntax
In Informix® SQL, outer joins can be defined in the FROM clause with the OUTER keyword:

```sql
SELECT ... FROM a, OUTER (b) WHERE a.key = b.akey

SELECT ... FROM a, OUTER(b,OUTER(c)) WHERE a.key = b.akey
AND b.key1 = c.bkey1 AND b.key2 = c.bkey2
```

Informix® also supports the ANSI OUTER join syntax, which is the recommended way to specify outer joins with recent SQL database engines:

```sql
SELECT ... FROM cust LEFT OUTER JOIN order
ON cust.key = order.custno
WHERE ...
```

SAP HANA®
SAP HANA® supports the ANSI outer join syntax:

```sql
SELECT ...
FROM cust LEFT OUTER JOIN order
  LEFT OUTER JOIN item
  ON order.key = item.ordno
  ON cust.key = order.custno
WHERE order.cdate > current date
Solution
The Genero database drivers can convert Informix® OUTER specifications to ANSI outer joins.

Note: For better SQL portability, use the ANSI outer join syntax instead of the old Informix® OUTER syntax.

The outer join translation can be controlled with the following FGLPROFILE entry:

```
dbi.database.dsnname.ifxemulouters = true false
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

- **Prerequisites:**
  1. In the FROM clause, the main table must be the first item and the outer tables must be listed from left to right in the order of outer levels.
     
     Example which does not work:
     
     ```
     ... FROM OUTER(tab2), tab1
     ```
  2. The outer join in the WHERE clause must use the table name as prefix:
     
     ```
     ... WHERE tab1.col1 = tab2.col2
     ```

- **Restrictions:**
  1. Statements composed by 2 or more SELECT instructions are not supported:
     
     ```
     SELECT ... UNION SELECT ...
     ```
     
     or:
     
     ```
     SELECT ... WHERE col IN (SELECT...)
     ```
  2. Additional conditions on outer table columns cannot be detected and therefore are not supported:
     
     ```
     ... FROM tab1, OUTER(tab2)
     WHERE tab1.col1 = tab2.col2
     AND tab2.colx > 10
     ```
  3. Using subscript in outer conditions:
     
     ```
     ... FROM tab1, OUTER(tab2)
     WHERE tab1.col1[1,3] = tab2.col2[1,3]
     ```

- **Notes:**
  1. Table aliases are detected in OUTER expressions.
     
     OUTER example with table alias:
     
     ```
     ... OUTER(tab1 alias1) ...
     ```
  2. In the outer join, `outertab.col` can be placed on both right or left sides of the equal sign:
     
     ```
     ... WHERE outertab.col1 = maintab.col2
     ```
  3. Table names detection is not case-sensitive:
     
     ```
     SELECT ... FROM tab1, TAB2
     WHERE tab1.col1 = tab2.col2
     ```
4. **Temporary tables** are supported in OUTER specifications:

```
CREATE TEMP TABLE tt1 ( ... )
SELECT ... FROM tab1, OUTER(tt1) ...
```

**Related concepts**

*Outer joins* on page 643

Use standard ISO outer join syntax instead of the old IBM® Informix® `OUTER()` syntax.

**Transactions handling**

**Informix®**

With the Informix® native mode (non ANSI):

- Transactions blocks start with `BEGIN WORK` and terminate with `COMMIT WORK` or `ROLLBACK WORK`.
- Statements executed outside a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.

```
UPDATE tab1 SET ... -- auto-committed
BEGIN WORK            -- start of TX block
UPDATE tab1 SET ...
UPDATE tab2 SET ...
...
COMMIT WORK           -- end of TX block
```

Informix® version 11.50 introduces savepoints:

```
SAVEPOINT name [UNIQUE]
ROLLBACK [WORK] TO SAVEPOINT [name]
RELEASE SAVEPOINT name
```

**SAP HANA®**

Transactions in SAP HANA®:

- Beginning of transactions are implicit; two transactions are delimited by `COMMIT` or `ROLLBACK`.
- DDL statements can be executed in transactions, but these will persist on rollback.

SAP HANA® (version 2 SPS 04) supports savepoints with similar features as Informix SQL with:

- `SAVEPOINT name`
- `ROLLBACK TO SAVEPOINT name`
- `RELEASE SAVEPOINT name`

**Solution**

The Informix® behavior is simulated with an autocommit mode in the SAP HANA® database interface. A switch to the explicit commit mode is done when a `BEGIN WORK` is performed by the BDL program. Regarding the transaction control instructions, the BDL applications do not have to be modified in order to work with SAP HANA®.

**Note:** Savepoints are not supported.

See also *SELECT FOR UPDATE*

**Related concepts**

*Database transactions* on page 600
Database transactions define a set of SQL instructions to be executed as a whole, or rolled back as a whole.

**Temporary tables**

**Informix®**

Informix® temporary tables are created with the `CREATE TEMP TABLE` DDL instruction or with `SELECT ... INTO TEMP` statement:

```sql
CREATE TEMP TABLE tt1 ( pkey INT, name VARCHAR(50) )
CREATE TEMP TABLE tt2 ( pkey INT, name VARCHAR(50) ) WITH NO LOG
SELECT * FROM tab1 WHERE pkey > 100 INTO TEMP tt2
```

Temporary tables are automatically dropped when the SQL session ends, but they can also be dropped with the `DROP TABLE` command. There is no name conflict when several users create temporary tables with the same name.

BDL REPORTs can create a temporary table when the rows are not sorted externally (by the source SQL statement).

Informix® allows you to create indexes on temporary tables. No name conflict occurs when several users create an index on a temporary table by using the same index identifier.

When creating temporary tables in Informix®, the `WITH NO LOG` clause can be used to avoid the overhead of recording DML operations in transaction logs.

**SAP HANA®**

SAP HANA® supports several types of temporary tables (local/global).

Then equivalent of Informix® temporary tables are session temporary tables created with `CREATE LOCAL TEMPORARY TABLE` instruction:

```sql
CREATE LOCAL TEMPORARY TABLE #tt1 ( pk INT, name NVARCHAR(50) )
```

**Note:** SAP HANA® temporary tables must be specified with a # number sign prefix.

For more details, see the SAP HANA® documentation.

**Solution**

In accordance with some prerequisites, temporary tables creation in BDL programs can be supported with SAP HANA®.

**Important:**

Simple Informix-style SQL statement creating temporary tables can be converted to a native SQL equivalent instruction. However, complex SQL statements such as `SELECT ... INTO TEMP` with subqueries may fail. In such cases, create a view from the complex query and then create the temp table from the view. Or, disable Informix emulation and use the native SQL syntax to create the temporary table (EXECUTE IMMEDIATE "/* fglhint_no_ifxemul */ ...")

With Informix SQL, if the source table has a column defined as `SERIAL` or `BIGSERIAL`, a `SELECT ... INTO TEMP` will produce a new temp table with an auto-incremented serial column. With the `SELECT ... INTO TEMP` emulation for non-Informix databases, not using the native sequence generators (such as `IDENTITY` columns in SQL Server), the resulting temporary table will get a simple `INTEGER` or `BIGINT` column, instead of an auto-incremented column.

The general FGLPROFILE entry to control temporary table emulation is:

```sql
dbi.database.dsname.ifxemul.temptables = {true | false}
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.
**How does it work?**

Informix-specific statements involving temporary table creation are automatically converted to SAP HANA®
CREATE LOCAL TEMPORARY TABLE statements.

The database interface automatically adds a # number sign prefix before the table name. When used in other SQL
statement the temporary table name is also prefixed with a # character.

**Limitations**

- Tokens matching the original table names all get a # number sign prefix in all SQL statements. Make sure you
are not using a temp table name for other database objects, like columns. The following example illustrates this
limitation:

```sql
CREATE TEMP TABLE tmp1 ( col1 INTEGER, col2 CHAR(20) )
SELECT tmp1 FROM table_x WHERE ...
```

- Only the 'native' serial emulation mode is supported with temporary tables.

  The serial emulation based on triggers is not supported, because triggers cannot be created on temporary tables.

  For more details see SERIAL and BIGSERIAL data types on page 1084.

- SAP HANA® does not support index creation on temporary tables.

**Substrings in SQL**

**Informix®**

Informix® SQL statements can use subscripts on columns defined with the character data type:

```sql
SELECT ... FROM tab1 WHERE col1[2,3] = 'RO'
SELECT ... FROM tab1 WHERE col1[10] = 'R' -- Same as col1[10,10]
UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...
SELECT ... FROM tab1 ORDER BY col1[1,3]
```

**Important:** With other database servers as Informix®, when the subscript notation is used to modify column values
in UPDATE statement, or as ORDER BY element, you will get and SQL error:

```sql
UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...
SELECT ... FROM tab1 ORDER BY col1[1,3]
```

**SAP HANA®**

SAP HANA® provides the SUBSTRING(expr, start, length) function, to extract a substring from a string
expression:

```sql
SELECT SUBSTRING(col,10,5) ...
```

**Solution**

Replace all Informix® col[x,y] right-value expressions by SUBSTRING(col,x,y-x+1).

Rewrite UPDATE and ORDER BY clauses using col[x,y] expressions.

The translation of col[x,y] expressions can be controlled with the following FGLPROFILE entry:

```sql
dbi.database.dsname.ifxemul.colsubs = \{ true | false \}
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.
Related concepts
Substring expressions on page 644
Handle substrings expressions with different database engines.

String delimiters

Informix®
The ANSI SQL string delimiter character is the single quote ('string'), while double quotes are used to delimit database object names:

```
SELECT ... WHERE "tabname"."colname" = 'a string value'
```

In Informix® databases created in native mode (non-ANSI), you can use double quotes as string delimiters:

```
SELECT ... WHERE tabname.colname = 'a string value'
```

This is important, since many BDL programs use that character to delimit the strings in SQL commands.

Note: This problem concerns only double quotes within SQL statements. Double quotes used in pure BDL string expressions are not subject to SQL compatibility problems.

SAP HANA®

SAP HANA® follows the ANSI SQL specification, using single quotes for string delimiters and double quotes for database object names.

Solution

When using Static SQL statements, the fglcomp compiler converts string literals using double quotes to string literals with single quotes:

```
$ cat s.4gl
MAIN
  DEFINE n INT
  SELECT COUNT(*) INTO n FROM tab1 WHERE col1 = "abc"
END MAIN
$ fglcomp -S s.4gl
s.4gl^3^SELECT COUNT(*) FROM tab1 WHERE col1 = 'abc'
```

However, SQL statements created dynamically are not modified by the Genero compiler.

The Genero database interface can automatically replace all double quotes by single quotes in SQL statements. This applies to static and dynamic SQL statements.

The The translation of double quoted expression to single quoted expressions can be controlled with the following FGLPROFILE entry:

```
$dbi.database.dbname.ifxemul.dblquotes = \true \false
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

However, database object names must not be delimited by double quotes, because the database interface cannot determine the difference between a database object name and a quoted string! For example, if the program executes the SQL statement:

```
... WHERE "tabname"."colname" = "a string value"
```
replacing all double quotes by single quotes would produce:

```sql
... WHERE 'tabname'.'colname' = 'a string value'
```

This would produce an error since 'tabname'.'colname' is not allowed by ORACLE.

Escaped string delimiters can be used inside strings like the following:

```sql
'This is a single quote: '''
'This is a single quote: \\''
"This is a double quote: ""
"This is a double quote: \""
```

Although double quotes are replaced automatically in SQL statements, it is recommended that you use only single quotes to enforce portability.

**Single row SELECT**

**Informix®**
With Informix®, you must use the system table with a condition on the table id:

```sql
SELECT user FROM systables WHERE tabid=1
```

**SAP HANA®**
To retrieve a single row with SAP HANA®, use DUMMY table:

```sql
SELECT user FROM DUMMY
```

**Solution**
Check the BDL sources for "FROM systables WHERE tabid=1" and use dynamic SQL to resolve this problem.

Consider writing a FUNCTION which produces the FROM and WHERE part, depending on the target database type.

**MATCHES and LIKE**

**Informix®**
Informix® supports MATCHES and LIKE operators in SQL statements.
MATCHES expects * and ? wild-card characters, while LIKE uses the % and _ wild-cards as equivalents.

```sql
( col MATCHES 'Smi*' AND col NOT MATCHES 'R?x' )
( col LIKE 'Smi%' AND col NOT LIKE 'R_x' )
```

MATCHES accepts also brackets notation, to specify a set of matching characters at a given position:

```sql
( col MATCHES '[Pp]aris' )
( col MATCHES '[0-9][a-z]*' )
```

**SAP HANA®**
SAP HANA® does not provide an equivalent of the Informix® MATCHES operator.
The LIKE operator is supported.
Solution

The database driver is able to translate Informix® MATCHES expressions to LIKE expressions, when no [ ] bracket character ranges are used in the MATCHES operand.

The MATCHES to LIKE expression translation is controlled by the following FGLPROFILE entry:

```
dbi.database.dbname.ifxemul.matches = \{ true \} false \}
```

**Important:** Only [NOT] MATCHES followed by a search pattern provided as a string literal can be converted by ODI drivers. A [NOT] MATCHES followed by a ? question mark parameter place holder is not translated!

For maximum portability, consider replacing the MATCHES expressions with LIKE expressions in all SQL statements.

Avoid using CHAR (N) types for variable length character data (such as name, address).

Related concepts

MATCHES and LIKE operators on page 645
Use the standard LIKE operator instead of the MATCHES operator.

The LENGTH() function

Informix®

Informix® provides the LENGTH() function to count the number of bytes of a character string expression:

```
SELECT LENGTH("aaa"), LENGTH(col1) FROM table
```

Informix® LENGTH() does not count the trailing blanks for CHAR or VARCHAR expressions, while Oracle counts the trailing blanks.

Informix® LENGTH() returns 0 when the given string is empty. That means, LENGTH('') = 0.

**SAP HANA®**

SAP HANA® supports the LENGTH() function, but there are some differences with Informix® LENGTH().

The SAP HANA® LENGTH() function counts trailing blanks. When using a NCHAR column, values are blank padded, and the function returns the size of the NCHAR column. When using a NVARCHAR column, trailing blanks are significant, and the function returns the number of characters, including trailing blanks.

When passing NULL as parameter, the SAP HANA LENGTH() function returns NULL.

Solution

Check if the trailing blanks are significant when using the LENGTH() SQL function in your application.

To count the number of characters by ignoring the trailing blanks, you must use the RTRIM() function:

```
SELECT LENGTH(RTRIM(col1)) FROM table
```

The translation of LENGTH() expressions can be controlled with the following FGLPROFILE entry:

```
dbi.database.dsname.ifxemul.length = \{ true \} false \}
```

For more details see IBM Informix emulation parameters in FGLPROFILE on page 668.

Related concepts

The LENGTH() function in SQL on page 647
The semantics of the LENGTH() SQL function differs according to the database engine.

**Row limiting clause**

**Informix®**

Informix® SQL supports the SKIP and FIRST/LIMIT keywords to limit the number of rows of a result set:

```sql
SELECT SKIP 10 FIRST 20 customer.* FROM customer ... ORDER BY cust_name
```

This Informix SQL syntax is not portable.

Recent database engines support the row limiting clause syntax defined by the SQL standard:

```sql
SELECT ... OFFSET n ROWS FETCH FIRST m ROWS ONLY
```

This should be the preferred syntax to be used, if all target database types support this SELECT clause.

The ODI database drivers can convert the Informix SQL SKIP/FIRST row limiting clause to a native SQL equivalent, if the row limiting clause parameters are simple integer literals (the clause is not translated when using SQL parameters / program variables).

**Important:** In addition to the SKIP/FIRST clause of the projection clause, Informix SQL supports also a LIMIT clause after the ORDER BY clause:

```sql
SELECT customer.* FROM customer ... ORDER BY cust_name LIMIT 10
```

This Informix SQL syntax construction is not converted by the ODI drivers. To benefit from the conversion, review the code to use the Informix SQL SKIP/FIRST clause instead.

**SAP HANA®**

SAP HANA® supports the following row limiting clause:

```sql
SELECT ... ORDER BY ... LIMIT m [ OFFSET n ]
```

**Solution**

The Informix SQL row limiting clause can be converted by the SAP HANA® driver to the native SQL equivalent clause, when the parameters are simple integer literals.

**Note:** The row limiting clause must not use SQL parameters. Only row limiting clauses using integer constants will be converted.

The translation of the Informix SQL row limiting clause can be controlled with the following FGLPROFILE entry:

```python
dbi.database.dsnname.ifxemul.rowlimiting = true false
```

For more details see [IBM Informix emulation parameters in FGLPROFILE](#) on page 668.

**Related concepts**

Row limiting clause (SELECT) on page 650

How to use the right clause to limit the number of rows produced by a SELECT statement?

**BDL programming**

SAP HANA® related programming topics.
INSERT cursors

Informix®

Informix® provides insert cursors to optimize row creation in a database. An insert cursor is declared as a cursor, and rows as added with the PUT instruction. The rows are buffered and sent to the database server when executing a FLUSH instruction, or when the cursor is closed with CLOSE. When using transactions in Informix®, the OPEN, PUT and FLUSH instructions must be executed within a transaction block.

```
DECLARE c1 CURSOR FOR INSERT INTO tab1 ...
BEGIN WORK
OPEN c1
WHILE ...
  PUT c1 USING var-list
END WHILE
CLOSE c1
COMMIT WORK
```

SAP HANA®

SAP HANA® does not support insert cursors.

Solution

Insert cursors are emulated by the database interface, using basic INSERT SQL instructions.

The performances might be not as good as with Informix®, but the feature is fully supported.

Related concepts

Insert cursors on page 633
Using insert cursors with non-Informix databases.

Cursors WITH HOLD

Informix®

Informix® closes opened cursors automatically when a transaction ends, unless the WITH HOLD option is used in the DECLARE instruction:

```
DECLARE c1 CURSOR WITH HOLD FOR SELECT ...
OPEN c1
BEGIN WORK
FETCH c1 ...
COMMIT WORK
FETCH c1 ...
CLOSE c1
```

HANA DB®

HANA DB® does not close cursors when a transaction ends.

Solution

BDL cursors declared WITH HOLD remain open even after terminating a transaction with a COMMIT WORK or ROLLBACK WORK.

For consistency with other database brands, database cursors that are not declared WITH HOLD are automatically closed, when a COMMIT WORK or ROLLBACK WORK is performed.
**Important:** Opening a WITH HOLD cursor declared with a SELECT FOR UPDATE results in an SQL error; in the same conditions, this does not normally appear with Informix®. Review the program logic in order to find another way to set locks.

**Related concepts**

Cursors WITH HOLD on page 631  
Programming WITH HOLD cursors using SELECT with and without FOR UPDATE clause.

**SELECT ... FOR UPDATE**

**Informix®**

Legacy BDL programs typically use a cursor with SELECT FOR UPDATE to implement pessimistic locking and avoid several users editing the same rows:

```sql
DECLARE cc CURSOR FOR 
SELECT ... FROM tab WHERE ... FOR UPDATE 
OPEN cc 
FETCH cc <-- lock is acquired 
... 
CLOSE cc <-- lock is released
```

The row must be fetched in order to set the lock.

If the cursor is local to a transaction, the lock is released when the transaction ends. If the cursor is declared WITH HOLD, the lock is released when the cursor is closed.

Informix® provides the SET LOCK MODE instruction to define the lock wait timeout:

```sql
SET LOCK MODE TO \{ WAIT \| NOT WAIT \| WAIT seconds \}
```

The default mode is NOT WAIT.

**SAP HANA®**

SAP HANA® supports SELECT FOR UPDATE.

SAP HANA® releases locks when the transaction ends (at COMMIT WORK or ROLLBACK WORK).

SAP HANA®'s locking granularity is at the row level.

**Note:** When using a cursor declared with a SELECT FOR UPDATE, all rows of the result set will get an exclusive lock when the cursor is opened.

SAP HANA® has an equivalent for Informix® SET LOCK MODE TO NOT WAIT:

```sql
SET TRANSACTION LOCK WAIT TIMEOUT milliseconds
```

**Solution**

The database interface is based on an emulation of an Informix® engine using transaction logging. Therefore, opening a SELECT ... FOR UPDATE cursor declared outside a transaction will raise an SQL error -255 (not in transaction).

If the program is doing a SET LOCK MODE instruction, it will be converted to a SET TRANSACTION LOCK WAIT TIMEOUT command by the database interface.

**Important:** Since SAP HANA® sets locks when the cursor is opened, it is not possible to get the Informix® behavior, where locks are set at FETCH time.
LOAD and UNLOAD

Informix®
Informix® provides two SQL instructions to export / import data from / into a database table:

The UNLOAD instruction copies rows from a database table into a text file:

```
UNLOAD TO "filename.unl" SELECT * FROM tab1 WHERE ..
```

The LOAD instructions insert rows from a text file into a database table:

```
LOAD FROM "filename.unl" INSERT INTO tab1
```

SAP HANA®
SAP HANA® does not provide LOAD and UNLOAD instructions.

Solution
LOAD and UNLOAD instruction are implemented in the Genero BDL runtime system with basic INSERT (for LOAD) or SELECT (for UNLOAD) SQL commands. The LOAD and UNLOAD instruction can be supported with various database servers.

However, LOAD and UNLOAD require the description of the column types in order to work, that can lead to some differences in the data formatting.

Note: If no transaction is started, the LOAD instruction will automatically execute a BEGIN WORK and COMMIT WORK when finished, or ROLLBACK WORK if a row insertion failed while loading. Terminating a transaction will automatically close cursors not defined WITH HOLD option. To workaround this situation, see more details in the LOAD on page 736 reference topic.

The LOAD and UNLOAD BDL instructions are supported with SAP HANA®.

Related concepts
LOAD and UNLOAD instructions on page 639
The LOAD and UNLOAD instructions can produce different data formats depending on the database server type.

UPDATE/DELETE ... WHERE CURRENT OF

Informix®
Informix® allows positioned UPDATEs and DELETEs with the "WHERE CURRENT OF cursor" clause, if the cursor has been DECLARED with a SELECT ... FOR UPDATE statement.

SAP HANA®
SAP HANA® supports UPDATE/DELETE with the WHERE CURRENT OF cursor clause in SQLScript, but not in pure SQL and ODBC client, used by the Genero ODI driver.

Solution
In pure SQL, UPDATE/DELETE ... WHERE CURRENT OF is not supported with SAP HANA®.

As a replacement of WHERE CURRENT OF, if the database table is defined with a primary key column, use the value fetched from the SELECT [FOR UPDATE] cursor in the WHERE clause of the UPDATE/DELETE statement.

Related concepts
Positioned UPDATE/DELETE on page 631
Using positioned updates/deletes with named database cursors.

**SQL Interruption**

**Informix®**
With Informix®, it is possible to interrupt a long running query if the `SQL INTERRUPT ON` option.

**SAP HANA®**
SAP HANA® supports SQL Interruption: The db client must issue an `SQLCancel()` ODBC call to interrupt a query.

**Important:** With SAP HANA, when an SQL statement is interrupted, the complete SQL transaction is rolled back: SAP HANA will cancel the SQL statements executed in the transaction before the interrupted SQL statement.

**Solution**
The SAP HANA® database driver supports SQL interruption and converts the native SQL error code -139 to the Informix® error code -213.

If your application allows SQL interruption in transaction blocks delimited by `BEGIN WORK / COMMIT WORK`, the code needs to be reviewed, because SQL statements prior to the interrupted SQL statement will be rolled back.

**Related concepts**
**Using SQL interruption** on page 601
Interrupt long running SQL queries, or interrupt queries waiting for locked data.

**Scrollable Cursors**

**Informix®**
Informix® SQL and Genero BDL support scrollable cursors when you specify the `SCROLL` clause in the `DECLARE` cursor instruction:

```
DECLARE c1 SCROLL CURSOR FOR SELECT ... 
```

**Important:** Informix does not allow to fetch TEXT/BYTE columns with scrollable cursors. If you declare a scroll cursor with a `SELECT` containing TEXT/BYTE columns, Informix will produce the SQL error -611 when executing the `OPEN` instruction.

**SAP HANA®**
SAP HANA® does not support scrollable cursors.

**Solution**
The SAP HANA® database driver emulates scrollable cursors by fetching rows in a temporary file.

**Important:** With SAP HANA is it NOT possible to use LOB columns in a scrollable cursor. If TEXT/BYTE columns are used with a scrollable cursor, the `OPEN` instruction will produce the SQL error -611 (as with Informix).

See **Scrollable cursors** on page 623 for more details about scroll cursor emulation.

**Stored procedure calls**
SAP HANA® supports stored procedures as a saved collection of SQL statements, which can accept and return user-supplied parameters.

SAP HANA® stored procedures can also produce one result set that can be accessed through a result view.
To create a stored procedure in SAP HANA® database, use the `CREATE PROCEDURE` statement. See reference documentation for more details.

**Stored procedures with output parameters**

SAP HANA® stored procedures must be called with the input and output parameters specification in the **USING** clause of the `EXECUTE`, `OPEN` or `FOREACH` instruction. As in normal dynamic SQL, parameters must correspond by position and the `IN/OUT/INOUT` options must match the parameter definition of the stored procedure.

To execute the stored procedure, you must use the `CALL` SQL instruction:

```
PREPARE stmt FROM "call proc1(?,?,?)"
```

Here is a complete example creating and calling a stored procedure with output parameters:

```
MAIN
DEFINE n INTEGER
DEFINE d DECIMAL(6,2)
DEFINE c VARCHAR(200)
DATABASE test1
EXECUTE IMMEDIATE
   "create procedure proc1(" 
   "    in p1 int," 
   "    out p2 decimal(6,2)," 
   "    inout p3 varchar(20)" 
   " )"
   " language sqlscript as"
   " begin"
   "    p2 := p1 + 0.23;"
   "    p3 := 'Value = ' || cast(p1 as varchar(10));"
   " end"
PREPARE stmt FROM "call proc1(?,?,?)"
LET n = 111
EXECUTE stmt USING n IN, d OUT, c INOUT
DISPLAY d
DISPLAY c
END MAIN
```

**Stored procedures with result set**

With SAP HANA®, you can execute stored procedures returning a result set.

To do so, you must:

1. Define a user type with `CREATE TYPE`, with the same structure as the result set.
2. DECLARE a cursor to fetch the rows from the result view associated to the stored procedure (the name of the result view is defined by the **WITH RESULT VIEW** clause in `CREATE PROCEDURE`.
3. The `SELECT` statement used by the cursor must specify named input parameters with the **PLACEHOLDER** keyword, and since parameter names must be enclosed in double quotes, you need to disable the Informix emulation with the `fglhint_no_ifxemul` special hint.

**Note:** Pay attention to VARCHAR/NVARCHAR translation done by the driver, to support non-ASCII character sets (the type must be created with `NVARCHAR` since the SQL translator does makes the conversion only for `CREATE TABLE`)

```
MAIN
DEFINE i, n INTEGER
DEFINE d DECIMAL(6,2)
DEFINE c VARCHAR(200)
DATABASE test1
CREATE TABLE tab1 ( c1 INTEGER, c2 DECIMAL(6,2), c3 VARCHAR(200) )
INSERT INTO tab1 VALUES ( 1, 123.45, 'aaaaaa' )
INSERT INTO tab1 VALUES ( 2, 123.66, 'bbbbbbb' )
```
INSERT INTO tab1 VALUES (3, 444.77, 'cccccc')
EXECUTE IMMEDIATE "create type type_tab1 as table (c1 INTEGER, c2 DECIMAL(6,2), c3 NVARCHAR(200))"
EXECUTE IMMEDIATE "create procedure proc2( in key integer, out rs type_tab1 )"
|| " language sqlscript reads sql data with result view
proc2_view as"
|| " begin"
|| " rs = select * from tab1 where c1 > :key;"
|| " end"
DECLARE curs CURSOR FROM "/* fglhint_no_ifxemul */ select * from proc2_view(placeholder."$$key$$"=>?)"
LET i = 1
FOREACH curs USING i INTO n, d, c
DISPLAY n, d, c
END FOREACH
END MAIN

**Stored procedures with output parameters and result set**

**Important:** It is no possible to execute SAP HANA® stored procedures with output parameters and a result set.

---

## User interface

These topics cover programming the user interface (UI) with the Genero Business Development Language.

### User interface basics

This section introduces to the foundation of the Genero user interface.

### The dynamic user interface

The dynamic user interface is the base concept of the Genero user interaction components.

The *dynamic user interface* (DUI) concept implements a flexible graphical user interface programming toolkit, based on the usage of XML standards, to define an abstract representation of the application forms. This abstract user interface representation can be displayed by different types of display devices called front-ends, which execute on the user workstation or on the same platform as the runtime system.

The abstract definition of the user interface can be manipulated at runtime as a tree of interface objects. This tree is called the abstract user interface tree.

The runtime system is in charge of the abstract user interface tree, and the front-end is in charge of rendering this abstract tree on the screen. The front-end gets a copy of that tree which is automatically synchronized by the runtime system by using the front-end protocol.

In development, application screens are defined by form specification files.

The following schema describes the dynamic user interface concept, showing how the abstract user Interface tree is shared by the runtime system and the front-end.
Figure 38: AUI tree shared between the runtime system and front-end

Note: When the AUI trees are synchronized, only the changes are sent to or received from the front-end.

Note that when running on a mobile device, both front-end and runtime system execute on the same platform. Still the AUI tree protocol takes place, and both components perform the tasks they are dedicated to.

Related concepts
Form specification files on page 1237
Form specification files are the source files defining the layout and content of application forms.

The abstract user interface tree

The abstract user interface tree is the XML representation of the application forms displayed to the end user.

The abstract user interface tree (AUI tree) is an DOM tree describing the objects of the user interface elements of a program at a given time.

A copy of the AUI tree is held by both the front-end and the runtime system.

AUI tree synchronization is automatically done by the runtime system using the front-end protocol, when the control goes back to the end user. If a program needs to synchronize the display while processing, it can use the `ui.Interface.refresh()` method.

The programs can manipulate the AUI tree element by using XML utility classes or high-level built-in classes such as `ui.Dialog` and `ui.Form`.

What does the abstract user interface tree contain?

The abstract user interface defines a tree of objects organized by parent/child relationship. The different kinds of user interface objects are defined by attributes. The AUI tree can be serialized as text based on the XML standard notation.

The following example shows a part of an AUI tree defining a toolbar serialized with the XML notation:

```xml
<ToolBar>
  <ToolBarItem name="f5" text="List" image="list" />
  <ToolBarSeparator/>
  <ToolBarItem name="Query" text="Query" image="search" />
  <ToolBarItem name="Add" text="Append" image="add" />
...  
</ToolBar>
```
Related concepts
Manipulating the abstract user interface tree on page 1107

Manipulating the abstract user interface tree

Modifying the AUI tree with user interface specific built-in classes

The objects of the abstract user interface tree can be queried and modified at runtime with specific built-in classes like ui.Form, provided to manipulate form elements.

The code in the example gets the current window object, then gets the current form in that window, and hides a group-box form element identified by the name "gb1":

```
DEFINE w ui.Window
DEFINE f ui.Form
LET w = ui.Window.getCurrent()
LET f = w.getForm()
CALL f.setElementHidden("gb1",1)
```

Using the user interface specific built-in classes is the recommended way to modify the AUI tree in your programs.

Using low-level APIs to modify the AUI tree

In very special cases, you can also directly access the nodes of the AUI tree by using DOM built-in classes like om.DomDocument and om.DomNode.

Important: As FOUR Js continues to add new features to Genero, we encounter situations that forces us to modify the AUI Tree to add new element types and attributes. If you are using the low level API to directly modify user interface elements, your code may be impacted when changes are made in the AUI tree definition. To minimize the impact, consider the following course of action with regards to use of the DOM/SAX API:

1. During a dialog instruction, do not create or delete AUI tree nodes, or change attributes that are not for decoration only (text, color and styles can be changed during dialog). When possible, consider using ui.Dialog and ui.Form built-in class methods instead of the low-level DOM/SAX API.
2. Place all custom changes to the DOM/SAX API within centralized library functions that are accessible to all modules, as opposed to scattering DOM/SAX calls throughout your code base.
3. Do not create nodes or change attributes that are not explicitly documented as modifiable. For example, TopMenu or ToolBar nodes can be created and configured dynamically, but it is not recommended to add FormField nodes to existing forms, or modify the active attribute of fields or actions. Do not create AUI nodes, that would not be created by a .42f file produced by the fglform compiler from a .per source file.
4. AUI tree nodes that have already been sent to the front-end can only be modified by updating, appending or deleting children nodes. Inserting nodes with the om.DomNode.insertBefore() method is not supported in the AUI tree synchronization protocol, once the parent node has been sent.

To get the user interface nodes at runtime, the language provides different kinds of API functions or methods, depending on the context. For example, to get the root of the AUI tree, call the ui.Interface.getRootNode() method. You can also get the current form node with ui.Form.getNode() or search for an element by name with the ui.Form.findNode() method.

Related concepts
Dialog instructions on page 1452

This section describes the dialog instructions to control application forms and the concepts related to dialog implementation.

XML node types and attribute names

XML is case-sensitive. By convention, node types in the AUI tree use uppercase/lowercase combinations to indicate word boundaries. Therefore, the nodes and attributes of an abstract user interface tree are handled as follows:
- Node types - the first letter of the node type is always capitalized. Subsequent letters are lowercase, unless the type consists of multiple words joined together. In that case, the first letter of each of the multiple words is capitalized (the camel-case convention). Examples: Label, FormField, DateEdit, Edit.
- Attribute names - the first letter of the name is always lowercase; subsequent letters are also lowercase, unless the name consists of multiple words joined together. In that case, the first letter of each subsequent word is capitalized (the Lower camel-case convention). Examples: text, colName, width, tabIndex
- Attribute values - the values are enclosed in quotes, and the runtime system does not convert them.

If you reference AUI tree XML nodes or attributes in your code, you must always respect the naming conventions.

**Actions in the abstract user interface tree**

The abstract user interface identifies all possible actions that can be received by the current interactive instruction with a list of Action nodes. The list of possible actions are held by a Dialog node. An Action node is identified by the 'name' attribute and defines common properties such as the accelerator key, default image, and default text.

Interactive elements are bound to Action nodes by the 'name' attribute. For example, a toolbar button (a.k.a toolbar item) with the name 'cancel' is bound to the Action node having the name 'cancel', which in turn defines the accelerator key, the default text, and the default image for the button.

![AUI Tree binding](image)

**Figure 39: AUI Tree binding**

When an interactive element is used (such as a form field input, toolbar button click, or menu option selection), an ActionEvent node is sent to the runtime system. The name of the ActionEvent node identifies what Action occurred and the 'idRef' attribute indicates the source element of the action.

**Related concepts**
Dialog programming basics on page 1706
This section describes basic dialog programming concepts.

**Inspecting the AUI tree of a front-end**

**AUI tree introspection**

When executing a program displaying on a front-end, it is possible to inspect the content of the abstract user interface built on the front-end side. The way to show the AUI tree depends on the type of front-end.

**Genero Desktop Client**

The GDC must have been started in debug mode (-aD option).
In the current window of the running program, do a control-right-click with the mouse. This opens the AUI tree debug window.

You can then browse the AUI tree created on the GDC side.

**Genero Browser Client**

The GAS / GBC must have been started with debug option. In the *as.xcf* configuration file, add the following line:

```xml
<APPLICATION_SERVER>
...
  <RESOURCE Id="res.uaproxy.param" Source="INTERNAL">--development</RESOURCE>
...
</APPLICATION_SERVER>
```

Start your application in a web browser; a debug icon appears on the right of the window. Click the icon to display the AUI debug tree.

You can then browse the AUI tree created on the GBC side.

**Genero Mobile for Android™**

The GMA must execute with debug mode enabled in the settings panel.

Open a web browser and enter the following URL:

```
http://device-ip-address:6480
```

You can then browse the AUI tree created on the GMA side.

**Genero Mobile for iOS**

The GMI must have been started in debug mode: the debug option needs to be enabled in GMI app settings on the device.

Open a web browser and enter the following URL:

```
http://device-ip-address:6480 (or 6400)
```

You can then browse the AUI tree created on the GMI side.

**Genero user interface modes**

User interface modes allow you to adapt the application form rendering to different types of displays.

**Text mode rendering (TUI mode)**

**What is the text mode (TUI)?**

The text user interface (TUI) has been designed for character-based terminals. This mode can be used to run your application on a text terminal hardware or in a terminal emulator.

In TUI mode, all application forms will display within the current terminal device or emulator as shown.
Enabling text mode

In order to run a Genero program on text mode, set the FGLGUI environment variable to 0 (zero).

**Important:** On UNIX™ platforms, you need to configure your terminal capabilities with the TERM, TERMINFO or TERMCAP environment variables.

Checking for text mode in programs

In the program code, use the `ui.Interface.getFrontEndName()` method to query for the front-end type. When this method returns "console", the program executes in text mode.

Related concepts

*Using a text terminal* on page 1123
This section covers topics about text terminal configuration when using the TUI mode.

Graphical mode rendering (GUI mode)

What is the graphical mode (GUI)?

Genero supports the *Graphical User Interface (GUI) mode* to display application windows and forms with a real graphical look and feel, for desktop workstation, web browsers and mobile front-end platforms.

The graphical mode supports additional options such as native or universal rendering mode, and modern or traditional mode, to adapt to the needs of your application.

Enabling the graphical mode

In order to use the graphical mode, the FGLGUI environment variable can be set to 1.

However, graphical mode is the default with Genero. The FGLGUI environment variable is usually set to 0 (zero) to run the application in text mode.
**Defining the target front-end**

In graphical mode, the application forms are displayed on the front-end workstation identified with the `FGLSERVER` environment variable.

If this variable is not defined, the runtime system (`fglr`un) assumes that the front-end executes on the same computer.

**Graphical mode options**

When using the Graphical mode, the default is the Native rendering: front-ends use native platform GUI objects (windows and widgets) to render application forms.

To unify front-end display, use the Universal Rendering mode and get the same HTML-based rendering on all sort of Genero front-ends.

To simplify migration from text mode to graphical mode with legacy applications, Genero supports the Traditional GUI mode option to render all application windows in a single front-end GUI window.

**Checking for graphical mode in programs**

In the program code, use the `ui.Interface.getFrontEndName()` method to query for the front-end type.

When this method return a value different from "console", the program executes in graphical mode.

**Graphical mode with Native Rendering**

**What is the Native rendering mode?**

Genero supports the Native Rendering mode to get a graphical look and feel based on platform GUI widgets, for desktop workstation and mobile front-end platforms.
Enabling the Native Rendering mode

The Native Rendering mode is the default.

Use the Universal Rendering mode, to get the same rendering on all Genero front-ends, using the Genero Browser Client (GBC) front-end.

Graphical mode with Universal Rendering

What is the Universal Rendering mode?

In graphical mode, each type of front-end uses by default the native Graphical User Interface API and widgets of the platform or framework used to implement the front-end. This is called the Native Rendering mode.

Genero supports also the Universal Rendering mode, to unify the display of application forms on all front-ends, with a common rendering solution based on the GBC front-end using web technologies.
Enabling Universal Rendering with FGLPROFILE: gui.rendering

The Universal Rendering mode is enabled with the following FGLPROFILE entry:

```
gui.rendering = "universal"
```

One can explicitly force Universal Rendering to be disabled by setting this entry to "native":

```
gui.rendering = "native"
```

Note: The gui.rendering option takes precedence over the Universal Rendering option in the front-end settings (when available). If the front-end option enables Universal Rendering by default, and gui-rendering="native" is set, the application forms will be displayed in Native Rendering mode.

Defining the GBC to be used in direct mode

When using Universal Rendering mode in direct mode, it is possible to configure which GBC version has to be transmitted by the runtime system to the front-end.

The GBC component files must be located on the computer where the fglrun program executes, and will be transferred when the application starts.

The GBC component will be searched in the following directories:

1. The appdir/gbc directory, where appdir is the directory where the program file is located,
2. The directory defined in the FGLGBCDIR environment variable,
3. The $FGLDIR/web_utilities/gbc/gbc directory.
If defined, make sure that FGLGBCDIR is set to a directory containing the GBC component files.

**Note:** The FGLGBCDIR environment variable is provided to select a given GBC in a development context when using the GUI direct mode. Do not use FGLGBCDIR in a production context: If a specific GBC is needed in production, consider shipping it in `appdir/gbc`. Otherwise, use the GBC available by default.

**Tip:** Set the FGLGUIDEBUG environment variable, to enable GUI protocol debug logging and verify which GBC is transmitted to the front-end by the runtime system. Search for log lines like:

```
*** uic_FT_processGet.1188: requestedName=gbc://index.html realName=/app/gbc/index.html
```

### Defining the GBC to be used with the GAS

When executing applications through the Genero Application Server, displaying on the GDC, GMA or GMI front-ends in Universal Rendering mode, the GBC component files are found with the mechanism available in the GAS. The `$APPDIR/gbc` directory is the recommended location for a production environment, when a specific GBC is required (typically, when using GBC customization).

Otherwise, if no specific GBC is required, the default GBC set in the GAS configuration will be used.

### Building mobile apps with a specific GBC

When building a GMA or GMI app using Universal Rendering, the GBC component needs to be bundled with the app package.

The `gmbuildtool` and `gmibuildtool` commands support an option to specify the GBC to be bundled with the app.

If this build option is used, the app will by default display in Universal Rendering mode: There is no need to define the `gui.rendering = "universal"` FGLPROFILE entry.

For more details, see Building Android apps with Genero on page 3591, Building iOS apps with Genero on page 3606.

### Checking for Universal Rendering in programs

In the program code, use the following APIs to check if Universal Rendering mode is used:

1. `ui.Interface.getUniversalClientName()`
2. `ui.Interface.getUniversalClientVersion()`

**Note:** These methods are especially useful for debugging purpose. Consider writing code that is not dependent on the Native Rendering or Universal Rendering mode.

**Related concepts**

- Graphical mode rendering (GUI mode) on page 1110
- Mobile development mode on page 3568

Set up a development environment to display app forms on a mobile front-end.

**Graphical mode with Traditional Display**

**What is the Traditional GUI mode?**

By default when using the graphical mode, application forms are rendered as real movable and resizable windows, form labels and fields become widgets using variable fonts, toolbars and pull-down menus are displayed, and error messages are displayed in the status bar.

However, this default graphical rendering is not always adapted when migrating from a project that was developed for dumb terminals (TUI mode).
In such case, the *Traditional GUI mode* can be used to ease migration from TUI based applications to GUI mode.

With the Traditional GUI mode, application windows bound to forms using a **SCREEN** section will be displayed as simple boxes in a main front-end window. Other windows bound to forms defined with the **LAYOUT** section will be displayed as new GUI windows.

![Traditional GUI mode example](image)

**Figure 43: Traditional GUI mode example**

**Enabling the Traditional GUI mode**

The Traditional GUI mode can be enabled with the following **FGLPROFILE** entry:

```plaintext
gui.uiMode = "traditional"
```

By default, the Traditional GUI mode is off.

**Window rendering rules**

If the Traditional GUI mode is enabled, the **OPEN WINDOW** statement works differently depending on the layout type of bound forms.

On the front-end side, there is one unique main graphical window (a top-level widget called "compatibility window container") created to host all the windows created by a program. Traditional forms are form files which have a **SCREEN** section instead of the **LAYOUT** section. When migrating from a TUI mode project, all forms initially contain a **SCREEN** section; hence all windows opened in traditional mode will appear in the compatibility window container.

To rebuild a form file with graphical items such as group boxes, buttons and tables, use a **LAYOUT** section. If the rebuilt form file is loaded via **OPEN WINDOW ... WITH FORM form-file** then, even in traditional mode, the newly created window will appear as a new top-level widget on the front-end side. This opens a smooth migration path using the traditional mode; as a first step, it is possible to migrate and enhance some application forms like typical search lists, while keeping the rest of the application forms running in the traditional rendering.

Note, however, that following instructions do not work in Traditional GUI mode:

1. **OPEN WINDOW window_id AT line, column WITH height ROWS, width COLUMNS**
2. **OPEN FORM form_id FROM "form_file"**
3. DISPLAY FORM form_id

A runtime error results, because you cannot display a form with dynamic geometry in a fixed geometry container.
Only forms with a SCREEN section can be displayed at a later stage in a window that was initially opened inside the compatibility window container.

**Function key shifting**

When the Traditional GUI mode is enabled, you can map Shift-Fx and Ctrl-Fx key strokes to F(x+offset) actions. The offset is defined with the gui.key.add_function entry:

```plaintext
gui.key.add_function = 12
```

This entry defines the number of function keys of the keyboard (default is 12). When defined as 12, a Shift-F1 will be received as an F13 (12+1) action event by the program, and a Control-F1 will be F25 (12*2+1).

**Related concepts**

[Graphical mode rendering (GUI mode)](on page 1110)

**GUI front-end connection**

This section explains runtime to front-end connection in its simplest form.

**Connecting with a front-end**

In graphical mode, depending on the front-end technology that is used (desktop client, mobile client, web server client), there are different solutions to establish the connection between the runtime system and the front-end.

This topic describes the development context case, where programs are executed directly with `fglr` run. In a production environment, programs will typically be started with another technology, since the execution of programs will be triggered by the end user interacting with the front-end. Read front-end specific documentation for more details.

From the point of view of the runtime system, the front-end acts as a graphical server and thus the programs must connect to that GUI server in order to display forms and get user input.

The runtime system will try to connect to the front-end only when the first interactive instruction like `MENU` or `INPUT` is reached.

For the runtime system, the front-end is identified by the `FGLSERVER` environment variable. This variable defines the host name of the machine where the front-end resides, and the number of the front-end instances to be used.

The syntax for `FGLSERVER` is:

```plaintext
{hostname|ip-address}[:servername]
```

For example:

```bash
$ FGLSERVER=fox:1
$ fglrun myprog
```

The `servername` parameter is a whole number that defines the instance of the front-end. It is actually defining a TCP port number the front-end is listening to, starting from 6400. For example, if `servername` equals 2, the TCP port number used is 6402 (6400+2).

This is the standard/basic connection technique, but you can set up different types of configurations. For example, you can have the front-end connect to an application server via `ssh`, to pass through firewalls over the internet. Refer to the front-end documentation for more details.

There is an exception to the standard `FGLSERVER` specification, if the front-end is denied permission to listen to a TCP port (this is the case with the Genero Mobile Development Client). If you need to revert the connection principle
in this particular case, use the \texttt{--gui-listen} option of \texttt{fglrun}. With this option, the runtime system will listen to the specified port, so the front-end can bind to the program and start to use the GUI protocol. The procedure to work in such configuration is the following:

1. Start the program with:
   \begin{verbatim}
   fglrun \texttt{--gui-listen=tcp-port} prog-name
   \end{verbatim}
2. Connect from the front-end, for example, with an URL using the following format:
   \begin{verbatim}
   fgl://dev-server-hostname:tcp-port
   \end{verbatim}

Related concepts

Mobile development mode on page 3568
Set up a development environment to display app forms on a mobile front-end.

Graphical mode rendering (GUI mode) on page 1110

The front-end protocol

The \textit{front-end protocol} (FEP) is an internal protocol used by the runtime system to synchronize the abstract user interface (AUI) representation on the front-end side. This protocol defines a simple set of operations to modify the AUI tree. This protocol is based on a command processing principle (send command, receive answer) that can be serialized for transport over any network protocol, like HTTP for example.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure44.png}
\caption{Typical communication between the Runtime System and the front-end}
\end{figure}

1. Initialization phase: The runtime system sends the initial AUI tree.
2. The front-end builds the graphical user interface based on the AUI tree.
3. The front-end waits for a user interaction (mouse click, keyboard typing).
4. When the user performs some interaction, the front-end sends front-end events corresponding to the modifications made by the user.
5. Front-end events are analyzed and validated by the runtime system.
6. The runtime system sends back the result of the front-end requests, by way of AUI tree modification commands.
7. When receiving these commands, the front-end modifies its version of the AUI tree and updates the graphical user interface. It then waits for new user interactions (step 3).
**Front-end identification**

To start a program from the front-end platform, the front-end can open a terminal session on the application server. This is done, for example, by using a `ssh`, `rlogin`, or `telnet` terminal session. When the terminal session is open, the front-end sends some shell commands to set environment variables, like `FGLSERVER`, before starting the program to display the application forms on the front-end where the terminal session was initiated.

In this configuration, front-end identification takes place. The front-end identification prevents the display of application forms on a front-end that did not start the program on the server. If the front-end is not identified, a serious security issue can arise, as anyone could run a fake program to display on any front-end and ask for a password.

**Important:** Front-end identification is achieved automatically by an initial protocol handshake. However, there can be a security hole if regular operating system users on the application server can overwrite the program or the shell script started by the front-end terminal session. Malicious programs can try to display the application on another workstation to read confidential data. As long as basic application users do not have read and write privileges on the program files, there is no risk. To make sure that program files on the server side are protected from basic users, create a special user on the server to manage the application program files, and give other users only read access to those files. As long as basic users cannot modify programs on the server side, there is no security issue.

**Related concepts**

- **FGLSERVER** on page 280
  Defines the graphical front-end for the application.

**GUI connection timeout**

When initiating the connection to the front-end, if the front-end software is stopped, the host machine is down, or a firewall drops connections for the TCP port used for the GUI connection, the program will stop with an error after a given timeout.

This timeout can be specified with the following `FGLPROFILE` entry:

```plaintext
gui.connection.timeout = seconds
```

The default timeout is 30 seconds.

**Related concepts**

- **The FGLPROFILE file(s)** on page 255
  FGLPROFILE environment variable defines Genero BDL configuration files

**Wait for front-end ping timeout**

Define the wait-for-ping timeout of the runtime system to control the built-in "keep alive" solution provided by Genero.

**Important:** This feature is not supported when running on mobile devices, or when displaying applications on mobile devices.

It can happen that the user leaves the program for a while without using it. The network policy (firewall) might force a close of the TCP connection after a given period of inactivity. To avoid such connection shutdown when there is no GUI exchange, the front-end sends successive ping events after a timeout period of inactivity, to keep the TCP connection alive. The inactivity timeout can be configured in front-ends. The front-end ping is normal behavior and part of the GUI client/server protocol.

**Important:** With this keep alive technique, a front-end connection always remains open, even if the user leaves the workstation for several hours. If your network connection comes at a cost to support this type of connection, it is recommended that you consider configuring the front-end to turn off the ping event or stop it after a given number of pings. Check the front-end configuration documentation for more details.

If the front-end program is not stopped properly as happens when, for example, it is killed by a system reboot, or when there is network failure, the TCP connection is lost and the runtime system no longer receives ping events. In
In this case, the runtime system waits for a specified time, before it raises the error -8063. By default (when no exception handler is defined), the runtime system just stops. However, the error -8063 can be trapped with a WHENEVER ERROR or TRY/CATCH block, to perform some finalization tasks. The program should then immediately stop with EXIT PROGRAM n (where n > 0).

The wait-for-ping timeout period of the runtime system can be configured with the gui.protocol.pingTimeout fglprofile entry. By default, the runtime system waits for 600 seconds (10 minutes):

```
gui.protocol.pingTimeout = 800
```

**Important:** If you set the wait-for-ping timeout to a value lower than the ping delay of the front-end, the program will stop with the -8063 error after that timeout, even if the TCP connection is still alive. For example, when a front-end has a ping delay of 5 minutes, the recommended minimum value for this parameter is about 330 seconds (5 minutes + 30 seconds to make sure the client ping arrives).

**Related concepts**
- The FGLPROFILE file(s) on page 255
- FGLPROFILE environment variable defines Genero BDL configuration files

**GUI protocol compression**

**GUI protocol compression** might be used to reduce the amount of data exchanged between the front-end and the runtime system. Compression is typically useful on slow networks. The compression algorithm is provided by the standard ZLIB library of the system.

When using the Genero Browser Client (GBC), compression is not useful and is automatically disabled.

Compression makes sense on slow networks (for example, with a phone-line dialup modem, or broadband modem based networks); On fast networks, compression is not required and will in fact use unnecessary processor time.

Compression is disabled by default, and can be enabled with this FGLPROFILE entry:

```
gui.protocol.format = "zlib"
```

If this parameter is defined, but the ZLIB library is not installed on your system or if the ZLIB version is not compatible with the version needed by the runtime system, compression cannot be supported, and the program will stop with error -6317. The ZLIB version must be 1.2.5 (or compatible with version 1.2.5). On Microsoft™ Windows® platforms, the name of the library must be ZLIB1.DLL. Precompiled binary packages can easily be found on the internet. On UNIX™ platforms, the name of the shared library must be libz.so (normally located in /usr/lib).

Note that on Linux® distributions, you typically have to install the zlib (or zlib1g) package and create a symbolic link for libz.so. The libz.so file is part of zlib-devel package, though.

**Related concepts**
- The FGLPROFILE file(s) on page 255
- FGLPROFILE environment variable defines Genero BDL configuration files

**Front-end protocol errors**

When the front-end receives an invalid instruction from the runtime system, it stops the application connection. The runtime system then stops and displays error -6313 with an additional message, for example:

```
Program stopped at 'myprog.4gl', line number 675.
FORMS statement error number -6313.
The User Interface has been destroyed: Unexpected interface version sent by the runtime system.
```

When the runtime system receives an invalid AUI event from a front-end, it will raise a C assertion and produce a core file on UNIX® systems.
Related concepts
Exceptions on page 502
Describes exception (error) handling in the programs.

Debugging the front-end protocol

When setting the FGLGUIDEBUG environment variable to 1, information about GUI communication will be written to stderr by the runtime system, and the GUI protocol exchange will be indented for a better readability in the front-end log window.

Important: Sensitive and personal data may be written to the output. Make sure that the log output is written to files that can only be read by application administrators.

UNIX™ (shell) example:

```
$ FGLGUIDEBUG=1
$ export FGLGUIDEBUG
$ fglrun myprog 2>guidbg.txt
```

Note: The output format of FGLGUIDEBUG is for debug purpose only and can change in next product releases.

Note that in TUI mode, displayed screens can be dumped by setting the DBSCREEN_DUMP or DBSCREEN_OUT environment variables. This can be used to take a snapshot of the current TUI screen, for debugging or testing purpose.

Related concepts
FGLGUIDEBUG on page 275
Defines the debug level in GUI mode.

Front-end protocol logging

GUI protocol exchanges can be logged to a file with the --start-guilogo=filename option of fglrun, and replayed with the --run-guilogo=filename option.

Important: Sensitive and personal data may be written to the output. Make sure that the log output is written to files that can only be read by application administrators.

The --start-guilogo/--run-guilogo options are used to simulate a program execution by connecting with a front-end, without the program files. Basically, the runtime system connects with the front-end and replays the abstract user interface exchanges.

The --start-guilogo=filename instructs fglrun to start the program and log into filename, all user interface exchanges with the front-end. This option is typically used on site, with the program files.

With the --run-guilogo=filename option, the runtime system replays abstract user interface exchanges, as when the program was executed with the --start-guilogo option. This step can be done without the program files.

This feature can be used to set up non-regression tests for front-ends, and to provide a log file to your support center in order to replay a specific application issue.

The options take the log file as parameter:

UNIX™ (shell) example:

```
$ fglrun --start-guilogo=mylog.txt myprogram
```

All user interaction and AUI tree updates will be logged into the mylog.txt file.

The log file can then be replayed with the --run-guilogo option, to mimic the user interaction and program, to reproduce potential issues in front-ends:

```
$ fglrun --run-guilogo=mylog.txt myprogram
```
Note: If the parent program starts other child programs with `RUN cmd [WITHOUT WAITING]`, the parent program and each child program will write into the same log file. When replaying the GUI log file, the runtime system is able to identify parent and child program logs in order to restart individual processes.

When replaying the GUI log file, the runtime system prompts you for each user interaction. Type `ENTER` to replay each event step by step, or type `c` to continue by playing the full log:

```
$ fglrun --run-guilog=mylog.txt myprogram
enter: one step; c: continue
*** new process
<< 32351 3 i:meta Client{{name "GDC"}{version "3.00.06-152753"}{host ...  
>> 32351 56 o:om 0 {{an 0 UserInterface 0 {{name "guilog"}{text "guilog"}}}}
...
$
<< 32351 4181 i:event _om 0{}{{ActionEvent 0{{idRef "96"}}}})
*** new process
<< 32355 4186 i:meta Client{{name "GDC"}{version "3.00.06-152753"}{ho ...  
>> 32355 4209 o:om 0 {{an 0 UserInterface 0 {{name "guilog"}{text "guilog"}}}}
$c
<< 32355 5428 i:event _om 0{}{{ActionEvent 0{{idRef "96"}}}})
>> 32355 5429 o:om 1 {{rn 0}}
*** process terminated
*** process changed
<< 32357 6278 i:event _om 1{}{{ActionEvent 0{{idRef "97"}}}})
>> 32357 6278 o:om 2 {{un 95 {{selection "97"}}} {un 0 {{runtimeStatus "chil ...  
>> 32357 6278 o:om 3 {{un 0 {{runtimeStatus "processing"}}}}}
>> 32357 6283 o:om 4 {{un 0 {{focus "97"} {runtimeStatus "interactive"}}}})
*** new process
<< 32360 6296 i:meta Client{{name "GDC"}{version "3.00.06-152753-testEN"} 
{ho ...  
>> 32360 6344 o:om 0 {{an 0 UserInterface 0 {{name "guilog"}{text "guilog"}}}}
<< 32360 7341 i:event _om 0{}{{ActionEvent 0{{idRef "96"}}}})
>> 32360 7342 o:om 1 {{rn 0}}
*** process terminated
*** process changed
>> 32360 8099 o:om 5 {{rn 0}}
*** process terminated
bye!
```

Related concepts

fglrun on page 2065

The fglrun tool is the runtime system program that executes p-code programs.

**Automatic front-end startup**

This section describes how to start a graphical front-end automatically when the runtime system and the front-end reside on the same computer.

When a program starts in graphical mode, the runtime system tries to open a connection to the graphical front-end set by the FGLSERVER on page 280 environment variable. This requires having the front-end already started and listening to the TCP port specified by FGLSERVER.

In some configurations, such as X11 workstations or METAFRAME/Citrix Winframe or Microsoft" Windows® Terminal Server, each user may want to start his own front-end to have a dedicated process. This can be done by starting the front-end automatically when the program executes, based on settings in the DISPLAY (X11) or SESSIONNAME/CLIENTNAME (WTSE) environment variables.

Automatic front-end startup settings are defined with gui.server.autostart.* entries in FGLPROFILE. In these FGLPROFILE entries, the term "GUI server" refers to the graphical front end.
In a first time, the runtime system tries to establish the connection without starting the front-end (in a normal usage, it is already started). The front-end is identified by the FGLSERVER environment variable. If FGLSERVER is not defined, it defaults to localhost:0. except if gui.server.autostart.wsmmap entries are defined in FGLPROFILE. When wsmmap entries are defined, workstation id to GUI server id mapping takes place and FGLSERVER defaults to localhost:n, where n is the GUI server number found from wsmmap entries.

If this first connection fails and the gui.server.autostart.cmd entry is defined, the runtime system executes the command to start the GUI server, then waits for n seconds as defined by gui.server.autostart.wait entry, and after this delay tries to connect to the front-end. If the connection fails, it tries again for a number of attempts defined by the gui.server.autostart.repeat entry. Finally, if the last attempt fails, the runtime system stops with a GUI connection error -6300.

If the gui.server.autostart.cmd entry is not defined, neither workstation id to GUI id mapping, nor automatic front-end startup is done.

Here is a detailed description of each gui.server.autostart FGLPROFILE entry:

The cmd entry is used to define the command to be executed to start the front-end:

```bash
gui.server.autostart.cmd = "/opt/app/gdc-2.30/bin/gdc -p %d -q -M"
```

Here, %d will be replaced by the TCP port the front-end must listen to.

By default the runtime system waits for two seconds before it tries to connect to the front-end. You can change this delay with the wait entry:

```bash
gui.server.autostart.wait = 5  -- wait five seconds
```

The runtime system tries to connect to the front-end ten times. You can change this with the repeat entry:

```bash
gui.server.autostart.repeat = 3  -- repeat three times
```

The following FGLPROFILE entries can be used to define workstation id to front-end id mapping:

```bash
gui.server.autostart.wsmmap.max = 3
gui.server.autostart.wsmmap.0.names = "fox:1.0,fox.sxb.4js.com:1.0"
gui.server.autostart.wsmmap.1.names = "wolf:1.0,wolf.sxb.4js.com:1.0"
gui.server.autostart.wsmmap.2.names = "wolf:2.0,wolf.sxb.4js.com:2.0"
```

The first wsmmap.max entry defines the maximum number of front-end identifiers to look for. The wsmmap.N.names entries define a mapping for each GUI server, where N is the front-end identifier. The value of those entries defines a comma-separated list of workstation names to match. If no wsmmap entries are defined, the GUI server number will default to zero.

For gui.server.autostart.wsmmap entries, the first GUI server number starts at zero.

On X11 configurations, a workstation is identified by the DISPLAY environment variable. In this example, fox:1.0 identifies a workstation that will make the runtime start a front-end with the number 1.

On Windows® Terminal Server, the CLIENTNAME environment variable identifies the workstation. If no corresponding front-end id can be found in the wsmmap entries, the front-end number defaults to the id of the session defined by the SESSIONNAME environment variable, plus one. The value of this variable has the form protocol#id; for example, RDP-Tcp#4 would automatically define a front-end id of 5 (4+1).

If the front-end processes are started on the same machine as the runtime system, you do not need to set the FGLSERVER environment variable. This will then default to localhost:id, where id will be detected from the wsmmap workstation mapping entries.

If the front-end is executed on a middle-tier machine that is different from the application server, MIDHOST for example, you can set FGLSERVER to MIDHOST without a GUI server id. The workstation mapping will automatically find the id from the wsmmap settings.
**Note:** The Genero Desktop Client (GDC), raise the control panel to the top of the window stack when you try to restart it. In this case the program window might be hidden by the GDC control panel. To avoid this problem, you can use the `-M` option to start the GDC in minimized mode.

**Using a text terminal**

This section covers topics about text terminal configuration when using the TUI mode.

The TUI mode is enabled when setting the `FGLGUI` environment variable to zero.

Terminal type and terminal capabilities definition is not a Genero-specific configuration: `TERM`, `TERMCAP` and `TERMINFO` are also used by other UNIX™ applications and commands.

On UNIX™ platforms, the `TERM` environment variable must be set to define the terminal type/name. For example, if you execute the application in an `xterm` X11 window, set `TERM=xterm`.

On Windows® platforms, you can run applications in text mode inside a CMD console window. You must not set the `TERM` environment variable in this case.

Genero supports both `termcap` and `terminfo` implementations of text terminal capabilities. The `INFORMIXTERM` environment variable defines the type of library used to interact with the terminal. When `INFORMIXTERM` is set to `termcap` (the default), the runtime system reads terminal capabilities from the file defined by the `TERMCAP` environment variable. When `INFORMIXTERM` is set to `terminfo`, the runtime system uses the ncurses library of the operating system to interact with the terminal. We strongly recommend you to use the terminfo solution.

**Related concepts**

- **Text mode rendering (TUI mode)** on page 1109

**TERMINFO terminal capabilities**

When the `INFORMIXTERM` environment variable is set to `terminfo`, the runtime system will use the ncurses or curses library of the UNIX™ system to display and interact with the terminal device defined by the `TERM` environment variable.

Make sure that the Curses library is installed on your UNIX™ operating system. Check operating system installation requirements for more details.

The `TERMINFO` environment variable can be used to define a different terminal capabilities database as the default. If your UNIX™ system is properly configured, there is no need to set the `TERMINFO` environment variable.

**Related concepts**

- **INFORMIXTERM** on page 284
  Defines terminal control library to be used.

- **TERM** on page 264
  Defines the type of terminal on UNIX™ platforms.

**TERMCAP terminal capabilities**

When the `INFORMIXTERM` environment variable is set to `termcap` or when this variable is undefined, the runtime system will use the `termcap` terminal capabilities database.

The `termcap` solution is provided for backward compatibility. You should use `terminfo` instead, by setting the `INFORMIXTERM` variable to `terminfo`.

The default `termcap` database is in the `/etc/termcap` file. If this file is not found, the runtime system will use its default file `$FGLDIR/etc/termcap`. Use the `TERMCPAP` environment variable to specify a different `termcap` file as the defaults. If you plan to modify the default `termcap` file, we strongly recommend that you make a copy of the original file and point to the new file with the `TERMCAP` variable.

In this section we will briefly describe the syntax of the `termcap` file. For a complete definition please refer to your operating system documentation (see man pages describing the `termcap` file syntax).

**Related concepts**

- **INFORMIXTERM** on page 284
Defines terminal control library to be used.

**TERM** on page 264
Defines the type of terminal on UNIX™ platforms.

**TERMCP** on page 265
Defines the *termcap* terminal capabilities database on UNIX™ platforms.

**Termcap syntax**

All termcap entries contain a list of terminal names, followed by a list of terminal capabilities, in the following format:

- Each capability, including the last one in the entry, is followed by a colon ( : ).
- ESCAPE is specified as a backslash ( \ ) followed by the letter E. CTRL is specified as a caret (^). Do not use the ESCAPE or CTRL keys to indicate escape sequences or control characters in a termcap entry.
- Entries must be defined on a single logical line; a backslash ( \ ) appears at the end of each line that wraps to the next line.
- Comment lines begin with a hash sign ( # ).

Example: xterm terminal definition:

```
xterm|xterm terminal emulator:\
:km:mi:ms:xn:pt:\n:co#80:li#24:\n:is=E[r]E[m]E[2J]E[H]E[?7h]E[?1;3;4;61:\n...```

**Terminal Names**

Termcap entries begin with one or more names for the terminal, each separated by a vertical ( | ) bar. Any one of these names can be used for access to the termcap entry.

**Boolean capabilities**

Boolean capabilities are two-character codes indicating whether a terminal has a specific feature. If the boolean capability exists in the termcap entry, the terminal has that particular feature.

For example:

```
:bs:am:
 # bs backspace with CTRL-H
 # am automatic margins
```

**Numeric Capabilities**

Numeric capabilities are two-character codes followed by a hash symbol ( # ) and a value.

For example:

```
:co#80:li#24:
 # co number of columns in a line
 # li number of lines on the screen
```

The runtime system assumes that the value is zero for any numeric capabilities that are not listed.

**String Capabilities**

String capabilities specify a sequence that can be used to perform a terminal operation.
A string capability is a two-character code, followed by an equal sign (=) and a string ending at the next delimiter (:).

Most termcap entries include string capabilities for clearing the screen, arrow keys, cursor movement, underscore, function keys, etc.

For example, this shows some string capabilities for a Wyse 50 terminal:

```
:ce=\Et:cl=\E*:\n:nd=^L:up=^K:\n:so=\EG4:se=\EG0:
# ce=\Et clear to end of line
# cl=\E* clear the screen
# nd=^L non-destructive cursor right
# up=^K up one line
# so=\EG4 start stand-out
# se=\EG0 end stand-out
```

Genero-specific termcap definitions

Controlling the delay for ESC key

When using termcap, the ESCDELAY environment variable can be set to mimic the feature of the terminfo implementation: ESCDELAY specifies the delay (in milliseconds) before returning the ESC as a single key. Otherwise, ESC is the leading key of a control character sequence. The granularity of the delay value is 100 milliseconds. If ESCDELAY is not set (or set to 0), a default value of 500 milliseconds is used.

```
$ export ESCDELAY=600
```

Extending Function Key Definitions

In TUI mode, the runtime system recognizes function keys F1 through F36. These keys correspond to the termcap capabilities k0 through k9, followed by kA through kZ.

The termcap entry for these capabilities is the sequence of ASCII characters your terminal sends when you press the function keys (or any other keys you choose to use as function keys).

This example shows some function key definitions for the xterm terminal:

```
k0=\E[11~:k1=\E[12~:k2=\E[13~:k3=\E[14~:
...k9=\E[21~:kA=\E[23~:kB=\E[24~:
```

Defining dialog action keys

Dialog action keys for insert, delete and list navigation can be defined with the following capabilities:

- **ki**: Insert line (default is CTRL-J)
- **kj**: Delete line (default is CTRL-K)
- **kf**: Next page (default is CTRL-M)
- **kg**: Previous page (default is CTRL-N)

**Note:** You can also use the OPTIONS statement to name other function keys or CTRL keys for these operations.

Specifying Characters for Window Borders

The runtime system uses the graphics characters in the termcap file when you specify a window border in an OPEN WINDOW statement.
The runtime system uses characters defined in the `termcap` file to draw the border of a window. If no characters are defined in this file, the runtime system uses the hyphen ( `- `) for horizontal lines, the vertical bar ( `|` ) for vertical lines, and the plus sign ( `+` ) for corners.

Steps to define the graphical characters for window borders for your terminal type:

1. Determine the escape sequences for turning the terminal graphics mode ON and OFF (Refer to the manual of your terminal). For example, on Wyse 50 terminals, the escape sequence for entering graphics mode is ESC H^B, and the escape sequence for leaving graphics mode is ESC H^C.

2. Identify the ASCII equivalents for the six graphics characters that Genero requires to draw the window borders. The ASCII equivalent of a graphics character is the key you would press in graphics mode to obtain the indicated character. The six graphical characters needed by Genero are:
   - a. The upper left corner
   - b. The lower left corner
   - c. The upper right corner
   - d. The lower right corner
   - e. The horizontal line
   - f. The vertical line

3. Edit the termcap entry for your terminal, and define the following string capabilities:
   - `gs`: The escape sequence for entering graphics mode. In the `termcap` file, ESCAPE is represented as a backslash ( `\` ) followed by the letter E; CTRL is represented as a caret ( `^` ). For example, the Wyse 50 escape sequence ESC-H CTRL-B is represented as `\EH^B`.
   - `ge`: The escape sequence for leaving graphics mode. For example, the Wyse 50 escape sequence ESC-H CTRL-C is represented as `\EH^C`.
   - `gb`: The concatenated, ordered list of ASCII equivalents for the six graphics characters used to draw the border. Using the order as listed in (2).
     For example, if you are using a Wyse 50 terminal, you would add the following, in a linear sequence:
     ```
     :gs=\EH^B:ge=\EH^C:gb=2135z6:\n
     For terminals without graphics capabilities, you must enter a blank value for the `gs` and `ge` capabilities. For `gb`, enter the characters you want Genero to use for the window border. The following example shows possible values for`gs`, `ge`, and `gb` in an entry for a terminal without graphics capabilities:

     ```
     :gs=:ge=:gb=.|.|_|:
     ```

     With these settings, window borders would be drawn using underscores ( `_ `) for horizontal lines, vertical bars ( `| `) for vertical lines, periods ( `.` ) for the top corners, and vertical bars ( `| `) for the lower corners.

**Adding Color and Intensity**

In TUI mode, a Genero program can be written either for a monochrome or a color terminal, and then you can run the program on either type of terminal. If you set up the `termcap` files as described, the color attributes and the intensity attributes are related.

**Table 301: Relationship between color attributes and intensity attributes**

<table>
<thead>
<tr>
<th>Number</th>
<th>Color</th>
<th>Intensity</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>WHITE</td>
<td>NORMAL</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>YELLOW</td>
<td>BOLD</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>MAGENTA</td>
<td>BOLD</td>
<td></td>
</tr>
<tr>
<td>Number</td>
<td>Color</td>
<td>Intensity</td>
<td>Note</td>
</tr>
<tr>
<td>--------</td>
<td>-------</td>
<td>-----------</td>
<td>------</td>
</tr>
<tr>
<td>3</td>
<td>RED</td>
<td>BOLD (*)</td>
<td>If the keyword BOLD is indicated as the attribute, the field will be RED on a color terminal</td>
</tr>
<tr>
<td>4</td>
<td>CYAN</td>
<td>DIM</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>GREEN</td>
<td>DIM</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>BLUE</td>
<td>DIM (*)</td>
<td>If the keyword DIM is indicated as the attribute, the field will be BLUE on a color terminal</td>
</tr>
<tr>
<td>7</td>
<td>BLACK</td>
<td>INVISIBLE</td>
<td></td>
</tr>
</tbody>
</table>

The background for colors is BLACK in all cases. In either color or monochrome mode, you can add the REVERSE, BLINK, or UNDERLINE attributes if your terminal supports them.

**The ZA String Capability**

Genero uses a parameterized string capability named ZA in the `termcap` file to determine color assignments. Unlike other `termcap` string capabilities that you set to a literal sequence of ASCII characters, ZA is a function string that depends on the following four parameters:

**Table 302: ZA function parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>p1</td>
<td>Color number between 0 and 7 (see Table 301: Relationship between color attributes and intensity attributes on page 1126).</td>
</tr>
<tr>
<td>2</td>
<td>p2</td>
<td>0 = Normal; 1 = Reverse.</td>
</tr>
<tr>
<td>3</td>
<td>p3</td>
<td>0 = No-Blink; 1 = Blink.</td>
</tr>
<tr>
<td>4</td>
<td>p3</td>
<td>0 = No-underscore; 1 = Underscore.</td>
</tr>
</tbody>
</table>

ZA uses the values of these four parameters and a stack machine to determine which characters to send to the terminal. The ZA function is called, and these parameters are evaluated, when a color or intensity attribute is encountered in a Genero program. Use the information in your terminal manual to set the ZA parameters to the correct values for your terminal.

The ZA string uses stack operations to push values onto the stack or to pop values off the stack. Typically, the instructions in the ZA string push a parameter onto the stack, compare it to one or more constants, and then send an appropriate sequence of characters to the terminal. More complex operations are often necessary; by storing the display attributes in static stack machine registers (named a through z), you can have terminal-specific optimizations.

The different stack operators that you can use to write the descriptions are summarized here. For a complete discussion of stack operators, see your operating system documentation.

**Operators that Send Characters to the Terminal**

- `%d` pops a numeric value from the stack and sends a maximum of three digits to the terminal. For example, if the value 145 is at the top of the stack, `%d` pops the value off the stack and sends the ASCII representations of 1, 4, and 5 to the terminal. If the value 2005 is at the top of the stack, `%d` pops the value off the stack and sends the ASCII representation of 5 to the terminal.
- `%2d` pops a numeric value from the stack and sends a maximum of two digits to the terminal, padding to two places. For example, if the value 145 is at the top of the stack, `%2d` pops the value off the stack and sends the ASCII representations of 4 and 5 to the terminal. If the value 5 is at the top of the stack, `%2d` pops the value off the stack and sends the ASCII representations of 0 and 5 to the terminal.

- `%3d` pops a numeric value from the stack and sends a maximum of three digits to the terminal, padding to three places. For example, if the value 7 is at the top of the stack, `%3d` pops the value off the stack and sends the ASCII representations of 0, 0, and 7 to the terminal.

- `%c` pops a single character from the stack and sends it to the terminal.

### Operators that Manipulate the Stack

- `%p[1-9]` pushes the value of the specified parameter on the stack. The notation for parameters is `p1`, `p2`, ..., `p9`. For example, if the value of `p1` is 3, `%p1` pushes 3 on the stack.

- `%P[a-z]` pops a value from the stack and stores it in the specified variable. The notation for variables is `Pa`, `Pb`, ..., `Pz`. For example, if the value 45 is on the top of the stack, `%Pb` pops 45 from the stack and stores it in the variable `Pb`.

- `%g[a-z]` gets the value stored in the corresponding variable (`P[a-z]`) and pushes it on the stack. For example, if the value 45 is stored in the variable `Pb`, `%gb` gets 45 from `Pb` and pushes it on the stack.

- `%"c` pushes a single character on the stack. For example, `%"k` pushes k on the stack.

- `%{n}` pushes an integer constant on the stack. The integer can be any length and can be either positive or negative. For example, `%{0}` pushes the value 0 on the stack.

- `%S[a-z]` pops a value from the stack and stores it in the specified static variable. (Static storage is nonvolatile since the stored value remains from one attribute evaluation to the next.) The notation for static variables is `Sa`, `Sb`, ..., `Sz`. For example, if the value 45 is on the top of the stack, `%Sb` pops 45 from the stack and stores it in the static variable `Sb`. This value is accessible for the duration of the Genero program.

- `%G[a-z]` gets the value stored in the corresponding static variable (`S[a-z]`) and pushes it on the stack. For example, if the value 45 is stored in the variable `Sb`, `%Gb` gets 45 from `Sb` and pushes it on the stack.

### Arithmetic Operators

Each arithmetic operator pops the top two values from the stack, performs an operation, and pushes the result on the stack.

- `%+` Addition.
  
  For example, `%{2}%{3}%+` is equivalent to 2+3.

- `%-` Subtraction.
  
  For example, `%{7}%{3}%-` is equivalent to 7-3.

- `%*` Multiplication.
  
  For example, `%{6}%{3}%*` is equivalent to 6*3.

- `%/` Integer division.
  
  For example, `%{7}%{3}%/` is equivalent to 7/3 and produces a result of 2.

- `%m` Modulus (or remainder).
  
  For example, `%{7}%{3}%m` is equivalent to (7 mod 3) and produces a result of 1.

### Bit Operators

The following bit operators pop the top two values from the stack, perform an operation, and push the result on the stack:

- `%&` Bit-and.
  
  For example, `%{12}%{21}%&` is equivalent to (12 and 21) and produces a result of 4.

- `%|` Bit-or.
  
  For example, `%{12}%{21}%|` is equivalent to (12 or 21) and produces a result of 29.
• %\^ Exclusive-or.

For example, %\{12\} %\{21\} %\^ is equivalent to (12 exclusive-or 21) and produces a result of 25.

The following unary operator pops the top value from the stack, performs an operation, and pushes the result on the stack:

• %\~ Bitwise complement.

For example, %\{25\} %\~ results in a value of -26.

**Logical Operators**

The following logical operators pop the top two values from the stack, perform an operation, and push the logical result (0 for false or 1 for true) on the stack:

• %= Equal to.

For example, if the parameter p1 has the value 3, the expression %p1%{2}%= is equivalent to 3=2 and produces a result of 0 (false).

• %> Greater than.

For example, if the parameter p1 has the value 3, the expression %p1%{0}%> is equivalent to 3>0 and produces a result of 1 (true).

• %< Less than.

For example, if the parameter p1 has the value 3, the expression %p1%{4}%< is equivalent to 3<4 and produces a result of 1 (true).

The following unary operator pops the top value from the stack, performs an operation, and pushes the logical result (0 or 1) on the stack.

• %! Logical negation.

This operator produces a value of zero for all nonzero numbers and a value of 1 for zero. For example, %\{2\} %! results in a value of 0, and %\{0\} %! results in a value of 1.

**Conditional Statements**

The conditional statement has the following format:

```
%? expr %t thenpart %e elsepart %;
```

The %e elsepart is optional. You can nest conditional statements in the thenpart or the elsepart.

When Genero evaluates a conditional statement, it pops the top value from the stack and evaluates it as either true or false. If the value is true, the runtime performs the operations after the %t; otherwise it performs the operations after the %e (if any).

For example, the expression:

```
%?%p1%{3}%=%t;31%;
```

is equivalent to:

```
if p1 = 3 then print ",;31"
```

Assuming that p1 in the example has the value 3, Genero would perform the following steps:

• %? does not perform an operation but is included to make the conditional statement easier to read.
• %p1 pushes the value of p1 on the stack.
• %{3} pushes the value 3 on the stack.
• `%=` pops the value of `p1` and the value 3 from the stack, evaluates the boolean expression `p1=3`, and pushes the resulting value 1 (true) on the stack.

• `%t` pops the value from the stack, evaluates 1 as true, and executes the operations after `%t`. (Since `;31` is not a stack machine operation, Genero prints `;31` to the terminal.)

• `%`; terminates the conditional statement.

**ZA example**

The ZA sequence for the ID Systems Corporation ID231 (color terminal) is:

```
ZA =
\E[0;                # Print lead-in
%??p1%{0}%=%t%{7}    # Encode color number (translate color number
# to number for the ID231 term)
%e%p1%{1}%=%t%{3}    #
%e%p1%{2}%=%t%{5}    #
%e%p1%{3}%=%t%{1}    #
%e%p1%{4}%=%t%{6}    #
%e%p1%{5}%=%t%{2}    #
%e%p1%{6}%=%t%{4}    #
%e%p1%{7}%=%t%{0};   #
%??p2%t30;%{40}%+%2d # if p2 is set, print '30' and '40' + color number
(reverse)
%e40;%{30}%+%2d%;   # else print '40' and '30' + color number (normal)
%??p3%t;5%;         # if p3 is set, print 5 (blink)
%??p4%t;4%;         # if p4 is set, print 4 (underline)
m                    # print 'm' to end character sequence
```

**Text mode screen dump**

For compatibility with IBM® Informix® 4GL, Genero supports the DBSCREENDUMP and DBSCREENOUT environment variables for debugging purpose, which allows you to take a screenshot when running in TUI mode and save the result in a file.

To enable TUI screenshot, set either DBSCREENDUMP or DBSCREENOUT to the name of the output file, then run your Genero program with FGLGUI=0 set and press the Ctrl-P key to dump the current screen. Each time you press Ctrl-P, the output file will be overwritten.

The DBSCREENDUMP variable writes the screen with escape sequences of TTY attributes, while DBSCREENOUT writes only the characters displayed on the screen, which makes the output more readable.

If both variables are set, the runtime will generate both output files; however, use different file names, otherwise the output is undefined.

**Related concepts**

**DBSCREENDUMP** on page 272
Defines the output file name for text screen shots.

**DBSCREENOUT** on page 273
Defines the output file name for text screen shots.

Form definitions

This section describes how to define application forms and program resources related to the presentation layer.

Windows and forms

The section describes the concept of windows and forms in the language.

OPEN WINDOW

Creates and displays a new window.

Syntax

```
OPEN WINDOW identifier
   ↓ AT line, column ↓
   WITH ↓ FORM form-file
      ↓ height ROWS, width COLUMNS
      ↓ ATTRIBUTES ( window-attributes ) ↓
```

where display-attribute is:

```
BLACK ↓ BLUE ↓ CYAN ↓ GREEN
MAGENTA ↓ RED ↓ WHITE ↓ YELLOW
BOLD ↓ DIM ↓ INVISIBLE ↓ NORMAL
REVERSE ↓ BLINK ↓ UNDERLINE
BORDER
TEXT = "string"
TYPE = ↓ RIGHT ↓ LEFT ↓ POPUP ↓ NAVIGATOR ↓
STYLE = "string"
PROMPT LINE = integer
MENU LINE = integer
MESSAGE LINE = integer
ERROR LINE = integer
COMMENT LINE = ↓ OFF ↓ integer ↓
```

1. identifier is the name of the window. It is always converted to lowercase by the compiler.
2. line is the integer defining the top position of the window. The first line in the screen is 1, while the relative line number inside the window is zero.
3. column is the integer defining the position of the left margin. The first column in the screen is 1, while the relative column number inside the window is zero.
4. form-file defines the name of the compiled form file, without .42f extension.
5. height defines the number of lines of the window in character units; includes the borders in character mode.
6. width defines the number of lines of the window in character units; includes the borders in character mode.

Usage

An OPEN WINDOW statement can have the following effects:

- Declares a name (the identifier) for the window.
- Indicates which form has to be used in that window.
- Specifies the display attributes of the window.
- When using character mode, specifies the position and dimensions of the window, in character units.

For graphical applications, use this instruction without the AT clause, and with the WITH FORM clause.
The window identifier must follow the rules for identifiers and be unique among all windows defined in the program. Its scope is the entire program. You can use this identifier to reference the same Window in other modules with other statements (for example, CURRENT WINDOW on page 1135 and CLOSE WINDOW on page 1135).

The compiler converts the window identifier to lowercase for internal storage. When using functions or methods receiving the window identifier as a string parameter, the window name is case sensitive. We recommend that you always specify the window identifier in lowercase letters.

The runtime system maintains a stack of all open windows. If you execute OPEN WINDOW to open a new window, it is added to the window stack and becomes the current window. Other statements that can modify the window stack are CURRENT WINDOW and CLOSE WINDOW.

**Example**

```plaintext
MAIN
  OPEN WINDOW w1 WITH FORM "customer"
  MENU "Test"
    COMMAND KEY(INTERRUPT) "exit" EXIT MENU
  END MENU
  CLOSE WINDOW w1
END MAIN
```

**Related concepts**

- **Graphical mode with Traditional Display** on page 1114
- **Form specification files** on page 1237
  Form specification files are the source files defining the layout and content of application forms.
- **The Window class** on page 2490
  The `ui.Window` class provides an interface to the window objects create with the OPEN WINDOW instruction.
- **The Form class** on page 2496
  The `ui.Form` class provides an interface to form objects created by an OPEN WINDOW WITH FORM or DISPLAY FORM instruction.

**OPEN WINDOW attributes**
List of attributes for the OPEN WINDOW instruction.

**Table 303: Window-attributes supported by the OPEN WINDOW statement**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TEXT = string</strong></td>
<td>Defines the default title of the window. When a form is displayed, the form title (LAYOUT(TEXT=&quot;mytitle&quot;)) will be used as window title. <strong>Tip:</strong> We recommend that you define the window title in the form file.</td>
</tr>
<tr>
<td><strong>STYLE = string</strong></td>
<td>Defines the default style of the window. If the form defines a window style, (LAYOUT(WINDOWSTYLE=&quot;mystyle&quot;)), it overwrites the default window style. <strong>Tip:</strong> We recommend that you define the window style in the form file.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>TYPE = [LEFT\RIGHT\POPUP\NAVIGATOR]</td>
<td>Defines the window type. According to the type specified, the window will appear differently, following front-end platform GUI standards. For example, on iOS devices, a window created with TYPE=POPUP will show up from the bottom of the screen. See Window TYPE attribute on page 1134.</td>
</tr>
<tr>
<td>BLACK, BLUE, CYAN, GREEN, MAGENTA, RED, WHITE, YELLOW</td>
<td>Default TTY color of the data displayed in the window.</td>
</tr>
<tr>
<td>BOLD, DIM, INVISIBLE, NORMAL</td>
<td>Default TTY font attribute of the data displayed in the window.</td>
</tr>
<tr>
<td>REVERSE, BLINK, UNDERLINE</td>
<td>Default TTY video attribute of the data displayed in the window.</td>
</tr>
<tr>
<td>PROMPT LINE integer</td>
<td>In character mode, indicates the position of the prompt line for this window. The position can be specified with FIRST and LAST predefined line positions.</td>
</tr>
<tr>
<td>FORM LINE integer</td>
<td>In character mode, indicates the position of the form line for this window. The position can be specified with FIRST and LAST predefined line positions.</td>
</tr>
<tr>
<td>MENU LINE integer</td>
<td>In character mode, indicates the position of the ring menu line for this window. The position can be specified with FIRST and LAST predefined line positions.</td>
</tr>
<tr>
<td>MESSAGE LINE integer</td>
<td>In character mode, indicates the position of the message line for this window. The position can be specified with FIRST and LAST predefined line positions.</td>
</tr>
<tr>
<td>ERROR LINE integer</td>
<td>In character mode, indicates the position of the error line for this window. The position can be specified with FIRST and LAST predefined line positions.</td>
</tr>
<tr>
<td>COMMENT LINE [OFF\integer]</td>
<td>In character mode, indicates the position of the comment line or no comment line at all, for this window. The position can be specified with FIRST and LAST predefined line positions.</td>
</tr>
<tr>
<td>BORDER</td>
<td>Indicates if the window must be created with a border in character mode. A border frame is drawn outside the specified window. This means, that the window needs 2 additional lines and columns on the screen.</td>
</tr>
</tbody>
</table>

The following list describes the default line positions in character mode:

- First line: Prompt line (output from PROMPT statement) and Menu line (command value from MENU statement).
- Second line: Message line (output from MESSAGE statement; also the descriptions of MENU options).
- Third line: Form line (output from DISPLAY FORM statement).
- Last line: Error line (output from ERROR statement). Also comment line in any window except SCREEN.
**WITH FORM clause**

Creating a window object with a form.

The `WITH FORM` clause can be used to specify the name of a compiled form file to be used to create a window. A window object is automatically opened and sized to the screen layout of the form.

```plaintext
OPEN WINDOW w1 WITH FORM "custlist"
```

**Note:** When using the TUI mode, the width of the window is from the left-most character on the screen form (including leading blank spaces) to the right-most character on the screen form (truncating trailing blank spaces). The length of the window is calculated as (form line) + (form length).

It is recommended that you use the `WITH FORM` clause, especially in the default GUI mode, because the window is created in accordance with the form. If you use this clause, you do not need the `OPEN FORM`, `DISPLAY FORM`, or `CLOSE FORM` statement to open and close the form. The `CLOSE WINDOW` on page 1135 statement closes the window and the form.

**Important:**

The form file name identifies the .42f compiled form file to be loaded. The file name may use a .42f extension, but this is not recommended.

The file name can be a simple file name, a relative file path, or an absolute file path.

- When using a simple file name or a relative path, form files are found relative to several directories in a given order, as described in the FGLRESOURCEPATH reference topic.
- When specifying an absolute path, FGLRESOURCEPATH (or DBPATH) is not used. On Windows®, an absolute file name must start with a drive letter (C:), a backslash (\) or a slash (/). If FGLRESOURCEPATH contains a driver letter (C:), a form file specified as "/foo/bar" will only be found in "C:/foo/bar", if C: is the current drive.

**Window TYPE attribute**

Use the `TYPE` attribute to define the type of a window.

**Important:** This feature is only for mobile platforms.

The type of a window can be specified with the `TYPE` attribute in the `OPEN WINDOW` instruction:

```plaintext
OPEN WINDOW w_main WITH FORM "navi"
   ATTRIBUTES( TYPE = NAVIGATOR )
```

This attribute was introduced to implement split-views on mobile front-ends.

Possible values for the `TYPE` attribute are described in the following table:

**Table 304: Supported window types**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEFT</td>
<td>Defines the window as the left pane when implementing split views.</td>
</tr>
<tr>
<td></td>
<td>The window will be the parent window of a window cascade displayed on the left-hand side.</td>
</tr>
<tr>
<td>NAVIGATOR</td>
<td>Defines the window as the action pane (i.e. iOS Tab bar) or drop down menu (Android) when implementing split views.</td>
</tr>
<tr>
<td></td>
<td>This type of window will be used as top-level navigator window, showing the options to switch between different windows controlled by parallel dialogs.</td>
</tr>
<tr>
<td>POPUP</td>
<td>Defines the window as pop-up (modal) window, to open on the top of other windows.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> The POPUP window type is ignored by GMA on Android™.</td>
</tr>
</tbody>
</table>
**RIGHT**
Defines the window as the right pane when implementing split views.
The window will be the parent window of a window cascade displayed on the right-hand side.

**Related concepts**
Parallel dialogs (START DIALOG) on page 1700  
The START DIALOG and TERMINATE DIALOG instructions provide for dialog functionality executing concurrently in different application forms.

**CLOSE WINDOW**
Closes and destroys a window.

**Syntax**

```plaintext
CLOSE WINDOW { identifier \ SCREEN }
```

1. `identifier` is the name of the window.

**Usage**
The `CLOSE WINDOW` closes the specified window. If the `OPEN WINDOW` statement includes the `WITH FORM` clause, it closes both the form and the window.

Closing a window has no effect on variables that were set while the window was open.

Closing the current window makes the next window on the stack the new current window. If you close any other window, the runtime system deletes it from the stack, leaving the current window unchanged.

If the window is currently being used for input, `CLOSE WINDOW` generates a runtime error.

You can close the default screen window with the `CLOSE WINDOW SCREEN` instruction.

**Example**

```plaintext
MAIN
  OPEN WINDOW w1 WITH FORM "customer"
  MENU "Test"
    COMMAND KEY(INTERRUPT) "exit" EXIT MENU
  END MENU
  CLOSE WINDOW w1
END MAIN
```

**Related concepts**
OPEN WINDOW on page 1131  
Creates and displays a new window.

**CURRENT WINDOW**
Makes a specified window the current window.

**Syntax**

```plaintext
CURRENT WINDOW IS { identifier \ SCREEN }
```

1. `identifier` is the name of the window.
Usage

Programs with multiple windows might need to switch to a different open window so that input and output occur in the appropriate window. To make a window the current window, use the CURRENT WINDOW statement.

When a program starts, the screen is the current window. Its name is SCREEN. To make this the current window, specify the keyword SCREEN instead of a window identifier.

If the window contains a form, that form becomes the current form when a CURRENT WINDOW statement specifies the name of that window. All interactive instruction such as CONSTRUCT, INPUT use only the current window for input and output. If the user displays another form (for example, through an ON KEY clause) in one of these statements, the window containing the new form becomes the current window. When an interactive instruction resumes, its original window becomes the current window.

The CURRENT WINDOW instruction is typically used in text user interface (TUI) based applications, when distinct areas of the screen are reserved for different usage. In a GUI application, windows are typically opened and closed sequentially or on a stack of windows.

Example

```
MAIN
  OPEN WINDOW w1 WITH FORM "customer"
  ...
  OPEN WINDOW w2 WITH FORM "custlist"
  ...
  CURRENT WINDOW IS w1
  ...
  CURRENT WINDOW IS w2
  ...
  CLOSE WINDOW w1
  CLOSE WINDOW w2
END MAIN
```

Related concepts

OPEN WINDOW on page 1131
Creates and displays a new window.

Text mode rendering (TUI mode) on page 1109

CLEAR WINDOW
Cleans the contents of a window.

Syntax

```
CLEAR WINDOW ↓ identifier ↓ SCREEN ↓
```

1. `identifier` is the name of the window.

Usage

The CLEAR WINDOW instruction cleans the content of the specified window that was declared in an OPEN WINDOW.

In TUI mode, if the window was created with borders, these are left untouched (only the content of the window is cleared).

If you specify CLEAR WINDOW SCREEN, the root screen will be cleared, except areas occupied by an existing window. CLEAR WINDOW SCREEN will not change the current window setting.

The CLEAR WINDOW instruction is typically used in text user interface (TUI) based applications, as it clears the whole content of the window, including static labels and messages.
Related concepts
Text mode rendering (TUI mode) on page 1109

OPEN FORM
Declares a compiled form in the program.

Syntax

\[
\text{OPEN FORM identifier FROM form-file}
\]

1. \text{identifier} is an identifier that defines the name of the form object.
2. \text{form-file} defines the name of the compiled form file, without .42f extension.

Usage

In order to use a .42f compiled version of a form specification file, the programs must declare the form with the OPEN FORM instruction and then display the form in the current window by using the DISPLAY FORM instruction.

\text{OPEN FORM/DISPLAY FORM} are typically used at the beginning of programs to display the main form in the default SCREEN window:

\begin{verbatim}
OPEN FORM custform FROM "customer"
DISPLAY FORM custform
\end{verbatim}

The form identifier does not need to match the name of the form specification files, but it must be unique among form names in the program. Its scope of reference is the entire program.

Important:

The form file name identifies the .42f compiled form file to be loaded. The file name may use a .42f extension, but this is not recommended.

The file name can be a simple file name, a relative file path, or an absolute file path.

- When using a simple file name or a relative path, form files are found relative to several directories in a given order, as described in the FGLRESOURCEPATH reference topic.
- When specifying an absolute path, FGLRESOURCEPATH (or DBPATH) is not used. On Windows®, an absolute file name must start with a drive letter (\text{C:}), a backslash (\text{	extbackslash}) or a slash (\text{/}). If FGLRESOURCEPATH contains a driver letter (\text{C:}), a form file specified as "\text{/foo/bar}" will only be found in "\text{C:/foo/bar}", if \text{C:} is the current drive.

If you execute an \text{OPEN FORM} with the name of an open form, the runtime system first closes the existing form before opening the new form.

The scope of reference of form identifier is the entire program.

When the window is dedicated to the form, use the \text{OPEN WINDOW WITH FORM} instruction to create the window and the form object in one statement.

In TUI mode, the form is displayed in the current window at the position defined by the \text{FORM LINE} attribute that can be specified in the \text{ATTRIBUTE} clause of \text{OPEN WINDOW} or as default with the \text{OPTIONS} instruction.

After the form is loaded, you can activate the form by executing a \text{CONSTRUCT}, \text{DISPLAY ARRAY}, \text{INPUT}, \text{INPUT ARRAY}, or \text{DIALOG} statement. When the runtime system executes the \text{OPEN FORM} instruction, it allocates resources and loads the form into memory. To release the allocated resources when the form is no longer needed, the program must execute the \text{CLOSE FORM} instruction. This is a memory-management feature to recover memory from forms that the program no longer displays on the screen. If the form was loaded with a window by using the \text{WITH FORM} clause, it is automatically closed when the window is closed with a \text{CLOSE WINDOW} instruction.
Example

```
MAIN
  OPEN FORM f1 FROM "customer"
  DISPLAY FORM f1
  CALL input_customer()
  CLOSE FORM f1
  OPEN FORM f2 FROM "custlist"
  DISPLAY FORM f2
  CALL input_custlist()
  CLOSE FORM f2
END MAIN
```

Related concepts

**Form specification files** on page 1237
Form specification files are the source files defining the layout and content of application forms.

**Defining the position of reserved lines** on page 559
The **OPTIONS** **element** **LINE** defines position of dedicated screen lines.

**DISPLAY FORM**
Displays and associates a form with the current window.

**Syntax**

```
DISPLAY FORM identifier
  \ ATTRIBUTES ( display-attributes ) \n```

1. `identifier` is the name of the form.
2. `window-attributes` defines the display attributes of the form.

where `display-attribute` is:

```
\ BLACK \ BLUE \ CYAN \ GREEN
\ MAGENTA \ RED \ WHITE \ YELLOW
\ BOLD \ DIM \ INVISIBLE \ NORMAL
\ REVERSE \ BLINK \ UNDERLINE
```

**Usage**
The **DISPLAY FORM** instruction creates a form element in the current window, from a form resource loaded by the **OPEN FORM** instruction.

**Important**: The **INVISIBLE** display attribute is ignored.

The runtime system applies display attributes that you specify in the **ATTRIBUTES** clause, to any fields that have not been assigned attributes by the **ATTRIBUTES** section of the form specification file, or by the database schema files, or by the **OPTIONS** runtime configuration statement. If the form is displayed in a window, color attributes from the **DISPLAY FORM** statement supersede any from the **OPEN WINDOW** on page 1131 statement. If however subsequent **CONSTRUCT**, **DISPLAY**, or **DISPLAY ARRAY** statements that include an **ATTRIBUTES** clause reference the form, their attributes take precedence over those specified in the **DISPLAY FORM** instruction.

In graphical mode, by default, the parent window adapts its size to the content of the form displayed with **DISPLAY FORM**. When successive **DISPLAY FORM** instructions use different forms (with different content), it is possible to control the parent window resizing with the **resetFormSize** style attribute. Note that this style attribute is to be used in the form element, not in the window element.

**Related concepts**

**CLOSE FORM** on page 1139
Closes the resources allocated by OPEN FORM.

CLOSE FORM
Closes the resources allocated by OPEN FORM.

Syntax

```
CLOSE FORM identifier
```

1. `identifier` is the name of the form.

Usage

The CLOSE FORM instruction releases the memory allocated to the form.

A form associated with a window by the OPEN WINDOW WITH FORM instruction is automatically closed when the program closes the window with a CLOSE WINDOW instruction.

Related concepts

CLOSE WINDOW on page 1135
Closes and destroys a window.

OPEN FORM on page 1137
Declares a compiled form in the program.

CLEAR SCREEN
Clears the complete application screen.

Syntax

```
CLEAR SCREEN
```

Usage

The CLEAR SCREEN instruction is typically used in TUI mode to clear the complete screen and make the root screen window the current window on the stack.

The whole screen will be cleared, including prompt, error and message lines (the menu line is not cleared).

Related concepts

Text mode rendering (TUI mode) on page 1109
CLEAR WINDOW on page 1136
Clears the contents of a window.

DISPLAY AT
Displays text at a given line/column position in the current window.

Syntax

```
DISPLAY text AT line, column [ ATTRIBUTES ( display-attributes ) ]
```

1. `text` is any expression to be evaluated and displayed at the given position in the current window.
2. `line` is an integer expression defining the line position in the current window.
3. `column` is an integer expression defining the column position on the screen.
4. `display-attributes` defines the display attributes for the `text`. 
Usage

The `DISPLAY AT` instruction evaluates a string expression and displays the result at a given line and column in the current window. This instruction is typically used in text user interface (TUI) text-based applications to display static text on the screen such as messages or decoration lines with – (dash) or _ (underscore) characters.

Use of the `DISPLAY AT` instruction is recommended only in TUI mode or GUI/traditional mode. To display data at a given place in a graphical form, use form fields and the `DISPLAY BY NAME` or `DISPLAY TO` instructions, or use interactive instructions with the `UNBUFFERED` mode to automatically display program variable data to form fields.

When using `DISPLAY AT` in GUI mode, the text will only be displayed if the current window contains no form, or contains a form defined with the `SCREEN` layout.

Table 305: Display-attributes supported by the `DISPLAY AT` statement

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLACK, BLUE, CYAN, GREEN, MAGENTA, RED, WHITE, YELLOW</td>
<td>The TTY color of the displayed text.</td>
</tr>
<tr>
<td>BOLD, DIM, INVISIBLE, NORMAL</td>
<td>The TTY font attribute of the displayed text.</td>
</tr>
<tr>
<td>REVERSE, BLINK, UNDERLINE</td>
<td>The TTY video attribute of the displayed text.</td>
</tr>
</tbody>
</table>

Using windows and forms

This section include details about Genero windows and forms usage.

Understanding windows and forms

This is an introduction to Genero windows and forms.

Programs manipulate windows and forms, to define display and input areas controlled by interactive instructions such as the `INPUT` dialog.

A default window is always created with the name `SCREEN` where the main form of the program can be displayed with `OPEN FORM/DISPLAY FORM`. Secondary forms are typically displayed in secondary windows with `OPEN WINDOW + WITH FORM`.

When a dialog is started, it uses the form associated with the current window.

Forms define the layout and presentation of areas used by the dialogs (`INPUT`), to display or input data.

Forms are defined in `.42f` compiled form files and are loaded at runtime to be displayed in windows.

Related concepts

What are dialog controllers? on page 1707

Application forms are controlled by interactive instruction blocks called dialogs. These blocks perform the common tasks associated with the form, such as field input and action handling.

The window concept

Windows are containers for `.42f` forms.

Creating windows

The windows are created from programs; they define a display context for sub-elements like forms, menus, message and error lines.

A window can contain only one form at a time, but you can display different forms successively in the same window.

A program creates a new window with the `OPEN WINDOW` instruction, which also defines the window identifier:

```sql
OPEN WINDOW mywindow WITH FORM "myform"
```
Destroying windows

A window is destroyed with the CLOSE WINDOW instruction:

CLOSE WINDOW mywindow

Windows rendering context

When using the text mode (FGLGUI=0), windows are displayed in the character terminal as fixed-size boxes, at a given line/column position, width and height.

When using a graphical desktop front-end (FGLGUI=1), windows are displayed by default as independent resizable native windows with the GDC front-end, or displayed as frames inside a web browser when using the GBC front-end.

Note: A GUI application can run in traditional mode (gui.uiMode="traditional" FGLPROFILE setting), displaying windows as simple static areas inside a real graphical parent window.

When using a mobile device front-end in Native Rendering mode, only one window is visible at the time, because of device platform GUI standards and the limited screen sizes (smartphones). Split views are the exception, as they allow for the display of two windows side by side in a typical list-detail display on tablets.

Default SCREEN window

When a program starts, the runtime system creates a default window named SCREEN.

This default window can be used as a regular window: it can hold a menu and a form.

You typically display the main form of your program in the SCREEN window, by using OPEN FORM + DISPLAY FORM:

```
MAIN
    -- The SCREEN window exists by default
    ...
    OPEN FORM f_main FROM "customers"
    DISPLAY FORM f_main -- displays in SCREEN
    ...
END MAIN
```

Tip: If needed, the default SCREEN window can be closed with CLOSE WINDOW SCREEN. However, in most cases, you want to keep this default window and display the main form of the program with OPEN FORM + DISPLAY FORM.

The current window

A program with user interface must always have a current window.

Several windows can be created, but there can be only one current window when using modal dialogs (only one dialog is active at the time, thus only the current window can be active).

Note: When using parallel dialogs, several windows can be active concurrently. Parallel dialogs were introduced to implement split views, for mobile devices.

There is always a current window. The last created window becomes the current window. When the last created window is closed, the previous window in the window stack becomes the current window.

Use the CURRENT WINDOW instruction to make a specific window current, before executing the corresponding dialog that is controlling the window content:

```
OPEN WINDOW w_customers ...
OPEN WINDOW w_orders ...
...
CURRENT WINDOW IS w_customers ...
```
CLOSE WINDOW w_customers
CURRENT WINDOW IS w_orders
...

However, this practice is not commonly used: A regular Genero program starts with a main window/form and opens new windows in cascade that results in a tree of windows. The last created window is closed to go back to the previous window/form.

**Displaying multiple forms in the same window**

When there is a current window, it is possible to display several forms successively in that same window.

The previous form is removed automatically by the runtime system when displaying a new form to the window:

```
OPEN WINDOW mywindow WITH FORM "form1"
INPUT BY NAME ... -- uses form1 elements
...
OPEN FORM f1 FROM "form2"
DISPLAY FORM f1  -- removes "form1" from the window
INPUT BY NAME ... -- uses form2 elements
...
```

**Windows types (TYPE attribute)**

By default, a window has no particular type and displays as a regular GUI window on the front-end, to be controlled by a modal dialog instruction. In some situations, you must specify the TYPE of the window, to get a specific rendering and behavior (for example when implementing split-views on mobile front-ends.

**Note:** Do not confuse the window TYPE attribute with the windowType style attribute, which can be used by specifying a STYLE attribute for the WINDOW.

The window type is defined with the TYPE attribute in the ATTRIBUTES clause of the OPEN WINDOW instruction:

```
OPEN WINDOW w_cust WITH FORM "f_cust" ATTRIBUTES(TYPE=LEFT)
...
OPEN WINDOW w_pref WITH FORM "f_pref" ATTRIBUTES(TYPE=POPUP)
...
```

For more details, see Window TYPE attribute on page 1134.

**Window presentation styles**

Window decoration and behavior options can be defined with a presentation style.

The window style is identified with the STYLE attribute of the ATTRIBUTES section of OPEN WINDOW, or it can also be specified at form level, with the WINDOWSTYLE form attribute in the LAYOUT of the form definition:

```
OPEN WINDOW w_cust WITH FORM "f_cust" ATTRIBUTES(STYLE="dialog2")
```

**API for window objects**

The `ui.Window` built-in class can be used to manipulate windows as objects.

The common practice is to get the current form of the window and use it as `ui.Form` object to manipulate its content.
The form concept
Forms define the layout and presentation of areas used by the dialogs, to display or input data.

Loading forms in programs
Forms are loaded by programs from external files with the .42f extension, the compiled version of .per form specification files.

Forms are loaded with the OPEN FORM instruction followed by a DISPLAY FORM, to display the form into the current window, or forms can be used directly as window creation argument with the OPEN WINDOW ... WITH FORM instruction:

```plaintext
OPEN FORM f_cust FROM "f_cust"
DISPLAY FORM f_cust -- into current window
...
OPEN WINDOW w_cust WITH FORM "f_cust"
```

The version of a form
Forms can be stamped with the VERSION attribute: This attribute is used to indicate that the form content has changed.

The front-end is then able to distinguish different form versions and avoid saved settings being applied for new form versions.

```plaintext
LAYOUT (VERSION="1.45")
...
```

Controlling forms with dialogs
The form that is used by interactive instructions like INPUT is defined by the current window containing the form.

Switching between existing windows (and thus, between forms associated with each window) is done with the CURRENT WINDOW instruction.

Several forms can be successively displayed in the same (current) window. The last displayed form will be used by the next dialog, while the form displayed before will be automatically removed from the window:

```plaintext
OPEN WINDOW w_common WITH 20 ROWS, 60 COLUMNS
...
OPEN FORM f1 FROM "f_cust"
DISPLAY FORM f1 -- f_cust is shown
INPUT BY NAME rec_cust.* ...
...
OPEN FORM f2 FROM "f_ord"
DISPLAY FORM f2 -- f_ord is shown (f_cust is removed)
INPUT BY NAME rec_ord.* ...
```

API for form objects
The ui.Form built-in class is provided to handle form elements.

You can, for example, hide some parts of a form with the setElementHidden() method.

A ui.Form object can be obtained with the ui.Window.getForm() method, or is created with the ui.Window.createForm() method (typically to generate forms at runtime).
Configuring windows with styles

Use the `STYLE` attribute to set a style for a window.

By default, windows are displayed as normal application windows. The `WINDOWSTYLE` attribute can be used to define the type of the window, for example to show it as a "modal window".

The window style defines the type of the window (normal, modal) and its decoration, via a presentation style. The presentation style specifies a set of attributes in an external file (`.4st`).

There are different ways to define the style of a window: The `STYLE` attribute on page 1395 can be used in the `OPEN WINDOW` on page 1131 instruction to define the default style for a window, but it is better to specify the window style in the form file, with the `WINDOWSTYLE` attribute of the `LAYOUT section` on page 1301. This avoids decoration-specific code in the programs.

Table 306: Standard window styles defined in the default presentation style file

<table>
<thead>
<tr>
<th>Style name in 4st file</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Window</td>
<td>Defines presentation attributes for common application windows. When using MDI containers, normal windows are displayed as MDI children.</td>
</tr>
<tr>
<td>Window.main, Window.main2</td>
<td>Defines presentation attributes for starter applications, where the main window shows a start menu if one is defined by the application.</td>
</tr>
<tr>
<td>Window.dialog, Window.dialog2, Window.dialog3, Window.dialog4</td>
<td>Defines presentation attributes for typical OK/Cancel modal windows. On iOS mobile devices, opening a new window with the predefined style 'dialog' causes the window to slide up from the bottom:</td>
</tr>
<tr>
<td>Window.naked</td>
<td>Defines presentation attributes for windows that do not show the default view for ring menus and action buttons (OK/Cancel).</td>
</tr>
<tr>
<td>Window.viewer</td>
<td>Defines presentation attributes for viewers as the report pager (fglreport.per).</td>
</tr>
</tbody>
</table>

It is recommended that you *not* change the default settings of windows styles in the `$FGLDIR/lib/default.4st` file. If you create your own style file, copy the default styles into your own file in a different directory.

It is not possible to change the presentation style attributes of windows dynamically in the AUI tree. The style is applied when the window and form are loaded.

If you open and display a second form in an existing window, the window style of the second form is not applied.

**Related concepts**

- **Start menus** on page 2019
  Start menus define a tree of application programs that can be started.

- **The abstract user interface tree** on page 1106
  The abstract user interface tree is the XML representation of the application forms displayed to the end user.

- **Understanding presentation styles** on page 1165
Presentation styles centralize the attributes related to the decoration of the graphical user interface elements.

Related reference

Window style attributes on page 1220
Window presentation style attributes apply to a window element.

Defining the window title

Use the TEXT attribute to define a title for a window.

The TEXT attribute in the ATTRIBUTE clause of OPEN WINDOW on page 1131 defines the default title of the window. If the window is opened with a form (WITH FORM clause) that defines a TEXT attribute in the LAYOUT section, the default is ignored. Subsequent OPEN FORM/DISPLAY FORM instructions may change the window title if the new form defines a different title in the LAYOUT section.

It is recommended that you specify the window title in the form file, instead of using the TEXT attribute of the OPEN WINDOW instruction.

If you want to set a window title dynamically, you can use the setText () method of the ui.Window built-in class.

Defining the window icon

Use a IMAGE attribute to define the icon for a window.

If the window is opened with OPEN WINDOW WITH FORM, by using a form file that defines an IMAGE attribute in the LAYOUT section, the window will use this image as icon.

Subsequent OPEN FORM/DISPLAY FORM instructions may change the window icon if the new form defines a different image in the LAYOUT section.

If you want to set a window icon dynamically, you can use the setImage () method of the ui.Window built-in class.

Window container interfaces

Windows can be displayed in parent containers.

The Window Container Interface (aka MDI) (GDC only)

With the GDC front-end, the windows of several programs can be displayed in a single "WCI" container application, by using the ui.Interface methods to define parent / child relationship.

Note: This feature is only supported by GDC.

For more details, see Window containers (WCI) on page 2023.

Tabbed containers (GBC and GDC+WCI)

With the GBC front-end, and with GDC (using WCI), enable tabbed containers by using the tabbedContainer style attribute for the main window of a parent program:

```
<StyleList>
  <Style name="Window.main">
    <StyleAttribute name="windowType" value="normal" />
    <StyleAttribute name="tabbedContainer" value="yes" />
  </Style>
  ...
</StyleList>
```

When tabbedContainer style attribute is set to "yes", the windows of several child programs will be rendered in folder tabs in a parent container.

Important: With GBC, the parent program using tabbedContainer must define a Start Menu.

For reference, see tabbedContainer in the Window style attributes topic.
Using distinct web browser tabs for each program (GBC)

When using the GBC front-end in a web browser, by default new programs started by a parent program with `RUN ... [WITHOUT WAITING]` will display in the same web browser tab as the parent program.

You can force each program to display its windows in a dedicated web browser tab, by specifying the `browserMultiPage` style attribute at the `UserInterface` node level:

```xml
<StyleList>
  <Style name="UserInterface">
    <StyleAttribute name="browserMultiPage" value="yes"/>
  </Style>
  ...
</StyleList>
```

When the `browserMultiPage` style attribute is set to "yes" for a given program, the subsequent child programs started by a `RUN` command will display in separate browser tabs.

For reference, see `browserMultiPage` in the `UserInterface` style attributes topic.

Related concepts
Presentation styles on page 1165
Use presentation styles to specify decoration attributes for window and form elements.

Position and size of a window
Window objects can be created with a position and size for the TUI mode.

Defining the window position and size
A typical `OPEN WINDOW` instruction for TUI mode specifies the position on the screen:

```plaintext
OPEN WINDOW w1 AT 10,5 WITH FORM "custlist"
```

When using the TUI mode (or traditional GUI mode), the `AT line, column` clause defines the position of the top-left corner of the window on the terminal screen.

The `WITH lines ROWS, characters COLUMNS` clause specifies explicit vertical and horizontal dimensions for the window. The expression at the left of the `ROWS` keyword specifies the height of the window, in character unit lines. This must be an integer between 1 and `max`, where `max` is the maximum number of lines that the screen can display. The integer expression after the comma at the left of the `COLUMNS` keyword specifies the width of the window, in character unit columns. This must return a whole number between 1 and `length`, where `length` is the number of characters that your monitor can display on one line.

Reserved window lines
In addition to the lines needed for a form, consider allowing room for the `COMMENT` line, the `MENU` line, the `MENU comment` line, and the `ERROR` line for the TUI mode.

The runtime system issues a runtime error if the window does not include sufficient lines to display both the form and these additional reserved lines.

The minimum number of lines required to display a form in a window is the number of lines in the form, plus an additional line below the form for prompts, messages, and comments.

Window geometry in GUI mode
When using the full GUI mode (without the traditional mode), the `AT line, column` clause is optional and if used, the `WITH lines ROWS, characters COLUMNS` clause is ignored, because the size of the window is automatically calculated based on its contents, and the position is controlled by the window manager.
Related concepts

Defining the position of reserved lines on page 559
The OPTIONS element LINE defines position of dedicated screen lines.

Using images

Describes how to use pictures in the forms of your application.

Image handling basics

This is an introduction to image handling in Genero.

Purpose of images in applications

Graphical applications typically use images for various purposes:

- Application icon for the operating system taskbar/window manager.
- Icons in pop-up messages, menu options, form buttons, toolbars, list elements, treeview nodes.
- Decoration pictures in forms like background images, company logo, etc.
- Application photos, to get a visual identification for objects or people.

Images can be static (like toolbar icons, logos), or can change during the program execution (images related to application data).

In .per form definition files, specify static or dynamic image form items, with the IMAGE item type on page 1279.

Sources for image data

An image can come from different sources:

- An image file located on the system where the program executes (available on the platform, or from your own application).
- An URL (or URI) resource: the image file is located on a Web server and can be downloaded from the internet.
- Image data stored in a database within Binary Large Object (BLOB) typed columns.
- Pictures coming from a mobile device photo gallery, or camera.

In all cases, the image data must be available locally on the front-end platform to be displayed. Since the program can run on a different platform as the front-end, Genero provides several solutions to transmit the image data to the front-end, when the image is not available as a local file. For more details, see Providing the image resource on page 1149.

Image triggering actions

If needed, it is possible to associate an action to an image by defining the ACTION attribute. The associated action handler will then be executed in the program code, for example to react to mouse clicks on the image for desktop front-ends:

```per
IMAGE i1: logo,
    IMAGE = "genero_logo",
    ACTION = show_about_box;
```

For more details about action handling, see Dialog actions on page 1739.
**Controlling the image layout**
Explains how image form items can be sized in different front-end layout systems.

**Image sizing basics**
It is important to differentiate the image and the image container (the widget): when designing a form, you're defining the image container. The actual image that will be displayed in this container can be smaller or larger. Genero provides several form file attributes, to control how the image and its container are sized.

How an IMAGE item renders on the front-end screen depends on these factors:
- The type of layout used (grid-based or stack-based layout).
- The size of the form item tag in the LAYOUT section, or the WIDTH and HEIGHT attributes defined for the IMAGE item.
- The combination of image item attributes (SIZEPOLICY, AUTOSCALE, STRETCH). These attributes may have a limited effect depending on the front-end platform.
- The image resource (actual picture file) size when displayed (especially when SIZEPOLICY=DYNAMIC/ INITIAL).
- The scaleIcon presentation style attribute, for elements using icons such as BUTTON or TOOLBAR items.

**Note:** The typical layout settings of an IMAGE item is a combination of SIZEPOLICY=FIXED, AUTOSCALE and optionally a WIDTH/HEIGHT: The fixed size policy is to get an image size that corresponds the rest of the form layout, while auto-scaling will make the image fit to its container and avoid scrollbars with large images. The WIDTH/HEIGHT attributes may be used to specify the size of the image container. However, the preferred way to define the size of an image container is the item tag dimensions, in the layout section of the form.

**Image size in grid-based layout**
The AUTOSCALE attribute indicates if the picture must be scaled to the available space in the image item. The space is defined by the SIZEPOLICY attribute, the STRETCH attribute, and the form item size (the form item tag in the layout or the WIDTH and HEIGHT attributes).

AUTOSCALE is only useful if the size of the image differs from the size of the container. AUTOSCALE is useless with SIZEPOLICY=DYNAMIC, as the container always fits to the image size.

The STRETCH attribute defines how the image item adapts to the parent container when it is re-sized. The default is NONE.

SIZEPOLICY and the WIDTH/HEIGHT attributes define the size of the container, not the size of the image.

The SIZEPOLICY attribute defines how the image widget gets its size, depending on the context:
- When SIZEPOLICY is INITIAL (the default) and AUTOSCALE is not used, the size of the widget is defined by the first picture displayed in the form element. The size will not change if other pictures with different sizes display in the widget. If no initial image is displayed (the image field value is NULL), the form item does not take up space in the layout (and also does not adapt the size if an image is displayed later on).
- When SIZEPOLICY is DYNAMIC, the size of the widget is automatically adapted to the size of the pictures displayed in the image form item. The AUTOSCALE attribute has no effect.
- If SIZEPOLICY attribute is set to FIXED, the size of the widget is defined by the form specification file, either by the size of the item-tag in the layout, or by the WIDTH and HEIGHT attributes. With a fixed image widget size, if AUTOSCALE is not used, scrollbars may appear if the picture is greater than the widget.

By default, the size of the image widget defaults to the relative width and height defined by the item-tag in the form layout section. The size of an image widget can also be specified in the WIDTH and HEIGHT attributes, but these attributes only have an effect when SIZEPOLICY=FIXED.

The WIDTH and HEIGHT attributes define the size of the container, but they are dependent on the SIZEPOLICY. It means the image container may grow or shrink even if WIDTH and HEIGHT are specified. If you really want to have a container with a fixed size, you have to use WIDTH and HEIGHT in combination with SIZEPOLICY=FIXED.
All image layout attributes (except AUTOSCALE) only have an impact on the container size, not on the image size. If AUTOSCALE is defined, the image fits to the container size, but without losing its initial proportionality. If STRETCH allows the container to grow/reduce in the X or Y direction, the image will grow with the container, but the original proportionality is always kept.

**Note:** On some platforms, the image widgets automatically add a border to the source picture. For these platforms, if the image form item is the same size as the image, you may need to increase the size of the image form item to avoid automatic scrollbars. For example, if your image source has a size of 500x500 pixels and the widget displays a border with a size of 1 pixel, you will have to set WIDTH and HEIGHT to 502 pixels. If you do not, scrollbars will appear or the image will shrink if AUTOSCALE is used. Alternatively, you can avoid the image border with the border presentation style attribute.

### Image size in stack-based layout

With a stacked layout, where form items display vertically on each other, by default the image is auto-scaled with the correct aspect/ratio in the available form space.

The image size can be controlled by the HEIGHT attribute of the image container.

If the HEIGHT attribute is set, it is expressed in CHARACTERS as for grid-based layout, and the width is determined by the correct aspect/ratio.

**Related concepts**

*Form rendering basics* on page 1415
Get the essentials about form rendering.

**Related reference**

*Button style attributes* on page 1185
Button presentation style attributes apply to BUTTON elements.

### Providing the image resource

There are several things you need to know about providing an image resource in a Genero program.

### Supported image formats

Genero supports various image data formats, typically PNG, JPEG and SVG.

True Type Font (TTF) files are also supported: The TTF format is used when image-to-font-glyph mapping is enabled by specifying a mapping file in the FGLIMAGEPATH environment variable.

#### Table 307: List of image file formats supported by different front-ends

<table>
<thead>
<tr>
<th>Image file format</th>
<th>GDC (native rendering mode)</th>
<th>GBC (or UR, see note 3)</th>
<th>GMA (native rendering mode)</th>
<th>GMI (native rendering mode)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.bmp</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>.gif</td>
<td>Yes (see note 1)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>.ico</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>.jpg</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>.png</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>.svg</td>
<td>Yes (see note 2)</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>.tiff</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Notes:**
1. With GDC, animated GIFs are only available for the `IMAGE` form item. They are not supported where the image appears due to the `IMAGE` attribute (in buttons, toolbars, topmenus and so on). For performance reasons, animated GIFs are not supported in `TABLE` containers.

2. The GDC front-end supports SVG through the Qt library. Qt only supports the static features of SVG 1.2 Tiny. For more information, see Rendering SVG Files in the Qt Documentation.

3. When using Universal Rendering, the rules for GBC image format support apply.

Image resolution

Consider using the appropriate image resolution for the target front-end platform. For example, mobile devices have a much higher pixel density (a higher resolution) than desktop monitors. An image which looks nice on a desktop can appear small or as an unscaled image on a mobile device.

Static versus dynamic images

The image resource specification is different for static and dynamic images:

- For static images (such as button icons), set the image resource in the image attribute (’IMAGE’, ’IMAGELEAF’, and so on). See Static images on page 1153.
- For dynamic images (such as image fields displaying photos from a database), the image resource is specified with the field/variable value, to be rendered in a form field. The form field is typically defined as an `IMAGE` item, or an `IMAGECOLUMN` in a table view. For more details, see Runtime images on page 1154.

Image resource lookup

The image data can be provided in different ways, depending on the image resource specification:

1. As a Uniform Resource Locator (URL), such as "https://4js.com/files/images/fourjs_logo.jpg".
2. As a simple image name (for TTF icons), such as "smiley".
3. As a simple file name, relative or absolute file path, with image extension (such as .png or .jpg).

Using an URL image resource

If the image specification starts with a URL prefix, the front-end will try to download the image from the location specified by the URL.

The network access to the Web server must exist and network bandwidth must be sufficient to rapidly download the images.

<table>
<thead>
<tr>
<th>Image resource location (URL)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://location-specification">http://location-specification</a></td>
<td>HTTP server</td>
</tr>
<tr>
<td><a href="https://location-specification">https://location-specification</a></td>
<td>HTTPS server (HTTP over SSL/TLS)</td>
</tr>
</tbody>
</table>

Using a simple image name (centralized icons)

If the image specification is a simple name (without a file extension), and the FGLIMAGEPATH environment variable defines an icon mapping file for the runtime system, the image name is converted to a font file and font glyph, based on the mapping file entries, and the image form item displays the glyph/icon found in the font definition file. The mapping file and the font definition file are centralized on the application server.

A line in the image-to-font-glyph mapping file must have the following format:

```
image-name=font-file:hexa-ordinal[:color-spec]
```
For example, if the image mapping file defines the following lines:

```plaintext
smiley=FontAwesome.ttf:f118
red_smiley=FontAwesome.ttf:f118:#8B0000
```

An image resource (IMAGE attribute, IMAGECOLUMN value, and so on) with the name "smiley" will be mapped to the glyph with ordinal position 0xf118 in the FontAwesome.ttf font file, and the image resources using "red_smiley" will use the same glyph, but will get a red color.

**Important:** The directory to the font file must be specified in FGLIMAGEPATH, except if the font file is located in the same directory as the mapping file.

A default color can be defined for all TTF icons of a window, by using the defaultTTFColor style attribute:

```xml
<StyleList>
  <Style name="Window.important">
    <StyleAttribute name="defaultTTFColor" value="red" />
  </Style>
  ...
</StyleList>
```

A default mapping file named "image2font.txt" and the "FontAwesome.ttf" font file are provided in $FGLDIR/lib. If FGLIMAGEPATH is not defined, the runtime system will use these files to make the image to font glyph mapping.

**Important:** When providing your own customized font file, it must be a valid TTF file. For example, changing the file name is not sufficient to turn it into a different font: In order to produce a valid TTF file, use font management tools such as FontForge (http://fontforge.github.io/en-US/) or Fontello (http://fontello.com). Furthermore, to target Microsoft® Internet Explorer (version 11), you will need to patch the generated TTF file to remove embedding limitations from TrueType fonts, by setting the fsType field in the OS/2 table to zero. This modification can be done with freeware tools like ttembed.

It is possible to mix several plain image file directories with several image-to-font glyph mapping files in FGLIMAGEPATH. The list of mapping files and directories defines the order of precedence, for example:

```bash
$ export FGLIMAGEPATH="/var/myapp/myimages:\n/var/myapp/myicons.txt:/var/myapp/fontfiles:\n$FGLDIR/lib/image2font.txt:$FGLDIR/lib"
```

In the above FGLIMAGEPATH configuration:

- /var/myapp/myimages: Directory where plain image files can be found
- /var/myapp/myicons.txt: Custom image-to-font-glyph mapping file (icons)
- /var/myapp/fontfiles: Font files used by the myicons.txt mapping file
- $FGLDIR/lib/image2font.txt: Default icon mapping files (using FontAwesome.ttf)
- $FGLDIR/lib: Location of the default FontAwesome.ttf file

Consider defining your own image mapping file and make FGLIMAGEPATH point to your own files.

**Note:** When executing the application on a mobile device, you must define the FGLIMAGEPATH environment variable with the mobile.environment.FGLIMAGEPATH entry in FGLPROFILE. Use $FGLAPPDIR and $FGLDIR placeholders to include the current appdir (program file directory) and the FGL runtime system directory, respectively.

See FGLIMAGEPATH on page 276 for more details about this environment variable.

**Using file names or paths**

If the image specification is a simple file name, relative or absolute path (without an URL prefix, and with a image file extension), the front-end gets the image file from the runtime system.

```plaintext
DISPLAY "bird.jpg" TO img1
```
The image file is searched on the platform where the program executes. The runtime system uses the FGLIMAGEPATH on page 276 environment variable when searching for the images. If FGLIMAGEPATH is not set, the current working directory is searched for the image files.

If FGLIMAGEPATH is defined, the current working directory is not searched. To find image files in the current working directory and in other directories, add "." to the FGLIMAGEPATH path list.

**Important:** Always use the image file extension (.png, .jpg): If the image file is specified without a file extension, Genero will try to find the image file with a predefined list of extensions (.png, .jpg, etc). The search depends on the name of the image file, the list of directories defined in FGLIMAGEPATH, and the predefined list of file extensions. This search procedure has been implemented to allow different types of front-ends to pass the preferred image compression formats, and allow you to specify the image name without any file extension in programs and forms. However, it is much more efficient to use images with common formats (such as .png or .jpg), and specify this extension in the file name. The file extension will also be used by the front-end to easily identify the compression format (for example, to define the Content-Type in an HTML entity).

**The resource file cache of the front-end**

The resource files transmitted by the runtime system to the front-end are automatically cached on the front-end workstation, to be reused by several programs and program instances.

If the content of the resource file on the program server side changes, and the image filename is redisplayed to the form field (by a DISPLAY TO / DISPLAY BY NAME or implicitly with the UNBUFFERED mode) as a new field value, the resource file content will be retransferred to the front-end.

For example, when a form holds an IMAGE field, the following code sequence will produce the results described in the inline comments (starting with an empty front-end file cache):

```plaintext
DISPLAY "/tmp/bird.png" TO image1  -- bird.png is transferred and cached.
DISPLAY "/tmp/cat.png" TO image1  -- cat.png is transferred and cached.
DISPLAY "/tmp/bird.png" TO image1  -- bird.png is cached, no transfer needed.
DISPLAY "/tmp/cat.png" TO image1  -- cat.png is cached, no transfer needed.
RUN "touch /tmp/bird.png"  -- updates modification timestamp of bird.png.
DISPLAY "/tmp/bird.png" TO image1  -- bird.png is retransferred and recached.
```

The **standard.clearFileCache** front call can be used in development or for debug purpose, to force the front-end to clear its resource file cache. However, this front call is not required in a production context:

```plaintext
DEFINE res BOOLEAN
CALL ui.Interface.frontCall("standard","clearFileCache",[],[res])
IF NOT res THEN
  ERROR "A problem occurred while clearing the front-end cache..."
END IF
```

**Important:** This feature is provided for development and debug purpose, and must not be used in a production environment.

Clearing the front-end file cache will retransfer resources that have already been transferred (and previously cached):

```plaintext
DISPLAY "/tmp/bird.png" TO image1  -- bird.png is transferred and cached.
DISPLAY "/tmp/cat.png" TO image1  -- cat.png is transferred and cached.
CALL ui.Interface.frontCall("standard","clearFileCache",[],[res])
DISPLAY "/tmp/bird.png" TO image1  -- bird.png is transferred and cached.
DISPLAY "/tmp/cat.png" TO image1  -- cat.png is transferred and cached.
```
Application images in Web Components

Web Components can display static images (part of the Web Component assets), and application images provided at runtime (for example, a photo gallery web component). In order to provide application images to a Web Component, the program must use the `ui.Interface.filenameToURI()` method to convert the local file name to a URI that can be accessed by the front-end.

For more details, see Using image resources with the gICAPI web component on page 1952.

Static images

Describes how to decorate forms with icons.

Static image usage context

Static images are application pictures that do not change during program executing, like icons in toolbar buttons and window icons.

Static images can be defined in different contexts within form definition, or configuration files:

- Global application icon for platform window managers (taskbars), by using the `ui.Interface.setImage` on page 2485 method. The recommendation for mobile devices is that the application icon is provided in the installation package (.apk for Android™, .ipa for iOS).
- Window specific icons, with the `IMAGE` attribute in the `LAYOUT definition` of a form (recommended) or at runtime, with the `ui.Window.setImage` on page 2495 method (if it must be changed during program execution).
- As default icon for action views, with the `IMAGE` action configuration attribute (in action defaults for example).
- As specific action view icons, directly in the form item definition with the `IMAGE` attribute (for toolbars, menu items, buttons, buttonedits, etc).
- Image form items (logos), defined by the `IMAGE item-tag : item-name` syntax, using the `IMAGE` attribute.
- Default treeview node icons, with the `IMAGEEXPANDED, IMAGECOLLAPSED, IMAGELEAF` attributes of a `TREE` container.

Static image examples

The following code example, defines an ITEM toolbar element using a icon, that is specified with the `IMAGE` attribute:

```plaintext
TOOLBAR
  ITEM print ( TEXT="Print", IMAGE="printer" )
```

Next example defines a BUTTONEDIT form field with an icon named "listchoice":

```plaintext
ATTRIBUTES
  BUTTONEDIT f05 = customer.cust_city,
    ACTION=get_city,
    IMAGE="listchoice",
    ...
```

Related concepts

Configuring actions on page 1744
Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with *action attributes*.

**Runtime images**
Explains how to display pictures at runtime.

**Dynamic image usage context**
Application images like photos or variable icons (in list views) are only known at runtime, and are displayed during program execution. Such images are typically centralized on a server, as BLOBs in a database, or on the file system, as regular files.

For simple files (not URLs), images to be displayed are automatically handled by Genero; the program just needs to specify the name of the file to be displayed. The image resources is then transmitted to the front-end.

This section describes programming patterns to handle application images. For a complete description of the mechanisms to provide images to front-ends, see *Providing the image resource* on page 1149.

**IMAGE form fields**
To display a picture dynamically in a form area, you must define a form field with the *IMAGE* item type:

```
LAYOUT
GRID
{
    [img1                   ]
    [                   ]
    [                   ]
} END
END
ATTRIBUTES
IMAGE img1 = FORMONLY.image_field, AUTOSCALE, ...
```

The program can then display an image dynamically by assigning the image resource to the form field, for example, with a `DISPLAY TO` instruction:

```
DEFINE image_field STRING
LET image_field = "local_image_file.png"
DISPLAY BY NAME image_field
```

It is also possible to use the program variable containing the image resource in a dialog using the `UNBUFFERED` option:

```
DEFINE rec RECORD
    pk INT,
    name VARCHAR(30),
    image_field VARCHAR(50)
END RECORD
INPUT BY NAME rec.* ATTRIBUTES(UNBUFFERED)
ON ACTION set_picture
    LET rec.image_field = "local_image_file.png"
...
```

**IMAGECOLUMN attribute of TABLE/TREE**
The *IMAGECOLUMN* attribute can be used to define a PHANTOM field that will hold the image resource for a TABLE or TREE column:

```
...
ATTRIBUTES
PHANTOM FORMONLY.item_icon;
```
EDIT FORMONLY.item_desc, IMAGECOLUMN=item_icon;
...
END
INSTRUCTIONS
SCREEN RECORD sr(FORMONLY.item_icon, FORMONLY.item_desc, ...);
...

In the program code, the image resource is specified in the array member attached to the icon field. Each row can define a different image for the cell:

```plaintext
LET arr[1].item_icon = "honda_logo.png"
LET arr[1].item_desc = "Honda CB600 Hornet (red)"
LET arr[2].item_icon = "honda_logo.png"
LET arr[2].item_desc = "Honda CB1000r (black)"
LET arr[3].item_icon = "ducati_logo.png"
LET arr[3].item_desc = "Ducati Diavel Carbon"
DISPLAY ARRAY arr TO sr.*
...
```

### Displaying images contained in BYTE variables

Application images managed by a program can be held in a **BYTE** on page 290 variable. This data type is required to hold the data of Binary Large OBject (BLOB) database columns.

When using an **IMAGE** field, if the BYTE variable holding the image data is located in a file (`LOCATE IN FILE [filename]`), the runtime system can automatically send the content of the BYTE file to the front-end when doing a **DISPLAY BY NAME, DISPLAY TO** field, or if the BYTE variable is controlled by a dialog using the **UNBUFFERED** option.

```plaintext
DEFINE pb BYTE
LOCATE pb IN FILE -- temp file used
...
OPEN FORM f1 FROM "myform"
DISPLAY FORM f1
...
SELECT image_col INTO pb FROM mytable WHERE pk = ...
DISPLAY pb TO image_field
...
```

Furthermore, if the image data is modified, without changing the name of the file (without a new **LOCATE IN FILE** instruction), the image data is transmitted again to the front-end. For example, consider the following program flow:

```plaintext
DEFINE pb BYTE
LOCATE pb IN FILE -- temp file used
...
-- A first SELECT fetches image data from row 345 into the BYTE
SELECT image_col INTO pb FROM mytable WHERE pk = 345
-- And displays the BYTE image to a field
DISPLAY pb TO image_field
-- A second SELECT fetches new image data from row 672 into the BYTE
SELECT image_col INTO pb FROM mytable WHERE pk = 672
-- And displays the BYTE image to a field
DISPLAY pb TO image_field
-- The BYTE file name has not changed, only the image data has changed
...
```
Images on mobile devices

When executing the application on a mobile device, it is possible to use a front call to choose or take a photo. Those front calls return an opaque file identifier referencing an image in the device photo gallery (or database).

On all mobile platforms, you can directly display the returned opaque file path to an IMAGE form field:

```define path string
-- Here we use "choosePhoto" front call, could be "takePhoto"
call ui.interface.frontcall("mobile", "choosePhoto", [], [path])
display path to ff_image
```

Consider the path returned by such a front call as an opaque local file identifier, and do not use it as a persistent file name for the picture. For example, if you store such a path name in a database, and if the mobile photo gallery storage technology changes, the stored file names will no longer be valid.

If you need to keep the image data in the application (to store it in a local file or in the database), grab the image data into the runtime system context with a `fgl_getfile()` call. The mobile picture path can be used in a `fgl_getfile()` call to the photo from the mobile device into the file storage context where the runtime system executes. When the runtime system executes on the mobile device, the `fgl_getfile()` call will copy the picture to the application sandbox. If the program executes on an application server, the call will transfer the picture to the application server file system. It is possible to load the picture data into a `BYTE` variable by transferring the image data directly into the file used by the `BYTE` variable located in `byte_file`, by doing a `fgl_getfile(mobile_path, byte_file)`. It is also possible to keep the transferred files on the file system where the VM executes, if you do not want to use `BYTE` variables to store images in your database.

```constant vm_fn = "mypic.tmp"
define md_fn string, image byte
call ui.interface.frontcall(
    "mobile",
    "choosePhoto", -- could be "takePhoto"
    [], [md_fn])
call fgl_getfile(md_fn,vm_fn)
locate image in file vm_fn
display image to ff_image
update mytab set pic = image where ...
```

Note: When using `fgl_getfile()` with `BYTE` variables located in files, pay attention to the fact that `initialize byte_var to null` will set the internal null indicator of the `BYTE` variable, and a subsequent `fgl_getfile(mobile_path, byte_file)` will only modify the file without touching the null flag. The recommended pattern is to re-locate the `BYTE` variable after the `fgl_getfile()` call:

```call fgl_getfile(mobile_path, byte_file)
locate byte_var in file byte_file
```

Videos on mobile devices

Let the user take videos or choose videos from the gallery with the `takeVideo` on page 2712 and `chooseVideo` on page 2700 front calls.

Similar to photo front calls, the video front calls return an opaque path to the video file, which can then be used in the `fgl_getfile()` function to transfer the video file from the device context to the runtime system context in a `BYTE` variable for persistent storage.

Note: The opaque path can, however, be used to show the video with the "launchURL" front call.

For example:

```import os
constant vm_movies = "/movies"
```
MAIN
DEFINE r INTEGER,
mb_path STRING,
vm_path STRING
LET r = os.Path.delete(VM_MOVIES)
LET r = os.Path.mkDir(VM_MOVIES)

MENU
COMMAND "take_video"
  CALL ui.Interface.Frontcall("mobile", "takeVideo", [], [mb_path])
  IF mb_path IS NOT NULL THEN
    LET vm_path = SFMT("%1/%2", VM_MOVIES, os.Path.baseName(mb_path) )
    CALL fgl_getfile(mb_path, vm_path)
  END IF
COMMAND "choose_video"
  CALL ui.Interface.Frontcall("mobile", "chooseVideo", [], [mb_path])
  IF mb_path IS NOT NULL THEN
    LET vm_path = SFMT("%1/%2", VM_MOVIES, os.Path.baseName(mb_path) )
    CALL fgl_getfile(mb_path, vm_path)
  END IF
COMMAND "show_video"
  IF mb_path IS NOT NULL THEN
    CALL ui.Interface.Frontcall("standard", "launchURL", [mb_path], [])
  END IF
COMMAND "quit"
  EXIT MENU
END MENU
END MAIN

Accessibility guidelines
This section describes the best practices to make your application accessible to disabled people.

Keyboard access
How to implement keyboard usage to follow accessibility standard?

Defining keyboard accelerators for every action
Since a mouse or other pointing devices may not be used by people with reduced vision, an accessible application must be usable with the keyboard alone. Therefore, all the possible actions that could be triggered by a user must have a keyboard shortcut.

We strongly suggest that you define consistent keyboard shortcuts for all actions through the use of action defaults. Developers can avoid overriding the system default shortcuts by checking the target platform guidelines, especially for system shortcuts that trigger accessible actions (for example, Ctrl-Shift-Enter, which triggers spoken information about the currently selected item). Overriding system shortcuts is generally a bad practice, even for non-accessible applications, although overriding may be unavoidable due to compatibility issues.

Keyboard focus and action views
Generally, keyboard navigation in an application may be easier if you keep the MENU actions in the menu frame; the actions can have the keyboard focus and the user can navigate through them using the up and down arrows.

Using topmenus is a good practice, as the users can pull it down with the keyboard (for example, the Alt key on Windows®) and then navigate using arrow keys, but it may be less accessible than the menu panel. To ensure that every item of the menu can be activated by a keyboard shortcut, use the & (ampersand) before a character in menu items to indicate the keyboard shortcut that triggers the menu action with that letter.
Avoid using toolbars alone in accessible applications, because toolbars by default are not accessible using the keyboard. Toolbars cannot have the keyboard focus, and there is no way to navigate through all toolbar items or to activate one of them using the keyboard. If you do use toolbars, provide keyboard shortcuts and duplicate them in a topmenu.

Related concepts
Configuring actions on page 1744
Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with action attributes.

Form description for screen readers
How to integrate with platform screen readers?

Understanding screen readers
Screen readers are special system applications that transform the application's graphical user interface into speech. The behavior may change between screen reader implementations, but, basically, each widget is named and described by speech. On some workstation operating systems, special keyboard shortcuts are available to trigger the complete enumeration of all the components of a window, or to describe only the component having the current focus.

Providing form item descriptions to screen readers
Screen readers use special bindings to get the information that they need (name, full description, hierarchy, triggered actions, and so on) about each graphical component of the entire graphical user interface. It is up to the programmer to provide these bindings to the screen reader, but most of the work is already done by the front-end.

Programmers can provide two things for each widget to provide speech information to screen readers:

• an accessible name, using the TEXT form attribute if available, otherwise with the COMMENT form attribute.
• an accessible description, with the COMMENT form attribute.

This can be tedious, but it absolutely must be done carefully, keeping in mind that the text will be spoken. As such, customer's name is preferable to cust_name_str.

Spaces and punctuation are allowed.

Most of the form items are supported: All kind of form field, static labels, static images, and action-based items (such as buttons); some containers (GROUP and FOLDER) work out of the box as soon as their TEXT attributes are set.

Examples
In an action defaults file (mydefaults.4ad)

```xml
<ActionDefaultList>
  <ActionDefault name="new" text="New..." image="new.svg"
               comment="Create a new database"
               acceleratorName="Control-N" />
  <ActionDefault name="open" text="Open..." image="open.svg"
               comment="Open an existing database"
               acceleratorName="Control-O" />
  <ActionDefault name="save" text="Save" image="save.svg"
               comment="Save the current database"
               acceleratorName="Control-S" />
  ...
</ActionDefaultList>
```

In field definitions on a form specification file (myform.per)

```xml
<ATTRIBUTES>
  EDIT login_name = formonly.login_name, NOT NULL, 
               COMMENT="Login name of the current user";
  EDIT password = formonly.password, NOT NULL, INVISIBLE, VERIFY, 
               COMMENT="Password of the current user";
  EDIT first_name = formonly.first_name, NOT NULL, 
               COMMENT="First name of the current user";
</ATTRIBUTES>
```
In this form specification file, the COMMENT attribute is used for both the accessible name and the accessible description.

**Usability and ergonomics**

How to adapt application forms to disabled users?

**Design simple application forms**

Keep your forms as simple as possible. Because everything will be described by the screen reader software, it is preferable to have a lot of small and concise forms with a few fields. With forms containing a lot of labels and fields, the screen reader will take a long time to enumerate every name and description. The end user must be able to create an image of the form in his mind, based on the form description.

**Make form content bigger**

Consider using a special .4st presentation styles file defining big fonts, big icons, and high contrast color themes; this will make your application a lot more efficient for users who are partially sighted. Forms will take more space on the screen, assuming that the forms have a limited number of fields to have sufficient room for large widgets.

Use large icons (such as 64x64 pixel icons), for people with impaired vision. Do not forget that most of the default sizes (font, icons, GUI components, and so on) were set when the default resolution was 640*640 pixels in 16 colors. Now, even if the user has very good sight, with the screen resolution available today, old-style icons look small.

Use a high contrast color theme. Although support of the system high contrast theme is only partial, nothing prevents you from setting up the correct theme using a specific presentation style attributes.

**Example**

Presentation styles file defining larger, bolder fonts and larger icons:

```
<StyleList>
  <Style name="*" >
    <StyleAttribute name="fontSize" value="10" />
  </Style>

  <Style name="Action" >
    <StyleAttribute name="scaleIcon" value="28px"/>
    <StyleAttribute name="fontSize" value="12" />
  </Style>

  <Style name="Window" >
    <StyleAttribute name="actionPanelPosition" value="bottom"/>
    <StyleAttribute name="actionPanelButtonSpace" value="huge"/>
    <StyleAttribute name="actionPanelHAlign" value="center"/>
    <StyleAttribute name="ringMenuPosition" value="bottom"/>
    <StyleAttribute name="ringMenuButtonSpace" value="huge"/>
    <StyleAttribute name="ringMenuHAlign" value="center"/>
  </Style>

  <Style name="ToolBar" >
    <StyleAttribute name="scaleIcon" value="32px"/>
  </Style>

  <Style name="Edit:focus" >
    <StyleAttribute name="color" value="blue"/>
  </Style>
</StyleList>
```
Related concepts
Presentation styles on page 1165
Use presentation styles to specify decoration attributes for window and form elements.

Message files
Message files centralize strings and larger texts identified by a number, that can be used in programs.

Understanding message files
This is an introduction to message files.

Message files define text messages with a unique integer identifier.

Several message files can be created and loaded by the same program.

Message files are typically used to implement application help system, and are especially designed for the TUI mode.

In order to use a message file, do the following:

1. Create the .msg source message file with a text editor.
2. Compile the source message file with fg1mkmsg to create the .iem binary format.
3. Copy the binary file to a distribution directory.
4. In programs, specify the message file with the OPTIONS HELP FILE instruction.
5. Use a specific message with the HELP clause of dialogs, or load a given message with the SHOWHELP () function.

Message files provide a simple way to implement a help system in your application.

For other application messages and texts, consider using localized strings instead of message files.

Related concepts
Localization on page 512
Localization support allows you to implement programs that follow specific language and cultural rules.

Localized strings on page 538
Localized strings provide a means of writing applications in which the text of strings can be customized on site.

Syntax of message files (.msg)
A message file contains a set of messages identified by an integer number.

filename.msg

1. filename is the name of the message source file.

Syntax of a message file

```markdown
message-definition
include-directive
new-page
```

where message-definition is:

```markdown
.message-number
message-line
```
where *include-directive* is:

```
.include filename
```

And where *new-page* is:

```
^L (Control-L, ASCII 12)
```

1. *message-number* is an integer in the range -2147483648 to 2147483647.
2. You can split the message into pages by adding the `^L` (Control-L / ASCII 12) in a line.
3. Note that multi-line messages will include the newline (ASCII 10) characters.

**Compiling message files (.msg)**

The `.msg` message files must be compiled to `.iem` binary files, in order to be loaded by the runtime system.

In order to use message files in a program, the message source files (with `.msg` extension) must be compiled with the `fglmkmsg` on page 2077 utility to produce compiled message files (with `.iem` extension).

The following command line compiles the message source file `mess01.msg`:

```
fglmkmsg mess01.msg
```

This creates the compiled message file `mess01.iem`.

For backward compatibility, you can specify the output file as second argument:

```
fglmkmsg mess01.msg mess01.iem
```

The `.iem` compiled version of the message file must be distributed on the machine where the programs are executed.

**Using message files at runtime**

In order to use compiled message files (.iem) in programs, specify the current message file with the `OPTIONS HELP FILE` command:

```
OPTIONS HELP FILE "mymessages.iem"
```

The message file provided in the `OPTIONS HELP FILE` command is searched for in several directories, as described in the `FGLRESOURCEPATH` reference topic.

After the message file is defined, you can start the help viewer by calling the `SHOWHELP()` function:

```
CALL showhelp(1242)
```

Use the `HELP` clause in a dialog instruction such as `INPUT` to define particular message number for that the dialog:

```
INPUT BY NAME ... HELP 455
```

The help viewer will automatically display the message text corresponding to the number when the user presses the help key. By default, the help key is Ctrl-W in TUI mode and F1 in GUI mode.

Note that you can implement your own help viewer by overloading the `SHOWHELP()` function defined in `$FGLDIR/src/fglhelp.4gl`. This allows you to customize the help system for your application.
Examples
Message files (.msg) usage examples.

Example 1: Help message file used in a MENU

The message source file `help.msg`:

```
.101
This is help about option 1
.102
This is help about help
.103
This is help about My Menu
```

Compiling the message file:

```
$ fglmkmsg help.msg
```

Program using the `.iem` compiled message file.

```
MAIN
  OPTIONS
    HELP FILE "help.iem"
  MENU "Sample"
    COMMAND "Option 1" HELP 101
      DISPLAY "Option 1 chosen"
    COMMAND "Help"
      CALL showhelp(103)
  END MENU
END MAIN
```

Action defaults files

Action defaults files allow to centralize action configuration parameters such as text, icon, accelerators and behavior options in XML format.

Understanding action defaults files

This is an introduction to action default files.

Action defaults files define the defaults for action attributes in an XML file. These defaults can be overwritten with form item attributes, or with dialog action handler attributes, when using default action views.

This section describes only the `.4ad` action defaults file reference, for more details see Configuring actions on page 1744.

Syntax of action defaults files (.4ad)

A `.4ad` action default file is an XML file defining default attributes for actions.

Action defaults are defined in the `.4ad` file with this syntax:

```
<ActionDefaultList>
  <ActionDefault name="action-name" [ attribute=value [.....] ] />
  [...] 
</ActionDefaultList>
```

1. `action-name` identifies the action.
2. `attribute` is the name of an attribute.
3. `value` defines the value to be assigned to `attribute`. 

Action default attributes reference (.4ad)
This topic contains all attributes you can define in a .4ad action defaults file.

Table 309: Action default attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name = &quot;action-name&quot;</td>
<td>This attribute identifies the action.</td>
</tr>
<tr>
<td>text = &quot;action-label&quot;</td>
<td>The default label to be displayed in action views (typically, the text of buttons).</td>
</tr>
<tr>
<td>comment = &quot;action-comment&quot;</td>
<td>The default help text for this action (typically, displayed as bubble help).</td>
</tr>
<tr>
<td>image = &quot;action-icon&quot;</td>
<td>The default image file to be displayed in the action view.</td>
</tr>
<tr>
<td>acceleratorName = &quot;key-name&quot;</td>
<td>The default accelerator key that can trigger the action, as defined in Keyboard accelerator names on page 1773.</td>
</tr>
<tr>
<td>acceleratorName2 = &quot;key-name&quot;</td>
<td>The second default accelerator key that can trigger the action, as defined in Keyboard accelerator names on page 1773.</td>
</tr>
<tr>
<td>acceleratorName3 = &quot;key-name&quot;</td>
<td>The third default accelerator key that can trigger the action, as defined in Keyboard accelerator names on page 1773.</td>
</tr>
<tr>
<td>acceleratorName4 = &quot;key-name&quot;</td>
<td>The fourth default accelerator key that can trigger the action, as defined in Keyboard accelerator names on page 1773.</td>
</tr>
<tr>
<td>defaultView = {&quot;yes&quot;</td>
<td>Defines whether the front-end must show the default action view (buttons in action panel).</td>
</tr>
<tr>
<td></td>
<td>yes&quot;,&quot;no&quot;,&quot;auto}\</td>
</tr>
</tbody>
</table>

Values can be:

- "no" the default action view is never visible.
- "yes" the default action view is always visible, if the action is visible (ui.Dialog.setActionHidden).
- "auto" the default action view is visible if no other action view is explicitly defined and the action is visible (ui.Dialog.setActionHidden).

The default is "auto".

See also: DEFAULTVIEW attribute on page 1362
### Attribute `contextMenu` = {"yes", "no", "auto"}

Defines whether the front-end must render the action in the default context menu.

Values can be:
- "no" the context menu option is never visible.
- "yes" the context menu option is always visible, if the action is visible (`ui.Dialog.setActionHidden`).
- "auto" the context menu option is visible if no other action view is explicitly defined and the action is visible (`ui.Dialog.setActionHidden`).

The default is "yes".

See also: [CONTEXTMENU attribute](#) on page 1359

### Attribute `validate` = "no"

Defines the behavior of data validation when the action is invoked.

Values can be:
- "no" no data validation is done (field text only available in input buffer).

By default, data validation is driven by the dialog mode (UNBUFFERED or default mode).

For more details, see [Data validation at action invocation](#) on page 1762.

See also: [VALIDATE attribute](#) on page 1403

---

### Related concepts

- [Configuring actions](#) on page 1744
  - Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with action attributes.
- [ON ACTION block](#) on page 1478

### Using action defaults files

To use action default files, you must understand how they work and how to structure the code.

Global action defaults are defined in an XML file with the `.4ad` extension. By default, the runtime system searches for a file named `default.4ad` in several directories as described in the [FGLRESOURCEPATH](#) reference topic. If no file was found, standard action default settings are loaded from the `$FGLDIR/lib/default.4ad` file.

Action defaults files usage is related to action configuration concepts. For more details, see Configuring actions on page 1744.

- [Related concepts](#) on page 279
  - [FGLRESOURCEPATH](#) on page 279
    - Defines search path for resource files.

### Examples

- [Action defaults files (.4ad) usage examples](#)

**Example 1: Loading a global action defaults file**

Some action defaults in XML format (exit action has Localized Strings):

```
<ActionDefaultList>
  <ActionDefault name="print" text="Print" image="printer">
```
The program loading the action defaults file:

```c
MAIN
    CALL ui.Interface.loadActionDefaults("mydefaults")
    OPEN FORM f FROM "myform"
    DISPLAY FORM f
    ...
END MAIN
```

### Presentation styles

Use presentation styles to specify decoration attributes for window and form elements.

**Understanding presentation styles**

*Presentation styles* centralize the attributes related to the decoration of the graphical user interface elements.

The decoration attributes are defined in a separate file, which can be easily modified to customize the application.

Presentation styles are only supported for the GUI front-ends. If you design an application for the TUI mode, you can use TTY attributes.

Styles are applied implicitly by using global styles, or explicitly by naming a specific style in the `style` attribute of the element.

Common presentation attributes define font properties, foreground colors and background colors. Some presentation attributes are specific to a given class of widgets (like the first day of the week in a `DATEEDIT` item type on page 1272).

Presentation styles are defined in a resource file having an extension of `.4st`. The `.4st` file must be distributed with the other runtime files.

Presentation styles are inspired by the *cascading style sheets* (CSS) used in HTML, with the following deviations:

1. The elements using style definitions are AUI tree elements; CSS styles apply to HTML elements.
2. To specify a style for an AUI tree element, you must use the "style" attribute; HTML/CSS use the "class" attribute.
3. Inline-style definition is not supported in the AUI tree.
4. Some pseudo selectors, such as "query, are specific to Genero.

**Figure 45: Form without presentation styles (GDC)** on page 1166 shows a desktop application without presentation styles:
Figure 45: Form without presentation styles (GDC)

Figure 46: Form using presentation styles (GDC) on page 1166 shows a desktop application with presentation styles:
Syntax of presentation styles file (.4st)
A .4st presentation styles file is an XML file defining style attributes to be applied by front-ends.

Syntax (.4st)

```
<StyleList>
  <Style name="style-identifier">
    <StyleAttribute name="attribute-name" value="attribute-value" />
    [...]
  </Style>
  [...]
</StyleList>
```

where style-identifier is:

```
[ element-type | .style-name | :pseudo-selector ]
```

and where pseudo-selector is one of:

```
  | focus | inactive | active |
  | query | display  | input  |
  | even   | odd      |
  | message | error   | summaryLine |
```

1. element-type is the type of the AUI tree element, such as Edit, Window. See also Element types on page 1173.
2. style-name is an explicit style name, that can be referenced in STYLE attributes of form items.
3. pseudo-selector indicates in what context the style should apply. See also Pseudo selectors on page 1168.
4. attribute-name defines the name of the style attribute.
5. attribute-value defines the value to be assigned to attribute-name.

Syntax of attribute values

Presentation style attribute values are always specified as strings, for example:

```
<StyleAttribute name="fontFamily" value="Serif" />
```

Numeric values must be specified in quotes:

```
<StyleAttribute name="completionTimeout" value="60" />
```

Boolean values must be specified with the values "yes" or "no":

```
<StyleAttribute name="forceDefaultSettings" value="yes" />
```

Note: For backward compatibility, the values 0/1 and true/false are supported by some front-ends. Use exclusively the yes/no values.

Using presentation styles

Use presentation styles to centralize the decoration of your user interface.

Defining a style

Styles can be defined to be global (for all elements), for an element in general, or for specific types of an element.

The style is identified by the name attribute, that can be a combination of element type, style name and pseudo selector, or the star character. See Syntax of presentation styles file (.4st) on page 1167 for a complete description of the presentation style definition syntax.
In the definition of a style, the *name* attribute is used as a selector to apply style attributes to graphical elements.

You can define a style as global or specific to a class of graphical object:

- A style identified by a star (*) is a global style that is automatically applied to all elements:
  
  `<Style name="*">`

- A style identified by an *element-type* is a global style that is automatically applied to all objects of this type:
  
  `<Style name="ComboBox">`

- A style identified by a *style-name* is a specific style that can be applied to any element types using that style name in a *STYLE* attribute:
  
  `<Style name=".important">`

- A style identified by an *element-type* followed by a dot and a *style-name* is a specific style that will only be applied to elements of the given type and using the style name in a *STYLE* attribute:
  
  `<Style name="Window.main">`

- A style identified by an *element-type* followed by a colon and a *pseudo-selector* is a style that will only be applied to elements of the given type, if the condition defined by the pseudo-selector is satisfied:
  
  `<Style name="Edit:focus">`

- A style identified by an *element-type* followed by a dot and a *style-name*, and a colon with a pseudo-selector, is a specific style that will only be applied to elements of the given type, using the style name in a *STYLE* attribute, if the condition defined by the pseudo-selector is satisfied:
  
  `<Style name="Edit.important:focus">`

- It is possible to combine pseudo-selectors:
  
  `<Style name="Edit:query:focus">`

**Pseudo selectors**

Pseudo selectors can be used to apply style only when some conditions are fulfilled.

Pseudo selectors are preceded by a colon and can be combined:

`<Style name="Table:even:input">`
`<Style name="Edit:focus">`
`<Style name="Edit.important:focus">`

When combining several pseudo selectors, the style will be applied if all pseudo selector conditions are fulfilled.

**Note:** Depending on the type of the front-end, some pseudo selectors are meaningless, or unsupported. See the table below to check which pseudo selectors are supported on your front-end platform.

Pseudo selectors have different priorities; the style with the most important pseudo selector will be used when several styles match.

### Table 310: Pseudo selectors for presentation styles

<table>
<thead>
<tr>
<th>Priority</th>
<th>Pseudo selectors</th>
<th>Condition</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>focus</td>
<td>The widget has the focus</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>query</td>
<td>The widget is in construct mode</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Pseudo selectors also define the priority of your styles. A more generic style will be used when the pseudo-selector has a higher priority.

For instance: you want all important edits to have red text, but you want the current field to be displayed in blue:

```xml
<Style name="Edit.important" />
<Style name=":focus" />
```

The style ":focus" is more generic than "Edit.important"; therefore, it will be used for the focused item, as the pseudo selector is more precise.

**Related concepts**  
Summary lines in tables on page 1866  
Table views can display a summary line, to show aggregate values for columns.

**Related reference**  
Message style attributes on page 1199  
Message presentation style attributes apply to an ERROR or MESSAGE instruction.

**Using a style**  
To apply a specific style, set the style-name in the style attribute of the node representing the graphical element in the abstract user interface tree.

There are different ways to set the style attribute of an element:

- As a form element attribute, with a STYLE attribute in the form specification file.
- In the ATTRIBUTES clause of instructions such as OPEN WINDOW, MESSAGE, ERROR.
- Dynamically by a program, using the ui.Form.setElementStyle() method.

For example, to define a style in a form file for an input field:

```java
EDIT f001 = customer.fname, STYLE = "info";
```

**Note:** The string used to define the STYLE attribute must be a style-name only, it must not contain the element-type that is typically used to define the style in a .4st file (as CheckBox.important for example)
Related concepts

The **ATTRIBUTES** section describes properties of grid-based layout elements used in the form.

**Order of precedence**

Style definitions are applied following and order of precedence.

If different styles can be applied to an element, the following priority is used to determine the style definition to be applied:

1. `element-type.style-name: pseudo-selector`
2. `.style-name: pseudo-selector`
3. `element-type.style-name`
4. `element-type: pseudo-selector`
5. `:pseudo-selector`
6. `.style-name`
7. `element-type`
8. `*`

**Note:** The precedence rules to apply styles can be specific to a front-end type. As a general rule, Genero presentation styles precedence rules are similar to HTML/CSS precedence rules.

For example, consider an **Edit** element with the style attribute set to 'mandatory':

```plaintext
EDIT f1 = FORMONLY.cust_name, STYLE="mandatory"
```

With the following style definitions (`mystyles.4st`):

```xml
<?xml version="1.0" encoding="ANSI_X3.4-1968"?>
<StyleList>
  <Style name="Edit.mandatory:focus">
    <StyleAttribute name="backgroundColor" value="yellow" />
  </Style>
  <Style name=".mandatory:focus">
    <StyleAttribute name="backgroundColor" value="blue" />
  </Style>
  <Style name="Edit.mandatory">
    <StyleAttribute name="backgroundColor" value="green" />
  </Style>
  <Style name="Edit:focus">
    <StyleAttribute name="backgroundColor" value="red" />
  </Style>
  <Style name=":focus">
    <StyleAttribute name="backgroundColor" value="cyan" />
  </Style>
  <Style name=".mandatory">
    <StyleAttribute name="backgroundColor" value="magenta" />
  </Style>
  <Style name="*">
    <StyleAttribute name="backgroundColor" value="orange" />
  </Style>
</StyleList>
```

The style definitions are scanned in the following order:

1. `Edit.mandatory:focus`
2. `.mandatory:focus`
3. `Edit.mandatory`
4. `Edit:focus`
5. `:focus`
6. mandatory
7. Edit
8. *

If the Edit field f1 has the focus, with the mystyles.4st definition file, the field background color will be yellow. If the Edit field f1 does not have the focus, the field background color will be green.

**Combining styles**

You can combine several styles, by using the space character as a separator in the **STYLE** attribute.

In the following example, the **STYLE** attribute defines three different style names:

```
EDIT f001 = customer.fname, STYLE = "info highlight mandatory"
```

When several styles are combined, the same presentation attribute might be defined by different styles. In this case, the first style listed that defines the attribute takes precedence over the other styles.

For example, if the **textColor** presentation attribute is defined as follows by the **info**, **highlight** and **mandatory** styles:

- info style does not define **textColor**.
- highlight style defines **textColor** as blue.
- mandatory style defines **textColor** as red.

The widgets having a style set to "info highlight mandatory" will get a blue text color, because highlight is listed before mandatory.

**Related concepts**

- **STYLE attribute** on page 1395

The **STYLE** attribute specifies a presentation style for a form element.

**Style attribute inheritance**

A style attribute may be inherited by the descendants of a given node in the abstract user interface tree.

The style inheritance is implicitly defined by the attribute. Common font and color related style attributes are typically inherited.

For example, when defining a style using the **fontFamily** in a window or group container, all the children of this container will get the same font family. However, most style attributes (such as "border" for the Window element type) are specific to a type of element, and are not inherited.

Style attribute inheritance can be overwritten by defining the same attribute for the type of elements that are used inside the parent container. In the example, the .4st style file defines a text color for groups and all its descendant nodes. The same attribute is then redefined explicitly for labels and edit nodes:

```
<StyleList>
  <Style name="Group">
    <StyleAttribute name="textColor" value="red" />
  </Style>
  <Style name="Label">
    <StyleAttribute name="textColor" value="windowText" />
  </Style>
  <Style name="Edit">
    <StyleAttribute name="textColor" value="windowText" />
  </Style>
</StyleList>
```

For more details, see **Style attributes reference** on page 1182.
Presentation styles in the AUI tree
Where to find presentation styles definitions in the AUI Tree?
Presentation styles are loaded in the abstract user interface tree, under the UserInterface node, in a StyleList node following the presentation style syntax.
The StyleList node holds a list of Style nodes that define a set of attribute values. Attribute values are defined in StyleAttribute nodes, with a name and a value attribute.

Related concepts
The abstract user interface tree on page 1106
The abstract user interface tree is the XML representation of the application forms displayed to the end user.
Syntax of presentation styles file (.4st) on page 1167
A .4st presentation styles file is an XML file defining style attributes to be applied by front-ends.

Loading presentation styles
Presentation styles are defined in an XML file with a .4st extension. In order to load the presentation styles, the runtime system needs to locate the appropriate style file.
By default, the runtime system searches for a file named default.4st in several directories, as described in the FGLRESOURCEPATH reference topic. If the file is not found using the FGLRESOURCEPATH/DBPATH environment variables, default presentation styles are loaded from the $FGLDIR/lib/default.4st file.
Overwrite the default search by loading a specific presentation style file with the ui.Interface.loadStyles() method:

```
MAIN
    CALL ui.Interface.loadStyles("mystyles")
    ...
END MAIN
```
This method accepts an absolute path with the .4st extension, or a simple file name without the .4st extension. If you give a simple file name, for example "mystyles", the runtime system searches for the mystyles.4st file in the current directory or in the application directory. If the file does not exist, it searches in the directories defined by the FGLRESOURCEPATH environment variable. If FGLRESOURCEPATH is not defined, it searches in the directories defined by the DBPATH environment variable.
The presentation styles must be defined in a unique .4st file. When loading a styles file with the ui.Interface.loadStyles() method, current styles created from the default file or from a prior load will be replaced. The styles will not be combined when loading several files.
The default styles file located in $FGLDIR/lib should not be modified directly: your changes would be lost if you upgrade the product. Make a copy of the original file into the program directory of your application, then modify the copied file.

Combining TTY and style attributes
TTY attributes can be specific to a form element or can be inherited by an element from a parent node (such as the form or window).
Specific element TTY attributes are directly set in the element node in the AUI tree; they can, for example, be defined with the COLOR attribute of form items. Inherited TTY attributes are taken from the parent nodes of the leaf element to be displayed. For example, when a form is displayed with DISPLAY FORM followed by an ATTRIBUTE clause containing TTY color, font option and/or video attributes, all static labels will be displayed with the TTY attributes of the form. Note that the form elements controlled by interactive instructions (i.e. form fields) will explicitly get the TTY attributes defined by the ATTRIBUTE clause of OPEN WINDOW, OPEN FORM, DISPLAY TO / BY NAME or the current dialog statement, and must be considered specific TTY attributes for the element.
Specific TTY attributes defined for a form element have a higher priority than style attributes, while inherited TTY attributes (set on one of the parent elements) have a lower priority than style attributes defined for the element.
To illustrate this rule, imagine a form defining two static labels and two fields, with all items using the mystyle presentation style, and one of the labels and fields defining a specific TTY attribute with COLOR=BLUE:

```
LABEL lab01: TEXT="Field 1:", COLOR = BLUE, STYLE = "mystyle";
EDIT fld01 = FORMONLY.field01, COLOR = BLUE, STYLE = "mystyle";
LABEL lab02: TEXT="Field 2:", STYLE = "mystyle";
EDIT fld02 = FORMONLY.field02, STYLE = "mystyle";
```

The program displays the form (or window) with an ATTRIBUTES clause using a red color, and the fields are used by an INPUT dialog, with no ATTRIBUTES clause, so the default TTY attributes are gotten from the OPEN FORM instruction:

```
OPEN FORM f FROM "ttyform"
DISPLAY FORM f ATTRIBUTES(RED)
INPUT BY NAME field01, field02 WITHOUT DEFAULTS
```

The .4st styles file defines the mystyle attributes as follows:

```
<StyleList>
  <Style name="Edit.mystyle">
    <StyleAttribute name="textColor" value="green" />
  </Style>
  <Style name="Label.mystyle">
    <StyleAttribute name="textColor" value="magenta" />
  </Style>
</StyleList>
```

The text in the form field fld01 is displayed in blue (from the specific COLOR attribute), while fld02 is displayed in red (the TTY attribute of the form, the style Edit.mystyle being ignored).

Since labels are not used by the interactive instructions, lab01 is displayed in blue (from the specific COLOR attribute), while lab02 is displayed in magenta (from the style Label.mystyle, the form TTY attribute red being ignored).

**Element types**

Which AUI tree elements can get a style?

Styles may apply to any graphical elements of the user interface, such as Button, Edit, ComboBox, ButtonEdit, Table, Window.

The name of the element when used in a style file is case sensitive (use CheckBox, not checkbox).

For example, the following style definition uses the "Window" element type in the style name:

```
<Style name="Window.dialog">
  <StyleAttribute name="position" value="center" />
</Style>
```

The supported element types is defined by the style attributes, for more details, see Style attributes reference on page 1182.

**Predefined attribute values**

This section describes the values that must be used for some style attributes.

**Colors**

When providing a value for style attributes that define color, you can specify a generic color name or its RGB value.

This section describes how to specify a value for style attributes defining colors, such as textColor.

**Syntax**

```
\ generic-color \ #rrggbb \n```
1. *generic-color* is any of the predefined colors supported by the language.
2. *#rrggbb* is a numerical color defined by a red/green/blue specification.

**Usage**

In most cases it is not possible to know what a potential end-user might expect regarding the font family. Therefore, avoid using explicit font families in your application and use only the `fontWeight/fontStyle/fontSize` properties. A specific font family is used only if the client cannot determine a proper default font family for the desired platform.

The language defines a set of generic color names, interpreted by the front-end depending on the graphical capability of the workstation.

**RGB notation**

When a predefined color is not what you require, you can specify the exact color with the RGB notation, starting with a # hash character: `<Style name="Edit.mandatory"> <StyleAttribute name="textColor" value="#50AEFF" /> </Style>`

Each value of the RGB color specification must be provided in hexadecimal, in the range [00-FF].

**Generic color names**

Use generic color names, to keep your style definitions portable across several front-end types.

**Note:** The exact rendered color depends on front-end type.

**Table 311: Generic color names**

<table>
<thead>
<tr>
<th>Generic color name</th>
<th>Visual result (1)</th>
<th>RGB value</th>
</tr>
</thead>
<tbody>
<tr>
<td>black</td>
<td><img src="image" alt="Black" /></td>
<td>#000000</td>
</tr>
<tr>
<td>blue</td>
<td><img src="image" alt="Blue" /></td>
<td>#0000FF</td>
</tr>
<tr>
<td>cyan</td>
<td><img src="image" alt="Cyan" /></td>
<td>#00FFFF</td>
</tr>
<tr>
<td>darkBlue</td>
<td><img src="image" alt="Dark Blue" /></td>
<td>#00008B</td>
</tr>
<tr>
<td>darkCyan</td>
<td><img src="image" alt="Dark Cyan" /></td>
<td>#008B8B</td>
</tr>
<tr>
<td>darkGray</td>
<td><img src="image" alt="Dark Gray" /></td>
<td>#A9A9A9</td>
</tr>
<tr>
<td>darkGreen</td>
<td><img src="image" alt="Dark Green" /></td>
<td>#006400</td>
</tr>
<tr>
<td>Generic color name</td>
<td>Visual result (1)</td>
<td>RGB value</td>
</tr>
<tr>
<td>--------------------</td>
<td>------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>darkMagenta</td>
<td><img src="image_url1" alt="Visual result" /></td>
<td>#8B008B</td>
</tr>
<tr>
<td>darkOlive</td>
<td><img src="image_url2" alt="Visual result" /></td>
<td>#505000</td>
</tr>
<tr>
<td>darkOrange</td>
<td><img src="image_url3" alt="Visual result" /></td>
<td>#FF8C00</td>
</tr>
<tr>
<td>darkRed</td>
<td><img src="image_url4" alt="Visual result" /></td>
<td>#8B0000</td>
</tr>
<tr>
<td>darkTeal</td>
<td><img src="image_url5" alt="Visual result" /></td>
<td>#005050</td>
</tr>
<tr>
<td>darkYellow</td>
<td><img src="image_url6" alt="Visual result" /></td>
<td>#AAAA00</td>
</tr>
<tr>
<td>gray</td>
<td><img src="image_url7" alt="Visual result" /></td>
<td>#808080</td>
</tr>
<tr>
<td>green</td>
<td><img src="image_url8" alt="Visual result" /></td>
<td>#008000</td>
</tr>
<tr>
<td>lightBlue</td>
<td><img src="image_url9" alt="Visual result" /></td>
<td>#ADD8E6</td>
</tr>
<tr>
<td>lightCyan</td>
<td><img src="image_url10" alt="Visual result" /></td>
<td>#E0FFFF</td>
</tr>
<tr>
<td>lightGray</td>
<td><img src="image_url11" alt="Visual result" /></td>
<td>#D3D3D3</td>
</tr>
<tr>
<td>lightGreen</td>
<td><img src="image_url12" alt="Visual result" /></td>
<td>#90EE90</td>
</tr>
<tr>
<td>lightMagenta</td>
<td><img src="image_url13" alt="Visual result" /></td>
<td>#FFE0FF</td>
</tr>
<tr>
<td>lightOlive</td>
<td><img src="image_url14" alt="Visual result" /></td>
<td>#AAAA44</td>
</tr>
<tr>
<td>lightOrange</td>
<td><img src="image_url15" alt="Visual result" /></td>
<td>#FFCC00</td>
</tr>
</tbody>
</table>
### Generic color name

<table>
<thead>
<tr>
<th>Generic color name</th>
<th>Visual result</th>
<th>RGB value</th>
</tr>
</thead>
<tbody>
<tr>
<td>lightRed</td>
<td><img src="image" alt="lightRed" /></td>
<td>#FF8080</td>
</tr>
<tr>
<td>lightTeal</td>
<td><img src="image" alt="lightTeal" /></td>
<td>#33CCCC</td>
</tr>
<tr>
<td>lightYellow</td>
<td><img src="image" alt="lightYellow" /></td>
<td>#FFFFFFE0</td>
</tr>
<tr>
<td>magenta</td>
<td><img src="image" alt="magenta" /></td>
<td>#FF00FF</td>
</tr>
<tr>
<td>olive</td>
<td><img src="image" alt="olive" /></td>
<td>#808000</td>
</tr>
<tr>
<td>orange</td>
<td><img src="image" alt="orange" /></td>
<td>#FFA500</td>
</tr>
<tr>
<td>red</td>
<td><img src="image" alt="red" /></td>
<td>#FF0000</td>
</tr>
<tr>
<td>teal</td>
<td><img src="image" alt="teal" /></td>
<td>#008080</td>
</tr>
<tr>
<td>white</td>
<td><img src="image" alt="white" /></td>
<td>#FFFFFF</td>
</tr>
<tr>
<td>yellow</td>
<td><img src="image" alt="yellow" /></td>
<td>#FFFF00</td>
</tr>
</tbody>
</table>

### Example

```xml
<StyleAttribute name="textColor" value="blue" />
<StyleAttribute name="textColor" value="#00FF45" />
```

### Related concepts

**Font families** on page 1178

Use the `fontFamily` style attribute to define a generic or specific font family.

**GDC System Colors**

System color names can be used to get a color from the current theme of the workstation windowing system where the GDC executes.

**Important:** While system color names have meaning for both the GDC and GBC front-end clients, the mapping of the system color name to its meaning in this topic is relevant for GDC applications only.
### Table 312: System color names

<table>
<thead>
<tr>
<th>System color name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>appWorkSpace</td>
<td>Background color of multiple document interface</td>
</tr>
<tr>
<td>background</td>
<td>Desktop background</td>
</tr>
<tr>
<td>buttonFace</td>
<td>Face color for three-dimensional display elements.</td>
</tr>
<tr>
<td>buttonText</td>
<td>Text on push buttons.</td>
</tr>
<tr>
<td>grayText</td>
<td>Grayed (disabled) text.</td>
</tr>
<tr>
<td>highLight</td>
<td>Item(s) selected in a control</td>
</tr>
<tr>
<td>highLightText</td>
<td>Text of item(s) selected in a control</td>
</tr>
<tr>
<td>infoBackground</td>
<td>Background color for tooltip controls.</td>
</tr>
<tr>
<td>infoText</td>
<td>Text color for tooltip controls.</td>
</tr>
<tr>
<td>systemAlternateBackground</td>
<td>Background color of the alternate row in listviews</td>
</tr>
<tr>
<td>window</td>
<td>Window background.</td>
</tr>
<tr>
<td>windowText</td>
<td>Text in windows.</td>
</tr>
</tbody>
</table>

**GBC System Colors**

System color names can be used to get a color from the current GBC theme.

For information about the colors used for the current GBC theme, see the *Genero Browser Client User Guide*.

**Important:** While system color names have meaning for both the GDC and GBC front-end clients, the mapping of the system color name to its meaning in this topic is relevant for GBC applications only.

### Table 313: System color names

<table>
<thead>
<tr>
<th>System color name</th>
<th>Meaning</th>
<th>Corresponding GBC Theme variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>appWorkSpace</td>
<td>Same as window.</td>
<td>theme-secondary-background-color</td>
</tr>
<tr>
<td>background</td>
<td>Background color.</td>
<td>theme-secondary-alternative-background-color</td>
</tr>
<tr>
<td>buttonFace</td>
<td>Button background color.</td>
<td>theme-primary-background-color</td>
</tr>
<tr>
<td>buttonText</td>
<td>Text color of buttons.</td>
<td>theme-primary-color</td>
</tr>
<tr>
<td>grayText</td>
<td>Grayed (disabled) text.</td>
<td>theme-disabled-background-color</td>
</tr>
<tr>
<td>highLight</td>
<td>Background color of row(s) selected in a array.</td>
<td>theme-primary-emphasis-background-color</td>
</tr>
<tr>
<td>highLightText</td>
<td>Text color of row(s) selected in a array.</td>
<td>theme-primary-color</td>
</tr>
<tr>
<td>infoBackground</td>
<td>N/A (same as window)</td>
<td>theme-secondary-background-color</td>
</tr>
<tr>
<td>infoText</td>
<td>N/A (same as windowText)</td>
<td>theme-secondary-color</td>
</tr>
<tr>
<td>systemAlternateBackground</td>
<td>Background color of the alternate row in array.</td>
<td>theme-primary-faded-background-color</td>
</tr>
<tr>
<td>window</td>
<td>Window background color.</td>
<td>theme-secondary-background-color</td>
</tr>
<tr>
<td>windowText</td>
<td>Text in windows.</td>
<td>theme-secondary-color</td>
</tr>
</tbody>
</table>
Fonts
A graphical application defaults to using the front-end platform theme. The front-end tries to determine the default font for the application screens.

Font families
Use the fontFamily style attribute to define a generic or specific font family.

This section describes the possible values of the fontFamily style attribute.

Syntax

\[
\text{font-family} \ [\,*\,*]\n\]

1. fontFamily defines a generic or a native font family.

Usage
Sets of generic font families are supported. These are interpreted by the front-end depending on the graphical capability of the platform.

If the fontFamily is not a generic font family, it is interpreted as a native font family, which identifies a local font supported by the front-end. Usually, it is one of the fonts installed on the platform operating system. See front-end documentation for a list of supported native fonts.

Tip: It is recommended to use a generic font family instead of a native font family.

A native font family is used only if the front-end cannot determine a proper default font family for the desired platform.

Important: A font family containing white-spaces must be single quoted. In the XML definition of the style, this leads to a single quoted string that is, in turn, enclosed in double quotes:

\[
<\text{StyleAttribute} \text{ name="fontFamily" value="'Courier New'"} />\n\]

When specifying a comma-separated list of font families, the front-end uses the best matching font available on the platform. You can mix generic and native font families:

\[
<\text{StyleAttribute} \text{ name="fontFamily" value="'Times New Roman',Times,serif"} />\n\]

Table 314: Generic font families to front-end platform fonts

<table>
<thead>
<tr>
<th>Generic font family name</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>serif</td>
<td>Times</td>
<td>serif (CSS)</td>
<td>Serif</td>
<td>Times New Roman</td>
</tr>
<tr>
<td>sans-serif</td>
<td>Arial</td>
<td>sans-serif (CSS)</td>
<td>Sans-Serif</td>
<td>Helvetica Neue</td>
</tr>
<tr>
<td>cursive</td>
<td>Comic Sans Ms</td>
<td>cursive (CSS)</td>
<td>N/A (keeps default font)</td>
<td>Marker Felt</td>
</tr>
<tr>
<td>fantasy</td>
<td>Algerian</td>
<td>fantasy (CSS)</td>
<td>N/A (keeps default font)</td>
<td>Papyrus</td>
</tr>
<tr>
<td>monospace</td>
<td>Courier New</td>
<td>monospace (CSS)</td>
<td>Monospace</td>
<td>Courier</td>
</tr>
</tbody>
</table>

Note:
- The GBC front-end uses the font family as fontFamily property in a CSS style. For more details, see CSS generic-font-families
• The GMI front-end tries to find a font family in the available fonts of the application (this means the iOS built-in fonts and any application-specific fonts) which matches the `fontFamily` given in the styles. If none is found, the fallback is "Helvetica Neue".

• The GMA front-end maps generic font family names to Android™ generic font names (Serif, Monospace), these are then mapped to real font names. The real font name depends on the Android™ brand. For example sans-serif is usually implemented with the "Roboto" font.

Example

```xml
<StyleAttribute name="fontFamily" value="sans-serif" />
<StyleAttribute name="fontFamily" value=""Courier New"" />
<StyleAttribute name="fontFamily" value=""Times New Roman",Times,serif" />
```

Related concepts

Syntax of presentation styles file (.4st) on page 1167

A .4st presentation styles file is an XML file defining style attributes to be applied by front-ends.

Font sizes

Use the `fontSize` style attribute to influence the size of a font.

Syntax

```
<generic-size ↓ pointspt ↓ sizeem ↓
```

1. `generic-size` is one of the generic font size names (such as 'small' or 'xx-large') listed in Table 315: Generic font sizes on page 1179.
2. `points` defines an absolute size in points. Specify a number followed immediately by `pt`, for example, 3pt.
3. `size` defines relative size. Specify a number followed immediately by `em`, for example, 3em.

Usage

Specify either a generic font size, an absolute size in points with the "pt" unit, or a relative size with the "em" unit.

Absolute sizes (using the "pt" suffix) define a font size in physical points. Physical points are much like pixels, in that they are fixed-size units and cannot scale in size. For example, on HTML pages using CSS styles, one point is equal to 1/72 of an inch.

Relative sizes (using the "em" suffix) define a font size in a scalable size unit that adapts to the front-end platform, where one "em" unit results in the same size as the size of the default font on the platform. For example, if the size of the platform default font is 16 points, 1em = 16pt, 2em = 32pt, etc.

Generic font sizes are interpreted by the front-end depending on the graphical capability of the platform.

Note: Use generic font sizes such as medium, large, small, or sizes relative to the user-chosen font (using em units), rather than absolute point values. In an HTML browser you can choose two fonts (proportional/fixed), and a well-designed document does not use more than 2 fonts. This is also valid for applications.

Table 315: Generic font sizes

<table>
<thead>
<tr>
<th>Generic font size name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>xx-small</td>
<td>Tiny font size</td>
</tr>
<tr>
<td>x-small</td>
<td>Extra-small font size</td>
</tr>
<tr>
<td>small</td>
<td>Small font size</td>
</tr>
<tr>
<td>medium</td>
<td>Medium font size</td>
</tr>
<tr>
<td>large</td>
<td>Large font size</td>
</tr>
</tbody>
</table>
## Generic font size name

<table>
<thead>
<tr>
<th>Generic font size name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>x-large</td>
<td>Extra-large font size</td>
</tr>
<tr>
<td>xx-large</td>
<td>Huge font size</td>
</tr>
</tbody>
</table>

You can also specify an absolute font size, by giving a numeric value followed by the units such as \texttt{pt} or \texttt{em}:

**Example**

```xml
<StyleAttribute name="fontSize" value="medium" />
<StyleAttribute name="fontSize" value="xx-large" />
<StyleAttribute name="fontSize" value="12pt" />
<StyleAttribute name="fontSize" value="1em" />
```

### Related concepts

**Font families** on page 1178

Use the `fontFamily` style attribute to define a generic or specific font family.

**Font styles**

Use the `fontStyle` style attribute to define the style of a font.

### Syntax

```xml
<StyleAttribute name="fontStyle" value="italic" />
```

### Usage

The style of a font can be specified with a generic name, that is interpreted by the front-end depending on the graphical capabilities of the platform. For example, on "Android™" devices, `italic` and `oblique` result in the same font aspect.

#### Table 316: Generic font style

<table>
<thead>
<tr>
<th>Generic font style name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>italic</td>
<td>Specifies an italic font style, using a typeface that slants slightly to the right. Uses a different glyph as the roman style.</td>
</tr>
<tr>
<td>oblique</td>
<td>Specifies an oblique font style. This style is similar to italic, except that it uses the same glyphs as the roman type, but distorted.</td>
</tr>
<tr>
<td>roman</td>
<td>Specifies a roman font style. This is the typical default font style in Latin-script typography.</td>
</tr>
</tbody>
</table>

**Example**

```xml
<StyleAttribute name="fontStyle" value="italic" />
```

**Font weights**

Use the `fontWeight` style attribute to define the aspect of a font.

### Syntax

```xml
<StyleAttribute name="fontWeight" value="black" />
<StyleAttribute name="fontWeight" value="bold" />
<StyleAttribute name="fontWeight" value="book" />
<StyleAttribute name="fontWeight" value="condensed" />
<StyleAttribute name="fontWeight" value="condensedbold" />
```
Usage
The availability of the weight depends on the chosen font family. For example, if the font family is defined as AmericanTypewriter, and the front-end platform supports the following set of font names (for this font family): AmericanTypewriter, AmericanTypewriter-Light, AmericanTypewriter-Bold, AmericanTypewriter-CondensedLight, AmericanTypewriter-CondensedBold, AmericanTypewriter-Condensed, you can only use the condensed, light and bold font weights.

Before using a font weight, make sure that the target platform supports the value. For example, on "Android™" devices, only normal and bold are supported.

Example

```
<StyleAttribute name="fontWeight" value="bold" />
```

Statusbar types
Possible values for Window status bar type.

This section describes how to specify a value for the `Window.statusBarType` style attribute.

Syntax

```
{statusbar-type}
```

1. `statusbar-type` is a predefined status bar type name.

Usage
The `statusBarType` style attribute can get one of the values listed in the following table, to customize the rendering of error and message texts.

Important: This style attribute is mainly used for desktop application using the GDC front-end.

Table 317: Possible status bar types for the `statusBarType` attribute

<table>
<thead>
<tr>
<th>Value</th>
<th>Screenshot</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td><img src="default.png" alt="Screenshot" /></td>
</tr>
<tr>
<td>lines1</td>
<td><img src="lines1.png" alt="Screenshot" /></td>
</tr>
<tr>
<td>lines2</td>
<td><img src="lines2.png" alt="Screenshot" /></td>
</tr>
<tr>
<td>Value</td>
<td>Screenshot</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>lines3</td>
<td><img src="image" alt="Screenshot lines3" /></td>
</tr>
<tr>
<td>lines4</td>
<td><img src="image" alt="Screenshot lines4" /></td>
</tr>
<tr>
<td>lines5</td>
<td><img src="image" alt="Screenshot lines5" /></td>
</tr>
<tr>
<td>lines6</td>
<td><img src="image" alt="Screenshot lines6" /></td>
</tr>
<tr>
<td>panels1</td>
<td><img src="image" alt="Screenshot panels1" /></td>
</tr>
<tr>
<td>panels2</td>
<td><img src="image" alt="Screenshot panels2" /></td>
</tr>
<tr>
<td>panels3</td>
<td><img src="image" alt="Screenshot panels3" /></td>
</tr>
<tr>
<td>panels4</td>
<td><img src="image" alt="Screenshot panels4" /></td>
</tr>
<tr>
<td>panels5</td>
<td><img src="image" alt="Screenshot panels5" /></td>
</tr>
<tr>
<td>panels6</td>
<td><img src="image" alt="Screenshot panels6" /></td>
</tr>
<tr>
<td>panels7</td>
<td><img src="image" alt="Screenshot panels7" /></td>
</tr>
<tr>
<td>none</td>
<td><img src="image" alt="Screenshot none" /></td>
</tr>
</tbody>
</table>

**Example**

```xml
<StyleAttribute name="Windows.statusBarType" value="panels2" />
```

**Style attributes reference**

A presentation style attribute may be a common attribute that can be applied to any graphical element. Most presentation style attributes apply only to a specific graphical element.

**Style attributes common to all elements**

Common presentation style attributes apply to any graphical element, such as windows, layout containers, or form items.

**Important:** Common style attribute apply to basic layout elements such as containers (Group) and form widgets (Label, Button, Edit, CheckBox). Depending on the front-end platform, common style attributes typically do not apply to advanced graphical elements such as TopMenu or ToolBar, especially when such widgets can be configured with the user interface theme of the front-end platform. Consider using common style attribute only for elements inside the form layout.
Table 318: Common style attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>backgroundColor</td>
<td>Yes (see note)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes (see note)</td>
</tr>
<tr>
<td>Defines the color to be used to fill the background of the object.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>For possible values, see Colors on page 1173.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Default is no value (default background color of the object, or inherited background color).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Note:</strong> To set the background color of GMI/iOS specific GUI elements like toolbars, tab bars and navigation bars, use the ios*TintColor attributes for Windows.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Note:</strong> The combobox.qtStyle style attribute must be set to &quot;Windows&quot;, to have other style attributes such as backgroundColor take effect.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>border</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Defines the border for the widget.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>If value is &quot;none&quot;, it removes the border.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Default is no value (the widget gets its default appearance).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>This attribute applies especially to the IMAGE form item type. Depending on the front-end, the border of other widgets may or may not be affected by this style attribute.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fontFamily</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Defines the name of the font.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>For possible values, see Font families on page 1178.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Default is no value (default object font or inherited font).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fontSize</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Defines the size of the characters.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>For possible values, see Font sizes on page 1179.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Default is no value (default object font or inherited font).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fontStyle</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Defines the style of characters.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>For possible values, see Font styles on page 1180.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Default is no value (default object font or inherited font).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fontWeight</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Defines the weight of the characters.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Possible values for font weights depend on the front-end native font names, see Font weights on page 1180 for details.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Default is no value (default object font or inherited font).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attribute</td>
<td>GDC</td>
<td>GBC</td>
<td>GMA</td>
<td>GMI</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>localAccelerators</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>For form items using shortcuts, defines how the widget must behave regarding keyboard accelerators. Values can be &quot;yes&quot; (default) or &quot;no&quot;. If value is &quot;yes&quot; (default), the local accelerators have higher priority. Ex: &quot;HOME&quot; key moves the cursor to the first position. If value is &quot;no&quot;, the application accelerators have higher priority. Ex: &quot;HOME&quot; key selects the first row of the current array. The following keys are managed &quot;locally&quot; if the attribute is defined with &quot;yes&quot;.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEXTEDIT: left, right, up, down, (control +)home, (control+)end, (control +)backspace, (control+)delete EDIT, BUTTONEDIT, DATEEDIT, etc: left, right, home, end, (control+)backspace, (control+)delete TABLE, TREE: (control+)left, (control+)right</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>showAcceleratorInToolTip</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Defines if the accelerator key(s) for an action is shown in the tooltip of the corresponding action view (Button, Toolbar Item, and so on.) Values can be &quot;yes&quot; or &quot;no&quot; (default). If value is &quot;yes&quot; the tooltip shows the accelerator key(s) after the action name, between brackets. By default, the tooltip only shows the action name.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>textColor</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes (see note)</td>
</tr>
<tr>
<td>Defines the color to be used to paint the text of the object. For possible values, see Colors on page 1173. Default is no value (default object color or inherited color). Note: In GMI, textColor affects the widgets they are defined on, not the labels in the form used to display the widgets. It is also used to set the tint of checkbox, radio group (horizontal), and spin edit.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>textDecoration</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Defines the decoration for the text. Values can be &quot;overline&quot;, &quot;underline&quot; or &quot;line-through&quot;. Default is no value (default object font or inherited font).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Related concepts

**STYLE attribute** on page 1395
The **STYLE** attribute specifies a presentation style for a form element.

**Button style attributes**

Button presentation style attributes apply to **BUTTON** elements.

**Note:** This topic lists presentation style attributes for a specific class of form element, **common presentation style attributes** can also be used for this type of element.

### Table 319: Button style attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>alignment</strong></td>
<td>No (see note)</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
| Defines the position of the image and/or text inside a button, when the button is bigger than the content.
| The value can be a combination of a vertical and horizontal alignment hints, separated by a space.
| The value can also be "center", which is equivalent to "verticalCenter horizontalCenter".
| Vertical alignment hints:
| • "top": anchor to the top edge.
| • "verticalCenter": center in middle.
| • "bottom": anchor to the bottom edge.
| Horizontal alignment hints:
| • "left": anchor to the left edge.
| • "horizontalCenter": center in middle.
| • "right": anchor to the right edge.
| The default is "center".

**Important:** With the GDC front-end, the **alignment** style attribute is not supported for **BUTTON** elements. However, the content is centered, if there is only a text or image to be displayed. When both text and image are present, these are left aligned and vertically centered.

<table>
<thead>
<tr>
<th><strong>buttonType</strong></th>
<th>Yes</th>
<th>Yes</th>
<th>No</th>
<th>No</th>
</tr>
</thead>
</table>
| Defines the rendering of a button.
| Values can be:
| • "normal" (default): The button is rendered as a regular push button.
| • "link": the button is rendered as an HTML hyper-link. In contrast to the label hyper-link support, clicking on a "link" button does not start the default browser, but triggers the corresponding action, like a normal button.
| • "commandLink": the button is rendered as a "Command Link" button on Microsoft® Windows® Vista and Windows® 7.

**Important:** The **buttonType** Button style attribute cannot be changed dynamically, once the widget has been displayed.
<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>scaleIcon</td>
<td>Yes (see note)</td>
<td>Yes (see note)</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Defines the scaling behavior of the associated icon, if the source image size is bigger than the place reserved for it in the widget.

**Note:** On GDC and GBC, if the `scaleIcon` attribute is undefined, the behavior depends on the type of action view: toolbar button icons and action panel button icons are scaled down to match the size of the widget. For other widgets, by default no scaling occurs, as for `scaleIcon="no"`.

Values can be:

- "no": No scaling occurs and the image is taken as-is. It is up to the developer to resize the source image to avoid misalignment. This is the default on GDC/GBC.
- "yes": Images are scaled down following the height of the widget (button or edit field). Setting a big font can result in a big icon. This is the default on GMA/GMI.
- "nnnpx": Images are scaled down based on the specified size. For example, `scaleIcon="128px"` will make every icon a maximum of 128*128 pixels. At least one side equal to 128 pixels, depending if the source image is square or not.

Independently of the `scaleIcon` attribute value, to avoid pixelization or blurring of images, raster images are never upscaled. However, SVG images and TTF icons can be upscaled without any penalty.

If the icon must be enlarged, the image is centered and a transparent border is added to "fill" the empty space. This allows a mix of larger and smaller icons while keeping widget alignment.

If scaling takes place, the aspect ratio of the original image is kept. A non-square source image displays as a non-square scaled icon.

**Related concepts**

**BUTTON item type** on page 1266
Defines a push-button that can trigger an action.

**ButtonEdit style attributes**

ButtonEdit presentation style attributes apply to `BUTTONEDIT` elements.

**Note:** This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.
Table 320: ButtonEdit style attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>scaleIcon</td>
<td>Yes (see note)</td>
<td>Yes (see note)</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Defines the scaling behavior of the associated icon, if the source image size is bigger than the place reserved for it in the widget.

**Note:** On GDC and GBC, if the `scaleIcon` attribute is undefined, the behavior depends on the type of action view: toolbar button icons and action panel button icons are scaled down to match the size of the widget. For other widgets, by default no scaling occurs, as for `scaleIcon="no"`.

Values can be:

- "no": No scaling occurs and the image is taken as-is. It is up to the developer to resize the source image to avoid misalignment. This is the default on GDC/GBC.
- "yes": Images are scaled down following the height of the widget (button or edit field). Setting a big font can result in a big icon. This is the default on GMA/GMI.
- "nnnpx": Images are scaled down based on the specified size. For example, `scaleIcon="128px"` will make every icon a maximum of 128*128 pixels. At least one side equal to 128 pixels, depending if the source image is square or not.

Independently of the `scaleIcon` attribute value, to avoid pixelization or blurring of images, raster images are never upscaled. However, SVG images and TTF icons can be upscaled without any penalty.

If the icon must be enlarged, the image is centered and a transparent border is added to "fill" the empty space. This allows a mix of larger and smaller icons while keeping widget alignment.

If scaling takes place, the aspect ratio of the original image is kept. A non-square source image displays as a non-square scaled icon.

**Related concepts**

BUTTONEDIT item type on page 1267
Defines a line-edit with a push-button that can trigger an action.

**CheckBox style attributes**

CheckBox presentation style attributes apply to CHECKBOX elements.

**Note:** This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.
Table 321: CheckBox style attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>customWidget</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Defines the type of widget to be used to render the CHECKBOX.

Values can be:

- "toggleButton": The checkbox is rendered as a toggle button (also known as "toggle switch" in HTML/CSS).

**Important:** NULL values cannot be managed with the toggleButton widget type.

Default is to render the checkbox with a classical box using a check mark when set.

**Important:** The customWidget CheckBox style cannot be changed dynamically, once the widget has been displayed.

| iosCheckBoxOnTintColor | N/A | N/A | N/A | Yes |

On iOS devices, defines the color for the checkbox marker when on. This is different from backgroundColor, which is used for the tint of the whole switch.

**Related concepts**

- **CHECKBOX item type** on page 1269
  - Defines a boolean or three-state checkbox field.

**ComboBox style attributes**

ComboBox presentation style attributes apply to COMBOBOX elements.

**Note:** This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.

Table 322: ComboBox style attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>autoSelectionStart</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Defines the item from which the auto-selection will start, when pressing keys.

Values can be:

- "current": (default): the auto-selection looks for the first corresponding item after the current item of the object.
- "first": the auto-selection looks for the first corresponding item after the first item of the object.
**Attribute**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>comboboxCompleter</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Important:** This feature is deprecated, its use is discouraged although not prohibited.

Activate the ComboBox completer mode.

Possible values are "yes" and "no" (default).

When this attribute is set to yes, the ComboBox will have the following behavior:

- The ComboBox is editable, but only characters that match an item in the list are allowed (if the list contains the item "aa" and the item "ab", you can type "a", "aa", "ab", but nothing else. If you paste text in the field, it will be truncated until the rule is fulfilled.
- The drop-down list will only display item which starts with the same characters as the edit field. It is dynamically updated as you type (if the list contains the item "aa" and the item "ab" and you type "a", you will see both items displayed, but if you continue to type another "a", you will only see "aa" in the list.
- The best match is automatically selected when leaving the field (thus performing an "on change") as soon as you hit "TAB" key, even if the input is not complete.

**qtStyle**

Defines the Qt style of the widget. The intent of this property is to allow applying other style attributes such as backgroundColor, on Windows platforms, which is not possible with the default Windows style.

**Note:** Avoid using this style attribute whenever possible especially on non-Windows platforms

Values can be:

- "default" (default): the default Qt style is used.
- "Windows": The older "Windows XP" style is used.

**Note:** The combobox.qtStyle style attribute must be set to "Windows", to have other style attributes such as backgroundColor take effect.

**completionTimeout**

Defines the timeout (in milliseconds) to build the character sequence for item lookup when the user presses several keys successively. When pressing multiple keys, a character sequence is built for item lookup. After the timeout delay has expired, the character sequence is reset.

**Related concepts**

COMBOBOX item type on page 1270
Defines a line-edit with a drop-down list of values.

**DateEdit style attributes**

DateEdit presentation style attributes apply to `DATEEDIT` elements.

**Note:** This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.

**Table 323: DateEdit style attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>buttonIcon</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Defines the icon name to use for the button.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>calendarType</td>
<td>No</td>
<td>Yes (see note 1)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Controls the type of calendar pop-up window. Possible values are &quot;dropdown&quot; and &quot;modal&quot;. Default is &quot;modal&quot;. When set to &quot;modal&quot;, the calendar is intrusive to the <code>DATEEDIT</code> field: Once the calendar opens, the user must validate the date selection by clicking the OK button, or keep the current field value by clicking the Cancel button. When set to &quot;dropdown&quot;, the calendar is less intrusive: The user can type directly into the <code>DATEEDIT</code> field. A single click on a date validates and closes the calendar. There is no way to cancel the selected date.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>daysOff</td>
<td>Yes</td>
<td>Yes (see note 1)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Defines the days of the week that are grayed out. Possible values are &quot;monday&quot;, &quot;tuesday&quot;, &quot;wednesday&quot;, &quot;thursday&quot;, &quot;friday&quot;, &quot;saturday&quot;, &quot;sunday&quot;. Default is &quot;saturday sunday&quot;. The days of week can be combined, as shown.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>firstDayOfWeek</td>
<td>Yes</td>
<td>Yes (see note 1)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Defines the first day of the week to be displayed in the calendar. Possible values are &quot;monday&quot;, &quot;tuesday&quot;, &quot;wednesday&quot;, &quot;thursday&quot;, &quot;friday&quot;, &quot;saturday&quot;, &quot;sunday&quot;. Default depends on the front-ends platform language settings: For example, the default first day of week will be Sunday for an English/US locale, Monday for a French or German locale.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>showCurrentMonthOnly</td>
<td>Yes</td>
<td>Yes (see note 1)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Defines if dates of the previous and next months are shown. Values can be &quot;yes&quot;, &quot;no&quot; (default).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attribute</td>
<td>GDC</td>
<td>GBC</td>
<td>GMA</td>
<td>GMI</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>showGrid</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Indicates if the grid lines between dates must be visible in the calendar. Values can be &quot;yes&quot;, &quot;no&quot; (default).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>showWeekNumber</td>
<td>Yes</td>
<td>Yes (see note 1)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Defines if the week numbers are displayed. Values can be &quot;yes&quot;, &quot;no&quot; (default).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes:

1. With GBC, this style attribute is only available when using the built-in date picker. When using the mobile theme of GBC, the native browser date picker is used and built-in date picker rendering attributes cannot be applied.

Related concepts

DATEEDIT item type on page 1272
Defines a line-edit with a calendar widget to pick a date.

DateTimeEdit style attributes

DateTimeEdit presentation style attributes apply to DATEEDIT elements.

Note: This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.

Table 324: DateTimeEdit style attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>buttonIcon</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Defines the icon name to use for the button.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>calendarType</td>
<td>No</td>
<td>Yes (see note 1)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Controls the type of calendar pop-up window. Possible values are &quot;dropdown&quot; and &quot;modal&quot;. Default is &quot;modal&quot;. When set to &quot;modal&quot;, the calendar is intrusive to the DATETIMEEDIT field: Once the calendar opens, the user must validate the date selection by clicking the OK button, or keep the current field value by clicking the Cancel button. When set to &quot;dropdown&quot;, the calendar is less intrusive: The user can type directly into the DATETIMEEDIT field. A single click on a date validates and closes the calendar. There is no way to cancel the selected date.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attribute</td>
<td>GDC</td>
<td>GBC</td>
<td>GMA</td>
<td>GMI</td>
</tr>
<tr>
<td>--------------------</td>
<td>-----</td>
<td>----------------------</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td><strong>daysOff</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Defines the days of the week that are grayed out. Possible values are &quot;monday&quot;, &quot;tuesday&quot;, &quot;wednesday&quot;, &quot;thursday&quot;, &quot;friday&quot;, &quot;saturday&quot;, &quot;sunday&quot;. Default is &quot;saturday sunday&quot;. The days of week can be combined, as shown.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>No</td>
<td>Yes (see note 1)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>enableCalendar</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Defines if the widget must be rendered with a date/time picker. Values can be &quot;yes&quot;, &quot;no&quot; (default).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>firstDayOfWeek</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Defines the first day of the week to be displayed in the calendar. Possible values are &quot;monday&quot;, &quot;tuesday&quot;, &quot;wednesday&quot;, &quot;thursday&quot;, &quot;friday&quot;, &quot;saturday&quot;, &quot;sunday&quot;. Default depends on the front-ends platform language settings: For example, the default first day of week will be Sunday for an English/US locale, Monday for a French or German locale.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>No</td>
<td>Yes (see note 1)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>showCurrentMonthOnly</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Defines if dates of the previous and next months are shown. Values can be &quot;yes&quot;, &quot;no&quot; (default).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>No</td>
<td>Yes (see note 1)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>showGrid</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Indicates if the grid lines between dates must be visible in the calendar. Values can be &quot;yes&quot;, &quot;no&quot; (default).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>showWeekNumber</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Defines if the week numbers are displayed. Values can be &quot;yes&quot;, &quot;no&quot; (default).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>No</td>
<td>Yes (see note 1)</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Notes:

1. With GBC, this style attribute is only available when using the built-in date/time picker. When using the mobile theme of GBC, the native browser date/time picker is used and built-in date/time picker rendering attributes cannot be applied.

Related concepts

DATETIMEEDIT item type on page 1273
Defines a line-edit with a calendar widget to pick a datetime.

Action/MenuAction style attributes
These style attributes apply to default action views (MenuAction and Action classes).

Note: This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.
Table 325: Action/MenuAction style attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>androidActionPosition</td>
<td>N/A</td>
<td>N/A</td>
<td>Yes (note)</td>
<td>N/A</td>
</tr>
<tr>
<td>On Android™, defines if the option corresponding to the action must be displayed in the menu bar.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Values can be:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• &quot;button&quot;: The action view will be displayed in the Android™ action bar as a button, if there is no room in the action bar, the action view is not displayed.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• &quot;overflow&quot;: The action view will be displayed in the Android™ action bar overflow dropdown list.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• &quot;default&quot;: The action view will be displayed in the Android™ action bar, or in the overflow dropdown, if there is no room in the action bar.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Note:</strong> See also Android default action views decoration on page 1788.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>androidActionWithIcon</td>
<td>N/A</td>
<td>N/A</td>
<td>Yes</td>
<td>N/A</td>
</tr>
<tr>
<td>On Android™, defines if the icon (default icon or icon specified with the IMAGE attribute) must be displayed for the action view.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Values can be:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• &quot;yes&quot; (default): The icon is visible (default).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• &quot;no&quot;: The icon is not shown.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>androidActionWithText</td>
<td>N/A</td>
<td>N/A</td>
<td>Yes (see note)</td>
<td>N/A</td>
</tr>
<tr>
<td>On Android™, defines if a label (specified with the TEXT attribute) must be displayed for the action view.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Values can be:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• &quot;yes&quot; (default): Option text is visible (if there is an icon)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• &quot;no&quot;: Option text is not shown.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Note:</strong> If the device orientation is in portrait mode, Android™ may not display the text, even if you force it with this attribute.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### scaleIcon

Defines the scaling behavior of the associated icon, if the source image size is bigger than the place reserved for it in the widget.

**Note:** On GDC and GBC, if the `scaleIcon` attribute is undefined, the behavior depends on the type of action view: toolbar button icons and action panel button icons are scaled down to match the size of the widget. For other widgets, by default no scaling occurs, as for `scaleIcon="no"`.

Values can be:

- "no": No scaling occurs and the image is taken as-is. It is up to the developer to resize the source image to avoid misalignment. This is the default on GDC/GBC.
- "yes": Images are scaled down following the height of the widget (button or edit field). Setting a big font can result in a big icon. This is the default on GMA/GMI.
- "nnnpx": Images are scaled down based on the specified size. For example, `scaleIcon="128px"` will make every icon a maximum of 128*128 pixels. At least one side equal to 128 pixels, depending if the source image is square or not.

Independently of the `scaleIcon` attribute value, to avoid pixelization or blurring of images, raster images are never upscaled. However, SVG images and TTF icons can be upscaled without any penalty.

If the icon must be enlarged, the image is centered and a transparent border is added to "fill" the empty space. This allows a mix of larger and smaller icons while keeping widget alignment.

If scaling takes place, the aspect ratio of the original image is kept. A non-square source image displays as a non-square scaled icon.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>scaleIcon</td>
<td>Yes (see note)</td>
<td>Yes (see note)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Edit style attributes

Edit presentation style attributes apply to an `EDIT` element.

**Note:** This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.
Table 326: Edit style attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>dataTypeHint</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Important:** This feature is deprecated, its use is discouraged although not prohibited.

**Important:** Consider using the `KEYBOARDHINT` form attribute instead of the `dataTypeHint` style attribute.

Defines the type of the input, to let the front-end render a field behavior suitable for the particular data type.

Values can be:
- "email" (for email addresses)
- "url" (for URLs)
- "tel" (for telephone numbers)
- "search" (for search box fields)

<table>
<thead>
<tr>
<th>spellCheck</th>
<th>No</th>
<th>No (see note)</th>
<th>No</th>
<th>Yes (see note)</th>
</tr>
</thead>
</table>

Defines if the edit field includes a spelling checker.

**Note:**
- With GBC, the attribute is not applicable: Input fields use the web browser spellchecker.
- With GMI, available values are "yes" or "no". If this attribute is not set, iOS will decide if spell checking is enabled, depending on the global auto-correction setting on the device.

Related concepts

**EDIT item type** on page 1274
Defines a simple line-edit field.

**Folder style attributes**
Folder presentation style attributes apply to `FOLDER` tab elements.

**Note:** This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.

Table 327: Folder style attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>collapserPosition</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Indicates the position of the collapser icon, when the `position` style attribute is "accordion".

Values can be "left" (default), "right".
Attribute | GDC | GBC | GMA | GMI
--- | --- | --- | --- | ---
position | Yes (see note) | Yes | No | No

Defines the position and type of the folder tabs.
Values can be "top" (default), "left", "right", "bottom" and "accordion".
With "accordion", the folder pages can be collapsed and expanded.

**Note:** The "accordion" position is only supported by the GBC front-end.

---

**Related concepts**
**FOLDER item type** on page 1275
Defines a layout area to hold folder pages.

**Form style attributes**
Form presentation style attributes apply to a form element.

**Note:** This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.

**Table 328: Folder style attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>resetFormSize</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

This style attribute controls the resizing of the parent window, depending on the form content, when using successive OPEN FORM/DISPLAY FORM instructions (within the same current window)

Values can be "no" (default), "yes".

**Note:** The resetFormSize style attribute applies only to GDC front-end.

---

**Related concepts**
**DISPLAY FORM** on page 1138
Displays and associates a form with the current window.

**Group style attributes**
Group presentation style attributes apply to a GROUP element.

**Note:** This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.

**Table 329: Group style attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>collapserPosition</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Indicates the position of the collapsor icon.
Values can be "left" (default), "right".
### User interface

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>collapsible</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Defines if the group element can be collapsed and expanded. By default groups are not collapsible. Values can be &quot;yes&quot;, &quot;no&quot; (default).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>initiallyCollapsed</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Defines if a collapsible group is collapsed or expanded when the form is displayed. <strong>Note:</strong> This style attribute is ignored, if the collapsible attribute is not defined to &quot;yes&quot; for this group. Values can be:  • &quot;no&quot; (default): When displayed for the first time (no stored settings exist), the group appears expanded.  • &quot;yes&quot;: When displayed for the first time (no stored settings exist), the group appears collapsed.  • &quot;never&quot;: Each time it is displayed, the group appears expanded (stored settings and any previous display state are ignored).  • &quot;always&quot;: Each time it is displayed, the group appears collapsed (stored settings and any previous display state are ignored).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Related concepts**

**GROUP item type** on page 1276
Defines a layout area to group other layout elements together.

**HBox style attributes**

HBox presentation style attributes apply to an HBox element. **Note:** This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.

### Table 330: HBox style attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>splitViewRendering</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Indicates if the HBox must be displayed as a split view. • in landscape mode, panes are side by side and scroll independently  • in portrait mode, user navigates between the panes by swiping left or right Values can be &quot;yes&quot; or &quot;no&quot; (default).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Related concepts**

**HBOX item type** on page 1278
Defines a layout area to render child elements in horizontal direction. **Rendering an HBox as a splitview** on page 1907
Achieve a split view display with HBOX container and style attribute.

**Image style attributes**

Image style presentation attributes apply to an IMAGE element.

**Note:** This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.

**Table 331: Image style attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>alignment</td>
<td>Yes (see note)</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Defines the image alignment when the container is bigger than the image itself. Possible values are a pair of horizontal (&quot;left&quot;, &quot;horizontalCenter&quot;, &quot;right&quot;) and vertical alignments (&quot;top&quot;, &quot;verticalCenter&quot;, &quot;bottom&quot;). To combine alignment options, use a space as separator. Value can also be &quot;center&quot;, which is equivalent to &quot;horizontalCenter verticalCenter&quot;. The default value is &quot;top left&quot;. <strong>Important:</strong> With the GDC front-end, the alignment style attribute is not supported for IMAGE elements in TABLE containers.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>imageContainerType</td>
<td>Yes (see note)</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Important: This feature is deprecated, its use is discouraged although not prohibited. When set to &quot;browser&quot;, defines an image container as a browser. To use the image field as a browser, set a URL instead of an image name. <strong>Important:</strong> Instead of IMAGE fields with the imageContainerType style attribute, use URL-based Web Components: URL Web Components are much easier to use and more powerful.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Related concepts**

**IMAGE item type** on page 1279

Defines an area that can display an image resource.

**Label style attributes**

Label presentation style attributes apply to LABEL elements.

**Note:** This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.
### Table 332: Label style attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>textFormat</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Defines the rendering of the content of the label widget.

Possible values are:

- "plain" (default): the value assigned to this widget is interpreted as plain text.
- "html": it is interpreted as HTML (with hyperlinks).

**Important:** The HTML content displayed inside a form element using the `textFormat:html` style must not be a complete HTML document (using CSS styles for example). The content must be simple HTML, with basic tags such as text decoration like `<b/>` for bold, `<ul/>+<li/>` for bullet lists, and inline styles.

### Related concepts

**LABEL item type** on page 1280

Defines a simple text area to display a read-only value.

### Menu style attributes

Menu presentation style attributes apply to a `MENU` dialog.

**Note:** This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.

This table shows the presentation attributes for `Menu`:

#### Table 333: Menu style attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>position</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Defines the position of the automatic menu for “popup” menus.

Values can be:

- "cursor" (default): the pop-up menu appears at the cursor position.
- "field", the pop-up menu appears below the current field.
- "center", the pop-up menu appears at the center of the screen.
- "center2", the pop-up menu appears at the center of the current window.

### Related concepts

**Ring menus (MENU)** on page 1467

The `MENU` instruction implements a list of options the end user can choose from.

### Message style attributes

Message presentation style attributes apply to an `ERROR` or `MESSAGE` instruction.

The element type for both `ERROR` and `MESSAGE` is `Message`. To distinguish `ERROR` from `MESSAGE`, the ":error" or ":message" pseudo-selectors can be used to specify a different style for the rendering of each
instruction: "Message:error" corresponds to the ERROR instruction, and "Message:message" corresponds to the MESSAGE instruction.

The ERROR and MESSAGE instructions can get a STYLE attribute in the ATTRIBUTES clause, to specify a particular style name:

```
MESSAGE "No rows have been found." ATTRIBUTES(STYLE="info")
```

A limited set of common style attributes are supported for error/message display. In addition to the attributes described in the section, you can only define font style attributes for messages.

Like simple form fields, TTY attributes have a higher priority than style attributes. By default, ERROR has the TTY attribute REVERSE, which explains why ERROR messages have a reverse background, even when you use a backgroundColor style attribute. Use the NORMAL attribute in ERROR, to avoid the default REVERSE TTY attribute and define your own background color with a style.

Consider centralizing your ERROR and MESSAGE instruction calls in a function, to simplify global modifications:

```
FUNCTION my_error(m, s)
    DEFINE m, s STRING
    IF s IS NULL THEN
        ERROR m ATTRIBUTES(NORMAL)
    ELSE
        ERROR m ATTRIBUTES(NORMAL, STYLE=s)
    END IF
END FUNCTION
```

This table shows the presentation attributes for ERROR and MESSAGE instructions:

### Table 334: Presentation attributes for ERROR and MESSAGE instructions

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>position</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Defines the output type of the status bar message field.

Values can be:

- "statusbar" (default): will display the text in the regular statusbar of the window.
- "popup": will bring a pop-up window to the front; it is recommended to use this with care, as overuse can annoy the user.
- "statustip": will add a small "down" arrow button that will show the pop-up once the user clicks on it. This can be useful to display very long text.
- "both": will display the text in a pop-up window and then in the status bar.
Defines the rendering of the content of the widget.

Possible values are:

- "plain" (default): the value assigned to this widget is interpreted as plain text.
- "html", it is interpreted as HTML (with hyperlinks).

**Important:** The HTML content displayed inside a form element using the `textFormat:html` style must not be a complete HTML document (using CSS styles for example). The content must be simple HTML, with basic tags such as text decoration like `<b/>` for bold, `<ul/>+<li/>` for bullet lists, and inline styles.

### Related concepts

- **Pseudo selectors** on page 1168
  Pseudo selectors can be used to apply style only when some conditions are fulfilled.

- **ProgressBar style attributes**
  ProgressBar presentation style attributes apply to `PROGRESSBAR` elements.

  **Note:** This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.

#### Table 335: ProgressBar style attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>textFormat</code></td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

### Related concepts

- **PROGRESSBAR item type** on page 1283
  Defines a progress indicator field.

- **RadioGroup style attributes**
  RadioGroup presentation style attributes apply to `RADIOGROUP` elements.

  **Note:** This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.
Table 336: RadioGroup style attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>autoSelectionStart</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Defines the item from which the auto-selection will start, when pressing keys.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Values can be:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• &quot;current&quot; (default): it will look for the first corresponding item after the current item of the object.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• &quot;first&quot;: the auto-selection will look for the first corresponding item after the first item of the object.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>completionTimeout</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Defines the timeout (in milliseconds) to build the character sequence for item lookup when the user presses several keys successively. When pressing multiple keys, a character sequence is built for item lookup. After the timeout delay has expired, the character sequence is reset.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Related concepts
RADIOGROUP item type on page 1284
Defines a mutual exclusive set of options field.

ScrollGrid style attributes
ScrollGrid presentation style attributes apply to SCROLLGRID container.

Note: This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.

Table 337: ScrollGrid style attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>customWidget</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Defines the display mode to be used for the SCROLLGRID.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Values can be:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• &quot;pagedScrollGrid&quot;: Stretchable scrollgrids (defined with WANTFIXEDPAGESIZE=NO) will be rendered as a responsive tile list. Each row is rendered as a tile. The page size of the scrollgrid defines the number of tiles in each page.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Default is to render the scrollgrid as a vertical list.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Important: The customWidget ScrollGrid style attribute cannot be changed dynamically, once the widget has been displayed.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Attribute: `itemsAlignment`

<table>
<thead>
<tr>
<th></th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Defines how items are aligned in **stretchable scrollgrids** (defined with `WANTFIXEDPAGESIZE=NO`).

For the default rendering of stretchable scrollgrids, values of `itemsAlignment` can be:

- "stretch" (default): Each element takes the whole width of the scrollgrid.
- "left": Each element takes its minimal width and aligns to the left of the scrollgrid.
- "center": Each element takes its minimal width and aligns to the center of the scrollgrid.
- "right": Each element takes its minimal width and aligns to the right of the scrollgrid.

For **paged scrollgrids**, values of `itemsAlignment` can be:

- "left" (default): Elements flow and align to the left of the scrollgrid.
- "center": Elements flow and align to the center of the scrollgrid.
- "right": Elements flow and align to the right of the scrollgrid.

### Attribute: `rowActionTrigger`

<table>
<thead>
<tr>
<th></th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

Defines the physical event that will fire the row selection action (**DOUBLECLICK**) in a scrollgrid.

Values can be "singleClick" or "doubleClick".

Default is "doubleClick": a left-double-click triggers the action.

### Attribute: `highlightColor`

<table>
<thead>
<tr>
<th></th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

Defines the highlight color of rows for the **SCROLLGRID**, used for selected rows.

For possible values, see **Colors**.

For more details see **Row and cell highlighting in SCROLLGRID** on page 1204.

### Attribute: `highlightCurrentCell`

<table>
<thead>
<tr>
<th></th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Indicates if the current cell is highlighted in a **SCROLLGRID**, with the system default highlight colors, or the colors defined by the `highlightColor/highlightTextColor` style attributes.

Values can be "yes","no" (default depends on front-end and dialog type)

For more details see **Row and cell highlighting in SCROLLGRID** on page 1204.
<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>highlightCurrentRow</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Indicates if the current row is highlighted in a SCROLLGRID, with system default highlight colors, or the colors defined by the highlightColor/highlightTextColor style attributes.

Values can be "yes" or "no" (default depends on front-end and dialog type)

For more details see Row and cell highlighting in SCROLLGRID on page 1204.

| highlightTextColor            | Yes | Yes | Yes | No  |

Defines the highlighted text color of rows for the SCROLLGRID, used for selected rows.

For possible values, see Colors.

For more details see Row and cell highlighting in SCROLLGRID on page 1204.

Related concepts

SCROLLGRID item definition on page 1344
Defines a scrollgrid layout tag in a grid-based layout.

Row and cell highlighting in SCROLLGRID
SCROLLGRID containers can be configured with presentation styles, to control row and cell highlighting, using specific foreground and background colors.

Row/cell highlighting with SCROLLGRID
When using a SCROLLGRID, the same row/cell highlighting style attributes can be used as for TABLE containers, with some exceptions, as described in this topic.

DISPLAY ARRAY using FOCUSONFIELD attribute

- With GDC:
  - If highlightCurrentRow=yes, the current row is highlighted.
  - If highlightCurrentRow=no, the current row is not highlighted (same as default).

Table style attributes
Table presentation style attributes apply to a TABLE container.

Note: This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.
Table 338: Table style attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>allowWebSelection</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
| Users need to select items from a table. Once selected, the keyboard shortcut for copying data can be used to copy the selection.
Values can be "yes" or "no" (default).
When allowWebSelection="no", item selection requires the user to hold down the CTRL key while dragging over the selection with the mouse.
When allowWebSelection="yes", item selection requires a mouse drag only. There is no need to press the CTRL key. However, the ability to drag-and-drop data is disabled.
| forceDefaultSettings       | Yes | Yes | No  | No  |
| Values can be "yes" or "no" (default).
By default, tables are reopened with column positions, visibility and sizes they had when the window was closed. By setting this attribute to true, the saved settings are ignored and the table gets the initial column layout. Note that the saved settings include also the sort columns, that will impact on the order of the rows in the table.
| headerAlignment            | Yes | Yes | No  | No  |
| Defines the column header alignment in a table.
Values can be:
• "default" (default): will use the system default. In most case it is left aligned.
• "left" will force all column headers to be left aligned.
• "center" will force all column headers to be centered.
• "right" will force all column headers to be right aligned.
• "auto" will first try to align each column header following the "justify" attribute of the column. If no "justify" attribute is set, the column header will be aligned based on the type of data: right for numeric data, left for text data.
| headerHidden               | Yes | Yes | No  | No  |
| Defines if the horizontal header must be visible in a table.
Values can be "yes" or "no" (default).
| highlightColor             | Yes | Yes | Yes | No  |
| Defines the highlight color of rows for the TABLE, used for selected rows.
For possible values, see Colors.
For more details see Row and cell highlighting in TABLE on page 1208.
<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>highlightCurrentCell</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Indicates if the current cell is highlighted in a TABLE, with system default highlight colors, or the colors defined by the highlightColor/highlightTextColor style attributes. Values can be &quot;yes&quot;,&quot;no&quot; (default depends on front-end and dialog type) For more details see Row and cell highlighting in TABLE on page 1208.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>highlightCurrentRow</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Indicates if the current row is highlighted in a TABLE, with the system default highlight colors, or the colors defined by the highlightColor/highlightTextColor style attributes. Values can be &quot;yes&quot; or &quot;no&quot; (default depends on front-end and dialog type) For more details see Row and cell highlighting in TABLE on page 1208.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>highlightTextColor</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Defines the highlighted text color of rows for the TABLE, used for selected rows. For possible values, see Colors. For more details see Row and cell highlighting in TABLE on page 1208.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>leftFrozenColumns</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Requires &quot;tableType&quot; set to &quot;frozenTable&quot;. Defines how many columns are frozen, starting from the left of the table. Values can be any numeric value matching the number of columns. Default is &quot;0&quot;.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>resizeFillsEmptySpace</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Defines if the resize of the table adapts the size of the last column to avoid unused space. Values can be &quot;yes&quot; or &quot;no&quot; (default).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>reduceFilter</td>
<td>N/A</td>
<td>N/A</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Controls the usage of a reduce filter on tables. Values can be &quot;yes&quot; (default) or &quot;no&quot;. For more details see Reduce filter on page 1865.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attribute</td>
<td>GDC</td>
<td>GBC</td>
<td>GMA</td>
<td>GMI</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>rightFrozenColumns</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Requires &quot;tableType&quot; set to &quot;frozenTable&quot;.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Defines how many columns are frozen, starting from the right of the table.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Values can be any numeric value matching the number of columns.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Default is &quot;0&quot;.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rowActionTrigger</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Defines the physical event that will fire the row selection action (DOUBLECLICK) in a table.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Values can be &quot;singleClick&quot; or &quot;doubleClick&quot;.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Default is &quot;doubleClick&quot;: a left-double-click triggers the action.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>showGrid</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Indicates if the grid lines must be visible in a table.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Values can be &quot;yes&quot; (default with INPUT ARRAY) or &quot;no&quot; (default with DISPLAY ARRAY).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>By default, when a table is editable (when controlled by an INPUT ARRAY), the front-end displays grid lines in the table. You can change this behavior by setting the showGrid attribute to &quot;no&quot;.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>By default, when a table is read-only (when controlled by a DISPLAY ARRAY), the front-end does not display grid lines in the table. You can change this behavior by setting the showGrid attribute to &quot;yes&quot;.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>summaryLineAlwaysAtBottom</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Defines the placement of the summary row containing aggregate fields.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Values can be &quot;yes&quot; or &quot;no&quot; (default).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>When set to &quot;yes&quot;, the row containing aggregate fields is rendered in the last line of the table.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>When set to &quot;no&quot;, the row containing aggregate fields is rendered immediately after the values being aggregated. This is the default.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attribute</td>
<td>GDC</td>
<td>GBC</td>
<td>GMA</td>
<td>GMI</td>
</tr>
<tr>
<td>----------------</td>
<td>------------</td>
<td>------------</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>tableType</td>
<td>Yes (see note)</td>
<td>Yes (see note)</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Defines the rendering type of the table.

**Important:** The `tableType` style attribute cannot be changed dynamically, once the widget has been displayed.

Values can be:

- "normal" (default): Regular table rendering.
- "pictureFlow": The first column of the table is used to define the list of images to be used in the picture flow.

  **Note:** `pictureFlow` is only supported by GDC.

- "frozenTable": Users can "freeze" some columns when scrolling, so that they always remain visible. Default frozen columns can be defined with "leftFrozenColumns" and "rightFrozenColumns" attributes.

  **Note:** `listView` is not supported by GBC.

- "listView": The table is rendered as a mobile list view.

**Related concepts**

**TABLE item definition** on page 1346  
Defines attributes for a table layout tag, in a grid-based layout.

**STYLE attribute** on page 1395  
The `STYLE` attribute specifies a presentation style for a form element.

**Summary lines in tables** on page 1866  
Table views can display a summary line, to show aggregate values for columns.

**Row and cell highlighting in TABLE**  
TABLE containers can be configured with presentation styles, to control row and cell highlighting, using specific foreground and background colors.

**Purpose of row/cell highlighting style attributes**

Depending on the type of front-end, TABLE containers controlled by DISPLAY ARRAY dialog use a default rendering for the current row, and a default rendering for the current cell, when using INPUT ARRAY, INPUT or CONSTRUCT.

The following style attributes can be combined to change the rendering of the current row or current cell:

- `highlightCurrentRow` (yes/no): Controls current row highlighting.
- `highlightCurrentCell` (yes/no): Controls current cell highlighting.
- `highlightColor`: Defines the background color.
- `highlightTextColor`: Defines the text/foreground color.

The `highlightColor`/`highlightTextColor` can get color values. If these attributes are not specified and highlighting is required for the row or the cell, the front-end will use platform default highlighting colors.

**Note:** When using a dialog allowing user input such as INPUT ARRAY, cells can be edited and text can be selected in editor-based field types. When a highlight colors are used at the cell level, the front-ends will render the selected text accordingly.

A TABLE style can for example be defined as follows:

```xml
<Style name="Table.custom_style">
  <StyleAttribute name="highlightCurrentRow" value="no"/>
</Style>
```
Note: Depending on the front-end platform, some widgets may support different highlight colors to the system defaults. If your application is intended for several types of front-ends, consider testing the style attributes with all of them.

**Regular DISPLAY ARRAY (no FOCUSONFIELD attribute)**

When using a regular DISPLAY ARRAY with row-level focus granularity, the rendering of the current row is as follows:

- By default:
  - On GBC and GDC, the current row is highlighted.
  - On GMA and GMI, the current row is not highlighted.
- If highlightCurrentRow=yes, the current row is highlighted (default with GDC and GBC).
- If highlightCurrentRow=no, the current row is not highlighted (default with GMA and GMI).
- The attribute highlightCurrentCell is ignored.

**DISPLAY ARRAY using FOCUSONFIELD attribute**

When using a DISPLAY ARRAY with FOCUSONFIELD attribute:

- By default, the current cell is highlighted
- If highlightCurrentRow=yes, the current row is highlighted.
- If highlightCurrentRow=no, the current row is not highlighted (default).
- If highlightCurrentCell=yes, the current cell is highlighted (default).
- If highlightCurrentCell=no, the current cell is not highlighted.

Note: Exception on GDC:

- The highlightCurrentRow attribute is ignored (due to platform limitations).

**INPUT ARRAY**

When using an INPUT ARRAY (and INPUT, CONSTRUCT):

- By default, the current cell is not highlighted.
- If highlightCurrentRow=yes, the current row is highlighted.
- If highlightCurrentRow=no, the current row is not highlighted (default).
- If highlightCurrentCell=yes, the current cell is highlighted.
- If highlightCurrentCell=no, the current cell is not highlighted (default).

Note: Exception on GMA and GMI:

- INPUT ARRAY is not yet supported.

**Related concepts**

Row and cell highlighting in SCROLLGRID on page 1204

SCROLLGRID containers can be configured with presentation styles, to control row and cell highlighting, using specific foreground and background colors.

**Tree style attributes**

Tree presentation style attributes apply to the TREE container.

Note: This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.
**Note:** In the .4st file, the style attribute for TREE containers must be specified with the "Table" class (in the AUI tree, TREE containers are represented with Table nodes)

### Table 339: Tree style attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>allowWebSelection</td>
<td>No</td>
<td>Yes</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Users need to select items from a treeview. Once selected, the keyboard shortcut for copying data can be used to copy the selection. Values can be &quot;yes&quot; or &quot;no&quot; (default). When allowWebSelection=&quot;no&quot;, item selection requires the user to hold down the CTRL key while dragging over the selection with the mouse. When allowWebSelection=&quot;yes&quot;, item selection requires a mouse drag only. There is no need to press the CTRL key. However, the ability to drag-and-drop data is disabled.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>forceDefaultSettings</td>
<td>Yes</td>
<td>Yes</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>By default, tables are reopened with column positions, visibility and sizes they had when the window was closed. By setting this attribute to true, the saved settings are ignored and the table gets the initial column layout. Note that the saved settings include also the sort columns, that will impact on the order of the rows in the table. Values can be &quot;yes&quot; or &quot;no&quot; (default).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>headerAlignment</td>
<td>Yes</td>
<td>Yes</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Defines the column header alignment in a table. Values can be:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• &quot;default&quot; (default): will use the system default. In most case it is left aligned.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• &quot;left&quot; will force all column headers to be left aligned.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• &quot;center&quot; will force all column headers to be centered.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• &quot;right&quot; will force all column headers to be right aligned.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• &quot;auto&quot; will first try to align each column header following the &quot;justify&quot; attribute of the column. If no &quot;justify&quot; attribute is set, the column header will be aligned based on the type of data: right for numeric data, left for text data.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>headerHidden</td>
<td>Yes</td>
<td>Yes</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Defines if the horizontal header must be visible in a treeview. Values can be &quot;yes&quot; or &quot;no&quot; (default).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attribute</td>
<td>GDC</td>
<td>GBC</td>
<td>GMA</td>
<td>GMI</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>highlightColor</td>
<td>Yes</td>
<td>Yes</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
| Defines the highlight color of rows for the treeview, used for selected rows.
| For possible values, see Colors. |
| For more details see Row highlighting in TREE on page 1212. |
| highlightCurrentRow | Yes | Yes | N/A | N/A |
| Indicates if the current row is highlighted in a TREE, with the system default highlight colors, or the colors defined by the highlightColor/highlightTextColor style attributes. |
| Values can be "yes" or "no" (default depends on front-end and dialog type) |
| For more details see Row highlighting in TREE on page 1212. |
| highlightTextColor  | Yes | Yes | N/A | N/A |
| Defines the highlighted text color of rows for the TREE, used for selected rows.
| For possible values, see Colors. |
| For more details see Row highlighting in TREE on page 1212. |
| leftFrozenColumns   | No  | Yes | N/A | N/A |
| Requires "tableType" set to "frozenTable". |
| Defines how many columns are frozen, starting from the left of the treeview. |
| Values can be any numeric value matching with the number of columns. |
| Default is "0". |
| resizeFillsEmptySpace | Yes | Yes | N/A | N/A |
| Defines if the resize of the treeview adapts the size of the last column to avoid unused space. |
| Values can be "yes" or "no" (default). |
| rightFrozenColumns  | No  | Yes | N/A | N/A |
| Requires "tableType" set to "frozenTable". |
| Defines how many columns are frozen, starting from the right of the treeview list part. |
| Values can be any numeric value matching with the number of columns. |
| Default is "0". |
### rowActionTrigger

Defines the physical event that will fire the row selection action (DOUBLECLICK) in a treeview.

Values can be "singleClick" or "doubleClick".

Default is "doubleClick": a left-double-click triggers the action.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>rowActionTrigger</td>
<td>Yes</td>
<td>Yes</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### showGrid

Indicates if the grid lines must be visible in a treeview.

Values can be "yes" or "no" (default with DISPLAY ARRAY).

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>showGrid</td>
<td>No</td>
<td>Yes</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### tableType

Defines the rendering type of the list part of the treeview.

Values can be:
- "normal" (default): Regular treeview rendering.
- "frozenTable": Users can "freeze" some columns from scrolling, so that they always remain visible. Default frozen columns can be defined with "leftFrozenColumns" and "rightFrozenColumns" attributes.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>tableType</td>
<td>No</td>
<td>Yes</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Related concepts

**TREE item definition** on page 1349

Defines attributes for a tree layout tag, in a grid-based layout.

**Row highlighting in TREE**

TREE containers can be configured with presentation styles, to control row highlighting, using specific foreground and background colors.

**Row highlighting with TREE**

When using a TREE, the same row highlighting style attributes can be used as for TABLE containers.

**Note:** Only row highlighting is supported, because DISPLAY ARRAY + FOCUSONFIELD and INPUT ARRAY dialogs are not supported with TREE containers.

**TextEdit style attributes**

Textedit presentation style attributes apply to the TEXTEDIT element.

**Note:** This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.

### Table 340: TextEdit style attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>integratedSearch</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Defines if the textedit field allows search facility (Control-F).

Values can be "yes" or "no" (default).
<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>showEditToolBox</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Defines if the toolbox for rich text editing is shown or not. Possible values are &quot;auto&quot;(default), &quot;yes&quot;, &quot;no&quot;. Only available if textFormat style attribute is set to &quot;html&quot;.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>spellCheck</td>
<td>Yes (see note)</td>
<td>No (see note)</td>
<td>No</td>
<td>Yes (see note)</td>
</tr>
<tr>
<td>Defines if the textedit field includes a spelling checker.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note:**

- With GDC, the possible values are the two dictionary files needed for each language (one .aff and one .dic). These files can be downloaded here. Only the files available for OpenOffice.org 2.x are working (files for OpenOffice.org 3.x are not supported yet):
  - Simple relative locale files like
    "my_affix_file.aff"
    "my_dictionary_file.dic"
  - Absolute paths such as "file:///c:/dics/
    my_dictionary_file.aff|file:///c:/
dics/my_dictionary_file.dic"
  - URLs such as http://mywebserver.com/
    "my_affix_file.aff"
    http://mywebserver.com/
    "my_dictionary_file.dic"

The local directory of dictionary files can be queried in the GDC with the standard.feInfo front call using the dictionariesDirectory parameter.

- With GBC, the attribute is not applicable: Input fields use the web browser spellchecker.
- With GMI, available values are "yes" or "no". If this attribute is not set, iOS will decide if spell checking is enabled, depending on the global auto-correction setting on the device.
### Attribute: textFormat

Defines the rendering of the content of the widget.

Values can be:

- "plain" (default): the value assigned to this widget is interpreted as plain text.
- "html", the value is interpreted as HTML (with hyperlinks), with rich text input feature enabled.

**Important:** The HTML content displayed inside a form element using the `textFormat:html` style must not be a complete HTML document (using CSS styles for example). The content must be simple HTML, with basic tags such as text decoration like `<b/>` for bold, `<ul/>+<li/>` for bullet lists, and inline styles.

**Note:** Consider using the `fglrichtext` web component, instead of the rich text option in TEXTEDIT fields.

For more details, see Rich Text Editing in TEXTEDIT on page 1214.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>textFormat</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

### Attribute: wrapPolicy

Defines where the text can be wrapped in word wrap mode.

Values can be:

- "atWordBoundary" (default): the text will wrap at word boundaries.
- "anywhere": the text breaks anywhere, splitting words if needed.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>wrapPolicy</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

### Related concepts

**TEXTEDIT item type** on page 1289

Defines a multi-line edit field.

**Rich Text Editing in TEXTEDIT**

The GDC and GBC front-ends support a rich text editing interface with TEXTEDIT fields.

**Note:** Consider using the `fglrichtext` web component, instead of richtext option in TEXTEDIT fields.

**Important:** The HTML content displayed inside a form element using the `textFormat:html` style must not be a complete HTML document (using CSS styles for example). The content must be simple HTML, with basic tags such as text decoration like `<b/>` for bold, `<ul/>+<li/>` for bullet lists, and inline styles.
Rich text editing provides:

- Text format: bold, italic, underline
- Paragraph alignment: left, center, right, justify
- Lists: bullet, decimal
- Paragraph indentation
- Font size

To enable rich text editing, set the `textFormat` `styleAttribute` to `html`:

```xml
<Style name="TextEdit.richText">
  <StyleAttribute name="textFormat" value="html"/>
</Style>
```

**Richtext toolbox**

By default, when the mouse reaches the top border of the textedit field where rich text editing has been enabled, a toolbox appears. The toolbox disappears when the mouse leaves the top border area. This implementation is useful if you only use the textedit field to display rich text, as the toolbox is only visible in input.

If you want to always display the toolbox, you can set the `showEditToolBox` `styleAttribute`.

```xml
<Style name="TextEdit.richText">
  <StyleAttribute name="textFormat" value="html"/>
  <StyleAttribute name="showEditToolBox" value="yes"/>
</Style>
```
The following textedit supports html and richtext editing. It provides an embedded toolbox with most common editing actions.

![Rich text editing interface with toolbox always displayed.](image)

**Figure 48: Rich text editing interface with toolbox always displayed.**

**Tip:** The textedit field will be wide enough to display the toolbox in its entirety, even if you define a small width in your form definition file. Take this into account when designing your form.

**Tip:** The textedit field will be high enough to display the number of lines defined in the form definition file (using the textedit font) and the toolbox when required. A textedit with a height of 1 will display the toolbox and one line, which is much higher than without the toolbox.

**Important:** The behavior of the attribute `showEditToolBox` with the value `auto` differs between the Genero Desktop Client and the Genero Browser Client. With the Genero Desktop Client, 'auto' is interpreted as 'no'. With the Genero Browser Client, 'auto' is interpreted as 'yes'.

**Rich text local actions**

When using the `TEXTEDIT` rich text capability, a set of local actions are automatically created. These local actions can be configured like regular actions, to change the accelerator and decoration.

**Table 341: Local action names, accelerators, and icons**

<table>
<thead>
<tr>
<th>Name</th>
<th>Default Accelerator</th>
<th>Icon Name</th>
<th>Icon</th>
</tr>
</thead>
<tbody>
<tr>
<td>richtextbold</td>
<td>Ctrl-b</td>
<td>textbold</td>
<td>B</td>
</tr>
<tr>
<td>richtextitalic</td>
<td>Ctrl-i</td>
<td>textitalic</td>
<td>I</td>
</tr>
<tr>
<td>richunderline</td>
<td>Ctrl-u</td>
<td>textunder</td>
<td>U</td>
</tr>
<tr>
<td>richtextalignleft</td>
<td>Ctrl-l</td>
<td>textleft</td>
<td>E</td>
</tr>
<tr>
<td>richtextaligncenter</td>
<td>Ctrl-e</td>
<td>textcenter</td>
<td></td>
</tr>
<tr>
<td>richtextalignright</td>
<td>Ctrl-r</td>
<td>textright</td>
<td>R</td>
</tr>
<tr>
<td>richtextalignjustify</td>
<td>Ctrl-j</td>
<td>textjustify</td>
<td>J</td>
</tr>
<tr>
<td>richtextlistbullet</td>
<td>None</td>
<td>textlistbullet</td>
<td>L</td>
</tr>
<tr>
<td>Name</td>
<td>Default Accelerator</td>
<td>Icon Name</td>
<td>Icon</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------------------</td>
<td>-----------------</td>
<td>------</td>
</tr>
<tr>
<td>richtextlistdecimal</td>
<td>None</td>
<td>textlistnumbered</td>
<td></td>
</tr>
<tr>
<td>richtextdecreaseindent</td>
<td>None</td>
<td>textindentdecrease</td>
<td></td>
</tr>
<tr>
<td>richtextincreaseindent</td>
<td>None</td>
<td>textindentincrease</td>
<td></td>
</tr>
<tr>
<td>richtextdecreasefontsize</td>
<td>None</td>
<td>textfontsizedown</td>
<td></td>
</tr>
<tr>
<td>richtextincreasefontsize</td>
<td>None</td>
<td>textfontsizeup</td>
<td></td>
</tr>
</tbody>
</table>

You can hide the toolbox using the `showEditToolBox` style attribute.

```xml
<StyleAttribute name="textFormat" value="html" />
<StyleAttribute name="showEditToolBox" value="no" />
```

**Related reference**

- [TextEdit style attributes](#) on page 1212
  Textedit presentation style attributes apply to the `TEXTEDIT` element.

- **ToolBar style attributes**
  ToolBar presentation style attributes apply to the `TOOLBAR` element.

**Note:** This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.
### Table 342: ToolBar style attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>scaleIcon</td>
<td>Yes (see note)</td>
<td>Yes (see note)</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Defines the scaling behavior of the associated icon, if the source image size is bigger than the place reserved for it in the widget.

**Note:** On GDC and GBC, if the `scaleIcon` attribute is undefined, the behavior depends on the type of action view: toolbar button icons and action panel button icons are scaled down to match the size of the widget. For other widgets, by default no scaling occurs, as for `scaleIcon="no"`.

Values can be:

- "no": No scaling occurs and the image is taken as-is. It is up to the developer to resize the source image to avoid misalignment. This is the default on GDC/GBC.
- "yes": Images are scaled down following the height of the widget (button or edit field). Setting a big font can result in a big icon. This is the default on GMA/GMI.
- "nnnpx": Images are scaled down based on the specified size. For example, `scaleIcon="128px"` will make every icon a maximum of 128*128 pixels. At least one side equal to 128 pixels, depending if the source image is square or not.

Independently of the `scaleIcon` attribute value, to avoid pixelization or blurring of images, raster images are never upscaled. However, SVG images and TTF icons can be upscaled without any penalty.

If the icon must be enlarged, the image is centered and a transparent border is added to "fill" the empty space. This allows a mix of larger and smaller icons while keeping widget alignment.

If scaling takes place, the aspect ratio of the original image is kept. A non-square source image displays as a non-square scaled icon.

| toolBarTextPosition | Yes | Yes | No | No |

 Defines the text position of a ToolBarItem.

Values can be:

- "textBesideIcon"
- "textUnderIcon" (default)

### Related concepts

**TOOLBAR section** on page 1298

The TOOLBAR section defines a toolbar with buttons that are bound to actions.

**ToolBarSeparator style attributes**

ToolBarSeparator presentation style attributes apply to the toolbar SEPARATOR element.

**Note:** This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.
Table 343: ToolBarSeparator style attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>iosSeparatorStretch</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Stretches the SEPARATORS between toolbar items on iOS devices. When this attribute is set to yes, separators act like springs between the individual toolbar items.

Values can be:
- "no" (default): do not stretch toolbar item separators.
- "yes": stretch toolbar item separators.

Related concepts

TOOLBAR section on page 1298
The TOOLBAR section defines a toolbar with buttons that are bound to actions.

UserInterface style attributes
UserInterface presentation style attributes define general options related to the application user interface.

Note: This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.

Table 344: UserInterface style attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>androidKeepForeground</td>
<td>N/A</td>
<td>N/A</td>
<td>Yes (see note)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

An Android™ app can switch between foreground to background states.

By default, when the app goes to background state, a notification is shown by GMA to give a higher priority to the app, and to prevent Android™ stopping the app if resources are required for other apps. The notification disappears, when the app returns to foreground state.

Use the androidKeepForeground style attribute to control the way the GMA forces Android™ to keep your app alive.

Values can be:
- "yes" (default): The app remains in foreground state: to keep it in foreground state, a notification is displayed when the app goes background.
- "no": The app can switch between foreground and background state (no notification is displayed to force the app to remain in foreground state)

Important: When using androidKeepForeground=no, GMA will no longer use a notification to keep the app in foreground, and Android™ may stop the app at any time.
**browserMultiPage**

Defines whether subsequent RUN and RUN WITHOUT WAITING instructions will be executed in the current browser tab or in a new browser tab.

This style only works if the application having this style is the first of the session.

This style requires the Genero Application Server 3.00.22 or greater.

Values can be:
- "no" (default): Subsequent RUN and RUN WITHOUT WAITING instructions are executed in the current browser tab.
- "yes": Subsequent RUN and RUN WITHOUT WAITING instructions are executed in a new browser tab.

See also Window container interfaces on page 1145.

**reverse**

Global configuration option to display forms in reverse mode and enable right to left input, for right-to-left language support.

Values can be:
- "no" (default): Display forms for left-to-right languages.
- "yes": Display mirrored forms for right-to-left languages.

---

**Related concepts**

**The abstract user interface tree** on page 1106
The abstract user interface tree is the XML representation of the application forms displayed to the end user.

**Right-to-left languages support** on page 534
Genero supports right-to-left languages, such as Arabic and Hebrew.

**Background/foreground modes** on page 3573
Describes how to handle background or foreground modes in mobile apps.

**Window style attributes**

Window presentation style attributes apply to a window element.

**Note:** This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.

**Related concepts**

**Windows and forms** on page 1131
The section describes the concept of windows and forms in the language.

**Window style attributes: Basics**

Basic presentation style attributes for window elements.

Style attributes listed in this page are general and supported on several front-end types, when the platform allows it.
See also other pages about window style attributes.

---

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>browserMultiPage</td>
<td></td>
<td></td>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td>reverse</td>
<td></td>
<td></td>
<td></td>
<td>N/A</td>
</tr>
</tbody>
</table>

Defined attributes and values:

- **browserMultiPage**: N/A, Yes, N/A, N/A
- **reverse**: N/A, Yes, N/A, N/A
### Table 345: Window style attributes: Basics

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>border</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Defines the border type of the window.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Values can be:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• &quot;normal&quot; (default): the border is standard, with a normal window header with a caption.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• &quot;frame&quot;: only a frame appears, typically without a window header.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• &quot;tool&quot;: a small window header is used.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• &quot;none&quot;: the window gets no border.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>On Mac® platforms, using &quot;tool&quot; is not effective.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| defaultTTFColor       | Yes | Yes | Yes | Yes |
| Defines the default color used for TTF icons. |
| All icons displayed in the window using this style default get the color specified in the `defaultTTFColor` attribute. |
| The value for this attribute must be an RGB specification or a named color as listed in Colors on page 1173. |
| For more details about TTF icon usage see Using a simple image name (centralized icons) on page 1150. |

| errorMessagePosition  | Yes | No  | No  | No  |
| Defines the rendering of program errors displayed with the ERROR instruction. |
| Values can be: |
| • "statusbar" (default): displays the comment in the window status bar. |
| • "popup" will bring a pop-up window to the front; to be used with care, as it can annoy the end user. |
| • "statustip" adds a small "down" arrow button that shows the pop-up once the user clicks on it; useful to display very long text. |
| • "both" will display the comment text in a pop-up window and then in the status bar. |

| forceDefaultSettings | Yes | Yes | No  | No  |
| Indicates if the window content must be initialized with the saved positions and sizes. By default, windows are reopened at the position and with the size they had when they were closed. |
| You can force the use of the initial settings with this attribute. |
| This applies also to column position and width in tables. |
| Values can be "yes" or "no" (default). |
## Attribute: position

Indicates the initial position of the window.

Values can be:

- "default" (default): the windows are displayed depending on the window manager or browser rules.
- "field": the window is displayed below the current field (works as "default", when current field does not exist). The windowType style attribute must be "modal".
- "previous": the window is displayed at the same position (top left corner) as the previous window. (works as "default" when there is no previous window).

**Note:** "previous" is not supported by GBC.

- "center": the window is displayed in the center of the screen.
- "center2": the window is displayed in the center of the current window.

**Note:** "center2" is not supported by GBC.

For front-ends using stored settings, "field" and "previous" have higher priority than the stored settings.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>position</td>
<td>Yes</td>
<td>Yes (see notes)</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

## Attribute: sizable

Defines if the window can be resized by the user.

Values can be "yes" (default), "no" or "auto".

**Note:** With GDC:

- When using "auto", the window becomes resizable if the content of the first displayed form has resizable elements, for example when using a form with a TABLE container or an TEXTEDIT with STRETCH attribute.
- On Linux® and Mac® platforms, some window managers do not allow the size of a window to be frozen and therefore, sizable set to "no" may have no effect.

**Note:** With GBC:

- On a normal window, the behavior is applied to the form instead of the window. When set to "no", the form content is not stretched, even if the form contains stretchable items.
- On a modal window, when set to "no", the end-user cannot resize the modal window.
<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>tabbedContainer</td>
<td>Yes (see note)</td>
<td>Yes (see note)</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Values can be "yes" or "no" (default).

**Note:** For the GDC, `tabbedContainer=yes` indicates that the WCI container must display the child application windows in a folder tab.

**Note:** For the GBC (WCI is not supported), if a window of an application uses `tabbedContainer=yes`, every application except the one hosting the tabbed container is embedded in a tab container generated by GBC. When using this style attribute, the GBC side bar is replaced by the StartMenu, which is mandatory when `tabbedContainer=yes`.

See also Window container interfaces on page 1145.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>tabbedContainerCloseMethod</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Defines the folder tab method of the container when `tabbedContainer` is set to "yes".

Values can be:

- "container" (default): container gets a close button in the tab.
- "page": each page has its own close button.
- "both": each page and the container has its close button.
- "none": no close button is shown.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>thinScrollbarDisplayTime</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Defines the display time (in seconds) of the automatic scrollbar displayed when scrolling on fixed screen array (a.k.a. "Matrix") and SCROLLGRIDs. After the delay, the scrollbar disappears.

A value of zero specifies an infinite time: The thin scrollbar remains visible while the record list can be scrolled (during dialog execution).

A negative value specifies that the scrollbar must always be hidden.

Default is 1 second.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>toolBarDocking</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Defines if the toolbar is movable and floatable.

Values can be "yes" (default) or "no".
### toolBarPosition

Indicates the position of the toolbar, when a toolbar is defined.

Values can be "none", "chrome", "top" (default), "left", "bottom" or "right".

**Note:** GBC supports only the "chrome" value for the toolBarPosition attribute.

**Note:** The "chrome" value is specific to GBC: It indicates that action views must be displayed in the frame of the browser surrounding the HTML content. This is especially used on mobile devices. See Action views with GBC on page 1769.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>toolBarPosition</td>
<td>Yes</td>
<td>Yes (see note)</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

### windowOptionClose

Defines if the window can be closed with a system menu option or window header button.

Values can be "yes", "no" or "auto" (default).

When value is "auto", the option is enabled depending on the window type.

This attribute may have different behavior depending on the front-end operating system. For example, when no system menu is used, it may not be possible to have this option enabled.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>windowOptionClose</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

### windowOptionMaximize

Defines if the window can be maximized with a system menu option or window header button.

Values can be "yes", "no" or "auto" (default).

When value is "auto", the option is enabled depending on the window type.

This attribute may have different behavior depending on the front-end operating system. For example, when no system menu is used, it may not be possible to have this option enabled.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>windowOptionMaximize</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

### windowOptionMinimize

Defines if the window can be minimized with a system menu option or window header button.

Values can be "yes", "no" or "auto" (default).

When value is "auto", the option is enabled depending on the window type.

This attribute may have different behavior depending on the front-end operating system. For example, when no system menu is used, it may not be possible to have this option enabled.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>windowOptionMinimize</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
### Table 346: Window style attributes: Miscellaneous

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>allowedOrientations</td>
<td>N/A</td>
<td>N/A</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

- Defines possible orientations for mobile device.
- Values can be:
  - "all" (default): Any orientation is allowed.
  - "landscape": Landscape orientation (the display is wider than it is tall).
  - "portrait": Portrait orientation (the display is taller than it is wide).
  - "landscape_reverse": Landscape orientation in the opposite direction from normal landscape.
  - "portrait_reverse": Portrait orientation in the opposite direction from normal portrait.
  - "landscape_all": Normal and reverse landscape orientations are allowed.
  - "portrait_all": Normal and reverse portrait orientation are allowed.

**Note:** This attribute is supported at the Window level only by GMA.
<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>formScroll</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Defines if scrollbars are always to be displayed when the form is bigger than the screen, or only when the window is maximized. Values can be &quot;yes&quot; (default) or &quot;no&quot;.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ignoreMinimizeSetting</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Defines if the stored settings &quot;state=minimize&quot; is ignored when loading settings. To be used when minimized windows are not to be shown minimized when reopened. Values can be &quot;yes&quot; or &quot;no&quot; (default).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>materialFABActionList</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Defines a comma-separated list of action names that are bound to the Floating Action Button (FAB button), on a device following the material design guidelines. To be used in conjunction with the materialFABType attribute. The order of the actions define which action is triggered when the FAB button is tapped, and several matching actions are active. The default list of actions is: &quot;new, append, insert, update, edit&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>materialFABType</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Controls the Floating Action Button (FAB button) on a device following the material design guidelines. Possible values are: • &quot;single&quot; (default) - the FAB button is shown and maps to the first active action defined in the materialFABActionList attribute. • &quot;none&quot; - no FAB button must be displayed.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>menuPopupPosition</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Defines the position of the automatic menu for &quot;popup&quot; menus. Values can be: • &quot;cursor&quot; (default) - the pop-up menu appears at the cursor position. • &quot;field&quot; - the pop-up menu appears below the current field. • &quot;center&quot; - the pop-up menu appears at the center of the screen. • &quot;center2&quot; - the pop-up menu appears at the center of the current window.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Attribute

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>messagePosition</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Defines the rendering for program messages displayed with the `MESSAGE` instruction.

Values can be:

- "statusbar" (default): displays the comment in the window status bar.
- "popup" brings a pop-up window to the front; to be used with care, as it can annoy the end user.
- "statustip" adds a small "down" arrow button that will show the pop-up once the user clicks on it; useful to display very long text.
- "both" will display the comment text in a pop-up window and then in the status bar.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>statusBarType</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Defines the type of status bar the window displays.

See [Statusbar types](#) on page 1181 for all possible values.

Default is "default".

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>windowMenu</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Defines if the WCI container is to display an automatic "Window" menu, with *Cascade* and *Tile* features, and list of child windows.

Values can be "yes" or "no" (default).

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>windowSystemMenu</td>
<td>Yes</td>
<td>N/A</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Defines if the window shows a system menu.

Values can be "yes", "no" or "auto" (default).

When value is "auto", the system menu is enabled depending on the window type.

### Related concepts

[Windows and forms](#) on page 1131
The section describes the concept of windows and forms in the language.

**Window style attributes: Action Panel**
Presentation style attributes that apply to the window action panel.

**Table 347: Window style attributes: Action Panel**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>actionPanelButtonSize</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Defines the width of buttons for default action views.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>By default, buttons are sized depending on the text and/or image.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>This attribute gets a size specification as follows:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• <code>&lt;number&gt;em</code>: a number followed by the &quot;em&quot; unit (such as &quot;20em&quot;), defines a size relative to the height of the current font.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• &quot;normal&quot;, &quot;shrink&quot;, &quot;tiny&quot;, &quot;small&quot;, &quot;medium&quot;, &quot;large&quot; or &quot;huge&quot;. When using &quot;normal&quot; and &quot;shrink&quot;, buttons are sized depending on the text or image, where &quot;shrink&quot; uses the minimum size needed to display the content of the button.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>actionPanelButtonSpace</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Defines the space between buttons for default action views.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Values can be &quot;none&quot;, &quot;tiny&quot;, &quot;small&quot;, &quot;medium&quot;, &quot;large&quot; or &quot;huge&quot;.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Default is &quot;medium&quot;.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>actionPanelButtonTextAlign</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Defines the text alignment inside buttons for default action views.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Values can be &quot;left&quot;, &quot;center&quot;, &quot;right&quot;.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Default is &quot;left&quot; when the button have an icon, &quot;center&quot; otherwise.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>actionPanelButtonTextHidden</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Defines the text visibility inside buttons for default action views.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Values can be &quot;yes&quot; (default) or &quot;no&quot;.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>actionPanelDecoration</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Defines the decoration of the action panel for default action views.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Values can be &quot;auto&quot; (default), &quot;yes&quot;, &quot;no&quot; and &quot;dockable&quot;.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attribute</td>
<td>GDC</td>
<td>GBC</td>
<td>GMA</td>
<td>GMI</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>actionPanelHAlign</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Defines the alignment of the action panel for default action views.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>This attribute takes effect when actionPanelPosition is &quot;top&quot; or &quot;bottom&quot;.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Values can be &quot;left&quot; (default), &quot;right&quot; or &quot;center&quot;.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>actionPanelPosition</td>
<td>Yes</td>
<td>Yes (see note)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Defines the position for default action views frame.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Values can be &quot;none&quot;, &quot;chrome&quot;, &quot;top&quot;, &quot;left&quot;, &quot;bottom&quot; or &quot;right&quot; (default).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Note: The &quot;chrome&quot; value is specific to GBC: It indicates that action views must be displayed in the frame of the browser surrounding the HTML content. This is especially used on mobile devices. See Action views with GBC on page 1769.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>actionPanelScroll</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Defines if the action panel is &quot;ring&quot; - that is, when the last button is shown, pressing on the &quot;down&quot; button will show the first one again.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Values can be &quot;0&quot; or &quot;1&quot; (default).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>actionPanelScrollStep</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Defines how the action panel scrolls when clicking the &quot;down&quot; button, and the visibility of the next button.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Values can be:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• &quot;line&quot;: (default): the panel scrolls by one line, and then shows only the next button.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• &quot;page&quot;: the scrolling is done page by page.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Window style attributes: Ring Menu**

Presentation style attributes that apply to a window ring menu.

### Table 348: Window style attributes: Ring Menu

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>ringMenuButtonSize</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
| Defines the width of buttons for default action views (MENU instruction).
| By default, buttons are sized depending on the text and/or image. This attribute gets a size specification as follows:|
| - `<number>em`: a number followed by the "em" unit (such as "20em"), defines a size relative to the height of the current font.|
| - "normal", "shrink", "tiny", "small", "medium", "large" or "huge". When using "normal" and "shrink", buttons are sized depending on the text or image, where "shrink" uses the minimum size needed to display the content of the button. |
| ringMenuButtonSpace        | Yes | No  | No  | No  |
| Defines the space between buttons for default action views (MENU instruction).
| Values can be "none", "tiny", "small", "medium" (default), "large" or "huge". |
| ringMenuButtonTextAlign    | Yes | Yes | No  | No  |
| Defines the text alignment inside buttons for default action views (MENU instruction).
| Values can be "left", "center", "right".
| Default is "left" when the button have an icon, "center" otherwise. |
| ringMenuButtonTextHidden  | Yes | Yes | No  | No  |
| Defines the text visibility inside buttons for default action views (MENU instruction).
| Values can be "yes" (default) or "no". |
| ringMenuDecoration        | Yes | No  | No  | No  |
| Defines the decoration of the menu panel (MENU instruction).
| Values can be "auto" (default), "yes", "no" and "dockable".

Attribute | GDC | GBC | GMA | GMI
---|---|---|---|---
ringMenuHAlign | Yes | No | No | No

Defines the alignment of the ring menu frame (MENU instruction).

This attribute takes effect when ringMenuPosition is "top" or "bottom".

Values can be "left" (default), "right" or "center".

ringMenuPosition | Yes | Yes (see note) | No | No

Defines the position of the ring menu frame (MENU instruction).

Values can be "none", "chrome", "top", "left", "bottom" or "right" (default).

**Note:** The "chrome" value is specific to GBC: It indicates that action views must be displayed in the frame of the browser surrounding the HTML content. This is especially used on mobile devices. See Action views with GBC on page 1769.

ringMenuScroll | Yes | No | No | No

Defines if the focus can wrap in the ring menu default actions views, when pressing up or down keys.

Values can be "0" or "1" (default).

ringMenuScrollStep | Yes | No | No | No

Defines how the ring menu scrolls when moving to a next button that is not visible.

Values can be:

- "line" (default): the menu will scroll by one line, and show only the next button.
- "page", the scrolling will be done page by page.

**Window style attributes: Start Menu**
Presentation style attributes that apply to a window start menu.

**Table 349: Window style attributes: Start Menu**

Attribute | GDC | GBC | GMA | GMI
---|---|---|---|---
startMenuAccelerator | Yes | No | No | No

Defines the shortcut keys to execute the selected start menu item, when the position is defined as "tree" or "poptree".

By default, "space", "enter" and "return" start the application linked to the current item.
Attribute | GDC | GBC | GMA | GMI
--- | --- | --- | --- | ---
startMenuExecShortcut2 | Yes | No | No | No

Defines the shortcut keys to execute the selected start menu item, when the position is defined as "tree" or "poptree".

By default, "space", "enter" and "return" start the application linked to the current item.

startMenuPosition | Yes | Yes (see note) | No | No

Indicates the position of the start menu, when one is defined.

Values can be:

- "none" (default): the start menu is not displayed.
- "tree": the start menu is displayed as a treeview, always visible on the right side of the window.
- "menu": the start menu is displayed as a pull-down menu, always visible at the top of the window.
- "poptree": the start menu is displayed as a tree view in a pop-up window that can be opened with a shortcut (see startMenuShortcut).

**Note:** With GBC:

- The "poptree" rendering is not supported and falls back to the "tree" rendering.

startMenuShortcut | Yes | No | No | No

Defines the shortcut key to open a start menu, when the position is defined as "poptree".

Default is "control-shift-F12".

startMenuSize | Yes | No | No | No

Defines the size of the start menu, when one is defined and the position is defined as "tree" or "poptree".

Values can be "tiny", "small", "medium" (default), "large" or "huge".

The size depends on the font used for the start menu.

---

**Window style attributes: GMI/iOS**

Presentation style attributes that apply to a window on iOS device.

Table 350: Window style attributes: GMI/iOS

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N/A</td>
<td>N/A</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
On iOS devices, indicates if default action views must be rendered with iOS standard icons and labels (yes, the default), or must use the text and image attributes of the corresponding action (no).

Values can be "yes" (default) or "no".

The `iosRenderSystemActions` is taken into account depending on the image action attribute usage:

- If no image and no text attribute is used, GMI always uses the system action rendering, no matter what the value of `iosRenderSystemActions` is.
- If an image attribute is used, the image is always used to render the default action view, no matter what the value of `iosRenderSystemActions` is.
- If only a text attribute is used (no image attribute), `iosRenderSystemActions` defines if the iOS system action rendering is used (yes, the default) or if the text attribute is used (no)

See also iOS default action views decoration on page 1784.

On iOS devices, defines the color for items used in the navigation bar, toolbar, and some items in the forms (Buttons, SpinEdit, Radiogroups, row check mark, and disclosure indicators in list views).

This style attribute does not apply to MENU with `STYLE=dialog_popup`.

On iOS devices, defines the text color of the navigation bar.

On iOS devices, defines the background color of the navigation bar.

On iOS devices, defines background color of the toolbar.

On iOS devices, defines the background color of the tab bar.

The iOS tab bar is created with a `TYPE=NAVIGATOR window`. 

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>iosRenderSystemActions</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>iosTintColor</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>iosNavigationBarTextColor</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>iosNavigationBarTintColor</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>iosToolBarTintColor</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>iosTabBarTintColor</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Attribute | GDC | GBC | GMA | GMI
--- | --- | --- | --- | ---
iosTabBarUnselectedColor | N/A | N/A | N/A | Yes

On iOS devices, defines the color of unselected items in the tab bar.

This attribute works only for TTF icons. Use `iosTintColor` to define the color for selected tab bar items.

The iOS tab bar is created with a `TYPE=NAVIGATOR` window.

**Examples**
Presentation style (*.4st) usage examples.

**Example 1: Defining styles for grid elements**

This example shows how to define styles for grid elements.

The presentation style definition file:

```xml
<?xml version="1.0" encoding="ANSI_X3.4-1968"?>
<StyleList>
<!-- Applies to all type of elements -->
<Style name=".bigfont">
  <StyleAttribute name="fontSize" value="large" />
</Style>
<!-- Default text color and font family for all labels -->
<Style name="Label">
  <StyleAttribute name="textColor" value="blue" />
  <StyleAttribute name="fontFamily" value="sans-serif" />
</Style>
<!-- Background color for Edits having focus -->
<Style name="Edit:focus">
  <StyleAttribute name="backgroundColor" value="yellow" />
</Style>
<!-- Text color for Edits with STYLE="mandatory" -->
<Style name="Edit.mandatory">
  <StyleAttribute name="textColor" value="red" />
</Style>
</StyleList>
```

The form definition file:

```
LAYOUT
GRID
{
  [11 ][f1  ]
  [12 ][f2  ]
  [13 ][f3  ]
}
END
ATTRIBUTES
LABEL l1: TEXT="Label 1:"
EDIT f1 = FORMONLY.field1;
LABEL l2: TEXT="Label 2:"
EDIT f2 = FORMONLY.field2;
LABEL l3: TEXT="Label 3:", STYLE="bigfont"
EDIT f3 = FORMONLY.field3, STYLE="bigfont mandatory";
END
```

Program source file:

```
MAIN
```
DEFINE rec RECORD
  field1 STRING,
  field2 STRING,
  field3 STRING
END RECORD

LET rec.field1 = "Field 1"
LET rec.field2 = "Field 2"
LET rec.field3 = "Field 3"

CALL ui.Interface.loadStyles("styles")

OPEN FORM f1 FROM "form"
DISPLAY FORM f1

INPUT BY NAME rec.* WITHOUT DEFAULTS

END MAIN

Graphical result:

![Form displayed based on styles applied](image)

**Figure 49: Form displayed based on styles applied**

**How the styles were applied**

1. All labels get a blue text color and sans-serif font family because of the name="Label" style.
2. Label 3 and Edit 3 defined with the bigfont style name get a large font because of the name=".bigfont" style.
3. The Edit field having the focus gets a yellow background color because of the name="Edit:focus" style (using the focus pseudo-selector).
4. Edit fields defined with the mandatory style name get a red text color because of the name="Edit.mandatory" style.

**Related concepts**

*Syntax of presentation styles file (.4st)* on page 1167

A .4st presentation styles file is an XML file defining style attributes to be applied by front-ends.

**Example 2: Defining styles for table rows**

This example shows how to define styles for tables and table rows.

The presentation style definition file:

```xml
<?xml version="1.0" encoding="ANSI_X3.4-1968"?>
<StyleList>
```
The form definition file:

```
LAYOUT
TABLE
{
  [c1 | c2 | c3 ]
  [c1 | c2 | c3 ]
  [c1 | c2 | c3 ]
  [c1 | c2 | c3 ]
}
END

ATTRIBUTES
EDIT c1 = FORMONLY.col1, TITLE="C1";
EDIT c2 = FORMONLY.col2, TITLE="C2";
EDIT c3 = FORMONLY.col3, TITLE="C3", STYLE="bigfont";
END

INSTRUCTIONS
SCREEN RECORD sr(FORMONLY.*);
END
```

Program source file:

```
MAIN
  DEFINE arr DYNAMIC ARRAY OF RECORD
    col1 INTEGER,
    col2 STRING,
    col3 STRING
  END RECORD,
  i INTEGER
  FOR i=1 TO 20
    LET arr[i].col1 = i
    LET arr[i].col2 = "Item #"||i
    LET arr[i].col3 = IIF(i MOD 2, "odd", "even")
  END FOR
  CALL ui.Interface.loadStyles("styles")
  OPEN FORM f1 FROM "form"
  DISPLAY FORM f1
  DISPLAY ARRAY arr TO sr.*
END MAIN
```

Graphical result:
Figure 50: Form displayed based on styles applied

How the styles were applied

1. The odd rows get a yellow background because of the name="Table:odd" style (using the odd pseudo-selector).
2. Column 3 defined with the bigfont style name gets a large font because of the name=".bigfont" style.

Related concepts

Pseudo selectors on page 1168
Pseudo selectors can be used to apply style only when some conditions are fulfilled.

Syntax of presentation styles file (.4st) on page 1167
A .4st presentation styles file is an XML file defining style attributes to be applied by front-ends.

Form specification files

Form specification files are the source files defining the layout and content of application forms.

Understanding form files

A form specification file is a source file that defines an application form providing for end user interaction with a program.

The form file defines the disposition, presentation (in other words the decoration), and behavior of screen elements called form items.

The source file must have the .per file extension: myform.per. Programs load the .42f compiled version of the form files, and use interactive instructions (dialogs) to control the form.

To compile a .per source file to a .42f format, use the fglform form compiler. When a SCHEMA is specified in the form file, fglform requires that the database schema file already exist. Compiled form files depend on both the source files and the database schema files.

Compiled forms will be loaded by the programs with the OPEN FORM or the OPEN WINDOW WITH FORM instructions. The .42f form file is searched for in several directories, as described in the FGLRESOURCEPATH reference topic.
Once a form is loaded, the program can manipulate forms to display or let the user edit data, with interactive instructions such as INPUT or DISPLAY ARRAY. Program variables are used as display and/or input buffers.

The content of a .per form file must follow a specific syntax as described in Form file structure on page 1294.

**Related concepts**
- What are dialog controllers? on page 1707
- Application forms are controlled by interactive instruction blocks called dialogs. These blocks perform the common tasks associated with the form, such as field input and action handling.
- Form rendering on page 1415
  - The section explains the layout rules to render forms on graphical front-ends.
- Windows and forms on page 1131
  - The section describes the concept of windows and forms in the language.

**Form file concepts**
- Form specification file contains different types of elements.
- Form items
  - The concept of form item includes all elements used in the definition of a form.

**Definition**
- A form item can be an input field such as an EDIT field, a push BUTTON, a GROUPBOX, or a TABLE container. A form item can also be an element of a TOOLBAR, TOPMENU, and ACTION DEFAULTS definition.

A form item can be:
- A satellite item
- A static item
- A layout item
- A stack item
- An action view
- A form field

**Form item types**
- A form item is defined by its type, called a form item type. For example, a form field can be an EDIT, or a COMBOBOX. A form layout container can be a GROUP, or a GRID. A toolbar item can be an ITEM or a SEPARATOR.

For a detailed description, see Form item types on page 1266.

**Form items in grid-based containers**
- In a grid-based container such as GRID, form items (typically, form fields) must be defined with a form tag in the LAYOUT section, bound by the tag name to a definition in the ATTRIBUTES section.

The form tag defines the position and length of the form item, while the appearance and the behavior of the form item is defined by a set of attributes in the ATTRIBUTES section:

```
LAYOUT
GRID
{
    [f1         ]
    ...
}
END
END
...
ATTRIBUTES
EDIT f1 = customer.cust_name, ...
```
Form items in stack-based forms

In a stack-based container (STACK), form items (typically, form fields), are grouped and arranged in a given order, that will define their position in the stacked layout. The appearance and the behavior of the form item is defined by a list of attributes in the stack item definition:

```
LAYOUT
  STACK
  GROUP
    EDIT customer.cust_name, ...
  END
END
```

Satellite form items

Other kinds of form items are defined in the section it belongs to (for example, an ITEM element of a TOOLBAR definition).

Related concepts

Examples on page 1414
Form definition (.per) examples.

Satellite items

Satellite items are display elements defined outside the LAYOUT section.

Satellite items like the TOOLBAR, TOPMENU and ACTION DEFAULTS section are form elements independent from the main form layout, and are defined in addition to the LAYOUT section.

```
TOOLBAR -- Toolbar section
...
END
LAYOUT -- Main layout section
...
END
```

Related concepts

LAYOUT section on page 1301
The LAYOUT section defines the graphical alignment of the form by using a tree of layout containers.

Static items

A static item defines a simple form item as a final grid element that does not change.

A static item is a form element, such as text (typically, a field label), that is defined directly in a GRID (or SCROLLGRID) container in the form LAYOUT section.

Static items are identified by the fglform compiler and converted to an AUI tree node element in the resulting .42f file.

Simple texts

It is possible to define simple texts and field labels in the form layout:

```
LAYOUT
  GRID
  {
    A simple text
  }
```
Note: To simplify internationalization, consider using named static labels instead of hard-coded text in the form layout.

**Horizontal lines**

You define a horizontal line with a sequence of hyphen-minus (−) characters in a grid:

```
LAYOUT
GRID
{This is a horizontal line: ---------}
END
END
```

Note: Horizontal lines are mainly provided for TUI mode applications. While horizontal lines will be represented by some GUI front-ends, it is not a typical practice in common graphical applications.

**Related concepts**

*LAYOUT section* on page 1301

The LAYOUT section defines the graphical alignment of the form by using a tree of layout containers.

*Layout items*

*Layout items* are containers with a body that can hold other form items, in a grid-based layout form.

Layout items can be specified as a tree of nested containers, or as layout tags within a single `GRID` container.

The example shows a tree of nested containers, where a `GRID` and `TABLE` are included in a `VBOX`:

```
LAYOUT
VBOX
  GRID ...
  { }
  END
  TABLE ...
  { }
  END
END
END
```

The example shows a `GRID` container including layout tags. The layout tags group form fields in dedicated areas. This syntax is usually more convenient to describe application forms:

```
LAYOUT
GRID
{<g g1                     >
  Name: [f01              ]<
  <
  <t t1                     >
  [c1  |c2                  ]<
  }
END
END
```
Related concepts

**LAYOUT section** on page 1301

The LAYOUT section defines the graphical alignment of the form by using a tree of layout containers.

**Stack items**

Stack items are form elements used to define a stack-based layout in a STACK container.

To define a stacked layout within a STACK container, leaf stack items (typically, form fields, labels, buttons) are specified inside grouping stack items such as GROUP or TABLE.

The example shows a stack-based form definition with a GROUP stack item containing two EDIT stack items:

```plaintext
LAYOUT STACK
  GROUP g1
    EDIT customer.cust_num, NOENTRY;
    EDIT customer.cust_name, REQUIRED;
  END
END
END
```

Related concepts

**Stack-based layout** on page 1431

A form file can define a stack-based layout within a tree of stack items.

**Action views**

An action view defines a form item that can trigger an action in the program.

**Action views as satellite items**

Below is TOOLBAR section defining a toolbar button using the close action name. Here no layout tag is used because the toolbar item is part of the toolbar graphical object (it will not appear in the form layout area):

```plaintext
TOOLBAR
  ITEM close (TEXT="Close")
END
```

**Action views in grid-based container**

In a GRID container, the position and size of the element is defined with an item tag, while the rendering and behavior is defined in the ATTRIBUTES section. Both parts are bound by the name of the item tag. The item tag name is local to the .per file and is not available at runtime.

The example defines a BUTTON form item, where the item tag name is "b_close", and the button name (and the action name) is "close":

```plaintext
LAYOUT GRID
{...
  [b_close ]
}
END
END
...
ATTRIBUTES
BUTTON b_close: close, TEXT="Close";
END
```
Action views in stack-based layout

In a stack-based container, action views are defined as stack items, with the attribute defining the rendering and behavior:

```
LAYOUT
STACK
GROUP group1 ( TEXT="Customer" )
...
BUTTON print, TEXT="Print Report", IMAGE="printer";
...```

Related concepts

Dialog actions on page 1739
Describes how to program action handling when the end user triggers an action on the front-end.

Form fields

Form fields are form elements designed for data input and/or data display.

Purpose of form fields

A form field is a form item dedicated to data management. It associates a form item with a screen record field. The screen record field is used to bind program variables in interaction instructions (for example, dialogs). The program variables are the data models for the form fields.

There are different types of form fields:

- Database column fields on page 1243
- Formonly fields on page 1245
- Phantom fields on page 1246
- Aggregate fields on page 1248

Form fields can be used in a grid-based layout or in a stack-based layout.

Form fields are identified by the field name in programs, and are grouped in screen records (or screen arrays in the case of list containers). The interactive instruction must mediate between screen record fields and database columns by using program variables.

Form fields are usually related to database column, which types are defined in the database schema file.

Forms fields in grid-based containers

In a grid-based container, the position and size of a form field is defined with an item tag in the form layout, while the rendering and behavior is defined in the ATTRIBUTES section. Both parts are bound by the name of the item tag. The item tag name is local to the .per file and is not available at runtime: It is just the key to bind the item tag (position) with the item definition (attributes).

In the grid-based example, the "f1" item tag (in the LAYOUT section) is linked to the "vehicle.num" form field definition (in the ATTRIBUTES section), which references a column of the "vehicle" table, defined in the "carstore" database schema:

```
SCHEMA carstore
LAYOUT
GRID
{  
  Number:   [f1            ]
  Name:     [f2                   ]
}
END
END
TABLES
  vehicle```
Forms fields in stack-based containers

In a stack-based container, the visual position of a form field is defined by the ordinal position of the stack item in the stack definition, while the rendering and behavior are defined with stack item attributes.

In the stack-based example, the "vehicle.num" form field definition references a column of the "vehicle" table, defined in the "carstore" database schema:

```
SCHEMA carstore
LAYOUT
  STACK
  GROUP
    EDIT vehicle.num, REQUIRED, STYLE="keycol";
  END
END
END
END
tABLES
  vehicle
END
```

Related concepts

ATTRIBUTES section on page 1330

The ATTRIBUTES section describes properties of grid-based layout elements used in the form.

Database column fields

Form fields defined with a table and column name get data type from the database schema file.

Syntax 1: In grid-based container

```
item-type item-tag = [table.]column
  ↓, attribute-list ↓ ;
```

Syntax 2: In stack-based container

```
item-type [table.]column
  ↓, attribute-list ↓ ;
```

1. `item-type` references an item type like EDIT.
2. `item-tag` identifies the layout location of the field.
3. `table` is the name or alias of a table, synonym, or view, as declared in the TABLES section.
4. `column` is the name of a database column.
5. `attribute-list` is a list of field attributes.

Usage

A form field is typically based on the definition of a database column found in the database schema specified with the SCHEMA clause at the beginning of the form file. The database column defines the data type of the form field.

Important: The data type of a form field is only used by the CONSTRUCT interactive statement to do database queries. When using the form field with an INPUT, INPUT ARRAY or DISPLAY ARRAY dialog, the type of the program variable defines the data type of the form field.
In order to reference database columns, the table name must be listed in the TABLES section of the form.

Fields are associated with database columns only during the compilation of the form specification file: The form compiler examines the database schema file to identify the data type of the column, and defines the form field with this type. This technique allows form field data types in the schema files to be centralized. If the data type of a column changes, extract the schema again and recompile your forms to take the new type into account.

Note: The compilers also grab other field attributes like validation rules and video display attributes from .val and .att schema files. However, this is supported for backward compatibility only (formerly stored in syscolval and syscolatt database tables). Consider reviewing programs using this feature.

After the form compiler identifies data types from the schema file, the association between fields and database columns is broken, and the form cannot distinguish the name or synonym of a table or view from the name of a screen record.

The programs only have access to screen record fields, in order to display or input data using program variables. Regardless of how you define them, there is no implicit relationship between the values of program variables, form fields, and database columns. Even, for example, if you declare a variable lname LIKE customer.lname, the changes that you make to the variable do not imply any change in the column value. Functional relationships among these entities must be specified in the program code, through screen interaction statements, and through SQL statements. It is up to the programmer to determine what data a form displays and what to do with data values that the user enters into the fields of a form. You must indicate the binding explicitly in any statement that connects variables to forms or to database columns.

If a form field is declared with a table column using the SERIAL, SERIAL8 or BIGSERIAL SQL type, the field will automatically get the NOENTRY attribute, except if the field is defined with the TYPE LIKE syntax.

Example

Grid-based container database form field definition:

```plaintext
SCHEMA stores -- Database schema
LAYOUT
GRID
{
  [f001             ]
  ...
}
END
END
TABLES
  customer -- Database table
END
ATTRIBUTES
EDIT f001 = customer.fname, -- DB-col form field
  REQUIRED, COMMENTS="Customer name";
...
```

Stack-based container database form field definition:

```plaintext
SCHEMA stores -- Database schema
TABLES
  customer -- Database table
END
LAYOUT
STACK
GROUP
  EDIT customer.fname, -- DB-col form field
    REQUIRED, COMMENTS="Customer name";
...
```
Related concepts
Formonly fields on page 1245

Formonly fields

**FORMONLY** form fields define their data type explicitly, with or without referencing a database column.

**Syntax 1: In grid-based container**

```plaintext
item-type item-tag = FORMONLY.field-name
  ↓ TYPE
    ↓ LIKE table.column
    ↓ datatype [NOT NULL] ↓
    ↓ attribute-list ↓ ;
```

**Syntax 2: In stack-based container**

```plaintext
item-type FORMONLY.field-name
  ↓ TYPE
    ↓ LIKE table.column
    ↓ datatype [NOT NULL] ↓
    ↓ attribute-list ↓ ;
```

where **datatype** is one of:

- CHAR
- DECIMAL (p,s)
- SMALLFLOAT
- REAL
- FLOAT
- MONEY (p,s)
- INTEGER
- SMALLINT
- DATE
- VARCHAR
- TEXT
- BYTE
- INTERVAL interval-qualifier
- DATETIME datetime-qualifier
- BIGINT
- BOOLEAN

1. **table** is the name or alias of a table, synonym, or view, as declared in the TABLES section.
2. **column** is the name of a database column.
3. **field-name** is the identifier that will be used in programs to handle the field.
4. **interval-qualifier** is an INTERVAL qualification clause such as HOUR(5) TO SECOND.
5. **datetime-qualifier** is a DATETIME qualification clause such as DAY TO SECOND.

**Usage**

Form fields can be specified with the **FORMONLY** prefix, when there is no corresponding database column, or when the field must be defined with another name to that of the database column.

**Important:** The data type of a form field is only used by the **CONSTRUCT** interactive statement to do database queries. When using the form field with an **INPUT**, **INPUT ARRAY** or **DISPLAY ARRAY** dialog, the type of the program variable defines the data type of the form field.
When using the LIKE [table.]column syntax, the form field gets the data type of the specific table column as defined in the database schema. The table name must be specified in the TABLES section.

When using the TYPE datatype clause, you explicitly specify the type of the field.

**Note:** For CHAR/VARCHAR data types, the size is defined by the item tag length in the layout.

If no data type is specified, and no database column is referenced, the default data type is CHAR.

Specifying a data type followed by the NOT NULL keyword is equivalent to the NOT NULL attribute.

The STRING data type is not supported in FORMONLY form field definitions.

The definition of FORMONLY fields can be completed by using the DISPLAY LIKE and VALIDATE LIKE attributes, to get the display and validation attributes from the .att and .val database schema files.

**Example**

**Grid-based container** FORMONLY form field definition (in the ATTRIBUTES section):

```plaintext
LAYOUT
    GRID
    {
        [f001   
        [f002   
        ...
    }
    END
    END
    ATTRIBUTES
    EDIT f001 = FORMONLY.total TYPE DECIMAL(10,2), NOENTRY ;
    EDIT f002 = FORMONLY.name TYPE LIKE customer.cust_name,
                VALIDATE LIKE customer.cust_name ;
```

**Stack-based container** FORMONLY form field definition:

```plaintext
LAYOUT
    STACK
    GROUP
    EDIT FORMONLY.total TYPE DECIMAL(10,2), NOENTRY ;
    EDIT FORMONLY.name TYPE LIKE customer.cust_name, REQUIRED;
```

**Related concepts**

*Database column fields* on page 1243

Form fields defined with a table and column name get data type from the database schema file.

**Phantom fields**

A PHANTOM field defines a screen-record field which is not rendered in the layout (it acts as a hidden field).

**Syntax**

```plaintext
PHANTOM \[ [table.]column \] \[ FORMONLY.field-name \] \[ TYPE \] \{ LIKE [table.]column \} \[ data-type [NOT NULL] \] \;
```

where datatype is one of:

```plaintext
\{ CHAR \} \{ DECIMAL \}(p,s)\}
```
1. **table** is the name or alias of a table, synonym, or view, as declared in the TABLES section.
2. **column** is the name of a database column.
3. **field-name** is the identifier that will be used in programs to handle the field.
4. **interval-qualifier** is an INTERVAL qualification clause such as HOUR(5) TO SECOND.
5. **datetime-qualifier** is a DATETIME qualification clause such as DAY TO SECOND.

**Usage:**

A PHANTOM field defines a form field listed in a screen-record or screen-array, that has no corresponding layout element. It is only used for the screen-record (or screen-array) definition, to bind with program variables used by dialogs, typically to match a given database table definition.

Phantom fields are used by dialog instructions as regular form fields, but are not displayed to the end user, and the end user is not able to enter values for these fields. Data held by phantom fields is never send to the front-ends. They can be used to store critical data that must not go out of the application server.

Phantom fields can be based on columns defined in a database schema file, or as FORMONLY field.

For example, if you want to implement a screen-array with all the columns of a database table defined in the database schema file, but you don't want to display all the columns in the TABLE container of the LAYOUT section, you must use PHANTOM fields. With the screen-array matching the database table, you can easily write program code to fetch all columns into an array defined with a LIKE clause.

**Example (grid-based layout)**

Form file:

```plaintext
SCHEMA carstore
LAYOUT( TEXT = "Vehicles" )
GRID
{
  <T t1                                  >
  Num      Name            Price   
  [c1      |c2             |c3           ]
  [c1      |c2             |c3           ]
  [c1      |c2             |c3           ]
}
END
END
TABLES
  vehicle
END
ATTRIBUTES
  TABLE t1: table1;
  EDIT c1 = vehicle.num;
  EDIT c2 = vehicle.name;
```
EDIT c3 = vehicle.price;
   PHANTOM vehicle.available; -- not used in layout
END
INSTRUCTIONS
   SCREEN RECORD sr(vehicle.*);
END

Program code:

SCHEMA carstore
...
DEFINE vl DYNAMIC ARRAY OF RECORD LIKE vehicle.*
...
DISPLAY ARRAY vl TO sr.*
...

Related concepts
Array binding in list controllers on page 1831
Program array elements are bound to screen arrays elements in the definition of the DISPLAY ARRAY or INPUT ARRAY list dialog.

GRID container on page 1308
Defines a layout area based on a grid of cells.

Aggregate fields
An AGGREGATE field defines a screen-record field to display summary information for a TABLE column.

Syntax

AGGREGATE item-tag = field-name \, attribute-list \ ;

1. item-tag is an identifier that defines the name of the item tag in the layout section.
2. field-name identifies the name of the screen record field.
3. attribute-list defines the aspect and behavior of the form item.

Usage

An AGGREGATE field defines a form field that is used to display a summary cell for a given column of a TABLE container. The aggregate fields are displayed after the last data line of the table. Such fields are typically used to show computed values for the corresponding column which appears above the aggregate cell.

Important: This feature is not supported on mobile platforms.

An aggregate field can be based on a database column defined in a schema file, or as FORMONLY field.

The AGGREGATETYPE attribute defines how the value of the field will be computed. For example, the SUM keyword (the default) can be used to instruct the runtime system to automatically compute the total of the associated column. By using the PROGRAM keyword, you indicate that the value of the aggregate field will be computed and displayed by program code. A simple DISPLAY BY NAME or DISPLAY TO can be used to show the summary value.

The value displayed in the AGGREGATE field follows the FORMAT attribute of the corresponding column, if defined. The FORMAT attribute is applied for automatically computed values, as well as for values displayed by user code with DISPLAY BY NAME or DISPLAY TO.

The label of an aggregate field can be specified with the AGGREGATETEXT attribute. The text defined with this attribute will be displayed on the left of the aggregate value (in the aggregate cell), except if there is no room to display the label (for example if the aggregate value is too large or if the column values are aligned to the left). An aggregate label can be a localized string with the \%"..." string syntax. You can also specify an AGGREGATETEXT attribute at the TABLE level, to get a global label for the summary line. If no text is defined for an aggregate field, the global aggregate text will appear on the left in the summary line.
Table aggregate decoration can be modified with a presentation style. Use the `summaryLine` pseudo-selector to change the font type and color, as well as the background of the summary line. Use the `summaryLineAlwaysAtBottom` table style attribute to force the summary line to stay on the bottom of the table.

**Aggregate fields in grid-based layout**

The item tag of an aggregate field must appear in the last line in the layout block of the `TABLE` container, and must be aligned vertically with a table column item tag. You can specify several aggregate item tags for the same table:

```plaintext
TABLE
{
| c1   | c2   | c3   | c4   | c5   |
| c1   | c2   | c3   | c4   | c5   |
| c1   | c2   | c3   | c4   | c5   |
| cnt  |     |     | tot_c4 | tot_c5 |
}
END
```

**Aggregate fields in stack-based layout**

**Important:** Aggregate fields are not supported in tables defined in a `STACK` container.

**Example (grid-based layout)**

```plaintext
SCHEMA stores
LAYOUT( TEXT = "Orders" )
GRID
{
<T t1                                  >
  Num      Date            Order total
| c1      | c2   | c3   | c4   | c5   |
| c1      | c2   | c3   | c4   | c5   |
| c1      | c2   | c3   | c4   | c5   |
| total   |     |     | tot_c4 | tot_c5 |
}
END
```

**Related concepts**

- [AGGREGATE item definition](page 1332)
Defines screen-record fields that hold computed values to be displayed as footer cells in a TABLE container.

**Identifying form items**
Elements defined in a form file can be identified with a name, to be used in programs.

Form fields are implicitly identified by the `tabname.colname` specification after the equal sign, while other (non-field) form items such as static labels and group boxes can get an optional item name.

The form item name defined in the form file will be copied to the `name` attribute of the corresponding node in the .42f file. It can then be used by programs to select a form element at runtime, to introspect or modify its attributes.

For example, specify the name for a GROUP container by writing an identifier after the layout container type:

```
GROUP group1 (TEXT="Customer")
```

Here the group name is 'group1', and it can be used in a program to identify the group element:

```
DEFINE w ui.Window
DEFINE g om.DomNode
LET w = ui.Window.getCurrent()
LET g = w.findNode("Group","group1")
CALL g.setAttribute("text","Another text")
```

Helper methods are provided for common tasks on form elements. For example, to hide a group with the identifier `group1`, you can use the `setElementHidden()` method on a `ui.Form` object:

```
DEFINE f ui.Form
...  
  LET f = DIALOG.getForm()
  ...  
  CALL f.setElementHidden("group1", TRUE)
```

**Note:** Consider defining unique names to identify form elements, and to simplify the search at runtime. A good practice is the use of a prefix based on the type of form element (`g_` for groups, `l_` for labels for example).

Static items in a grid-based layout container cannot get a name, because these are self-defined with the layout part of the item:

```
GRID
{  
  Name: [f1               ]
  ...  
}
END
```

In the above example, the label "Name:" cannot be identified. In order to give a name to such a label, use an item tag and add a `LABEL` line in the `ATTRIBUTES` section, and specify the name of the label after the colon:

```
GRID
{  
  [l1   ][f1               ]
  ...  
}
END
...  
ATTRIBUTES
LABEL 11: l_name, TEXT="Name:";
...  
```

**Related concepts**

`ATTRIBUTES section` on page 1330
The ATTRIBUTES section describes properties of grid-based layout elements used in the form.

**Screen records / arrays**

Form fields can be grouped in a screen record or screen array definition.

**Syntax**

```
SCREEN RECORD record-name [ [ size ] ] ( field-list )
```

where field-list is:

```
| table.* |
| field-name |
| first-field [ THROUGH | THRU ] last-field |
| , , .. , |
```

1. record-name is the name of an explicit screen record or screen array.
2. size is an integer representing the number of records in the screen array. If the size is not specified in the SCREEN RECORD definition, it is deduced from the corresponding list container in the form layout.
3. field-name is a field identifier as defined in the right operand of a field definition in the ATTRIBUTES section.
4. first-field and last-field are field identifiers like field-name. This notation instructs the form compiler to take all the fields defined between the first and last field (inclusive).
5. table is the name or alias of a table, synonym, or view, as declared in the TABLES section. This notation instructs the form compiler to build the screen record with all fields declared in the ATTRIBUTES section for the given table.

**Usage**

Screen records and screen arrays are defined with the SCREEN RECORD keywords in the INSTRUCTIONS section of a form specification file to name a group of fields.

**Screen records**

A screen record is a named group of fields that screen interaction statements of the program can reference as a single object.

A screen record is the form-counterpart of a RECORD variable in your program.

By establishing a correspondence between a set of screen fields (the screen record) and a set of program variables (typically a program record), you can pass values between the program and the fields of the screen record.

In many applications, it is convenient to define a screen record that corresponds to a row of a database table.

The elements of a screen record are associated by name to a form field and the corresponding form item tag in the layout section, through its definition in the ATTRIBUTES section:
Like the name of a screen field, the identifier of a screen record must be unique within the form, and its scope is restricted to when its form is open. Interactive statements can reference record-name only when the screen form that includes it is being displayed. The form compiler returns an error if record-name is the same as the name or alias of a table in the TABLES section.

SCHEMA myshop
LAYOUT
GRID
{
   Customer id: [f1 ]
   Name: [f2 ]
   Create date: [f3 ]
}
END

ATTRIBUTES
f1 = customer.cust_id;
f2 = customer.cust_name;
f3 = customer.cust_crea;
END

INSTRUCTIONS
SCREEN RECORD sr_customer
{
   customer.cust_id,
   customer.cust_name,
   customer.cust_crea
};
END

Default screen records
The form compiler builds default screen records that consist of all the screen fields linked to the same database table within a given form. A default screen record is automatically created for each table that is used to reference a field in the ATTRIBUTES section.

The components of the default record correspond to the set of display fields that are linked to columns in that table. The name of the default screen record is the table name (or the alias, if you have declared an alias for that table in the TABLES section).

For example, all the fields linked to columns of the "customer" table constitute a default screen record whose name is "customer".

Tip: To find out what default screen records are created by the form compiler, check the RecordView nodes produced in the .42f file.

If a form includes one or more FORMONLY fields, those fields constitute a default screen record with the name "formonly".
Screen arrays

A **screen array** is a screen record with a dimension, to handle a list of records in layout containers such as TABLE, TREE, SCROLLGRID, or static field lists.

A screen array is the form-counterpart of a DYNAMIC ARRAY OF RECORD variable in your program.

**Note:** Screen arrays are associated with list containers, by comparing the fields and columns used in their definitions. Therefore, it is not possible to define multiple screen arrays for the same list container.

Each row of a screen array is a screen record. Each column of a screen array consists of fields with the same field tag ([f1 ]) in the LAYOUT section.

Screen arrays are typically defined with SCREEN RECORD using a [size] specification:

```plaintext
SCREEN RECORD sa_custlist[20]
{
  customer.cust_id,
  customer.cust_name,
  customer.cust_crea
};
```

The [size] can be omitted: When not specified, the screen array automatically gets the size of the corresponding list container (TABLE, TREE, SCROLLGRID or static field list), as defined in the form layout.

**Important:** If the screen array size is 1, it becomes a simple screen record.

If the size is specified, it must be equal to the number of lines of the list container in the layout of the form.

For resizable list containers, when the size is specified in the SCREEN RECORD definition, it can be considered as the default number of rows. At runtime, the list container may be resized, and the screen array can be used in the program with a screen row index that is greater than the default size specified in the form file. For example:

```plaintext
INPUT cust_record.* FROM sa_custlist[arr_curr()].*
```

In this example, a TABLE container represents a set of fields organized in columns:

```plaintext
LAYOUT
TABLE
{
  [f1] | [f2] | [f3] |
  [f1] | [f2] | [f3] |
  [f1] | [f2] | [f3] |
  [f1] | [f2] | [f3] |
}
END
END
...
ATTRIBUTES
f1 = customer.cust_id;
f2 = customer.cust_name;
f3 = customer.cust_crea;
END
...
```

With the above TABLE container, when specified, the screen array size must be 4:

```plaintext
SCREEN RECORD sa_custlist[4]
{
  customer.cust_id,
  customer.cust_name,
  customer.cust_crea
};
```
Or, the screen array can be defined without a size:

```sql
SCREEN RECORD sa_custlist
{
    customer.cust_id,
    customer.cust_name,
    customer.cust_crea
};
```

**Using screen records and screen arrays in programs**

Screen records and screen arrays can display program records. If the fields in the screen record have the same sequence of data types as the columns in a database table, you can use the screen record to simplify operations that pass values between program variables and rows of the database.

Screen records are usually not referenced in programs within single record input statements, because program variable to form field binding is typically done by name with the `INPUT BY NAME` instruction:

```sql
DEFINE cust_rec RECORD LIKE customer.*
...
INPUT BY NAME cust_rec.*
...
```

Screen arrays are typically referenced in programs within interactive dialog controlling a list of records such as `DISPLAY ARRAY` and `INPUT ARRAY`. The current form must include that named screen array:

```sql
DEFINE cust_arr DYNAMIC ARRAY OF RECORD LIKE customer.*
...
DISPLAY ARRAY cust_arr TO sa_custlist.*
...
```

**Related concepts**

- **INSTRUCTIONS section** on page 1350
  The `INSTRUCTIONS` section is used to define screen arrays, non-default screen records, and global form properties.

- **Variable binding in INPUT** on page 1485

- **Form tags** on page 1254
  *Form tags* define layout elements inside a grid-based container.

  **Form tags**
  *Form tags* define layout elements inside a grid-based container.

  Form tags are place holders used inside a grid of the layout section, to define the position and the relation between form items.

  The syntax and purpose of a form tag depends on the type of form tag.

- **Layout tags**
  *Layout tags* define layout areas for containers inside the frame of a grid-based container.

**Syntax**

```html
<type [identifier]>
<content>
...
<>
```

1. *type* defines the kind of layout tag to be inserted at this position.
2. *identifier* references a form item definition in the `ATTRIBUTES` section, it must be unique, but is optional.
3. *content* defines other form items inside the layout tag.
4. The `(< >)` ending the layout tag body is optional.
Usage
A layout tag defines a layout region of a container, in the body frame of a `GRID` container.

While complex layout with nested frames can be defined with `HBOX` and `VBOX` containers, it is sometimes more convenient to define a form with a complex layout by using layout tags within a `GRID` container.

A layout tag has a type that defines the kind of container generated in the compiled form.

A layout tag is delimited by angle brackets (<>), and contains the tag type (G/GROUP, T/TABLE, etc) and an optional identifier.

Table 351: Types of layout tags

<table>
<thead>
<tr>
<th>Tag Type</th>
<th>Abbr.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GROUP</td>
<td>G</td>
<td>Defines a group box layout tag, resulting in the same presentation as the GROUP container.</td>
</tr>
<tr>
<td>TABLE</td>
<td>T</td>
<td>Defines a table view layout tag, resulting in the same presentation as the TABLE container.</td>
</tr>
<tr>
<td>TREE</td>
<td>N/A</td>
<td>Defines a tree-view list view layout tag, resulting in the same presentation as the TREE container.</td>
</tr>
<tr>
<td>SCROLLGRID</td>
<td>S</td>
<td>Defines a scrollable grid layout tag, resulting in the same presentation as the SCROLLGRID container.</td>
</tr>
</tbody>
</table>

The details of the layout tag definition are specified in the `ATTRIBUTE` section. Layout tags must be identified by an item tag name. In the example, the layout tag named "g1" is defined in the `ATTRIBUTE` section with the `GROUP` form item type to set the name and text of the element:

```plaintext
LAYOUT
GRID
{
<GROUP g1>
[text1]
[
[
[
<
>
]
]
}
END
END
ATTRIBUTES
GROUP g1:group1, TEXT="Description";
TEXTEDIT text1=FORMONLY.text1;
END
```

In the `ATTRIBUTES` section, the group element definition is bound to the layout item tag with the name "g1". The AUI element name will be "group1". This name can then be used in programs to manipulate the group element, for example with `ui.Form.setElementHidden("group1",1)`.

The layout region is a rectangle, in which the width is defined by the length of the layout tag, and the height by a closing tag (< >). In the example, the layout region is defined by the layout tag named "group1".

```plaintext
<GROUP group1 >
```
Form items must be placed inside the layout region. The [ ] square brackets are not part of the form item's width and can be place at the same X position as the layout tag delimiters:

```xml
<Group group1>
  Item:  [f001]
  Quantity: [f002]
  Date:   [f003]
</Group>
```

The [ ] square brace delimiters are not counted to define the width of an item tag. The width of the item is defined by the number of character between the square brackets. Thus, this layout is valid and can be compiled:

```xml
<Group group1>
  [f001]
  [f002]
  Static labels must fit!!
</Group>
<Table table1>
  [colA | colB ]
  [colA | colB ]
  [colA | colB ]
  [colA | colB ]
</Table>
```

You can place several layout tags on the same layout line in order to split the frame horizontally. This example defines six layout regions (four group boxes and two tables):

```xml
<Group group1> <Group group2> <Group group4>
  FName:  [f001] Phone: [f004] [f012]
  LName:  [f002] EMail: [f005]
</Group>
<Group group3>
  [f010]
</Group>
<Table table1> <Table table2>
  [c11  |c12  |c13] [c21  |c22]
  [c11  |c12  |c13] [c21  |c22]
  [c11  |c12  |c13] [c21  |c22]
  [c11  |c12  |c13] [c21  |c22]
</Table>
```

The < > closing layout tag is optional. When not specified, the end of the layout region is defined by the underlying layout tag or by the end of the current grid. However, the ending tag must be specified if the form compiler cannot detect the end of the layout region. This is usually the case with group layout tags.

In the example, the table does not need an ending layout tag because it is defined by the starting tag of the group, but the group needs an ending tag otherwise it would include the last field (field3). Additionally, if field3 had a different size, the form compiler would raise an error because the group and the last field geometry would conflict.
It is possible to mix container layout tags with singular form items. You typically put form items using a large area of the form, such as IMAGE fields or TEXTEDIT fields. The [ ] square brace delimiters are not used to compute the size of the singular form items:

```
<GROUP group1>
  FName: [f001]
  LName: [f002]
<>
[textedit1]
```

Table layout tags can be embedded inside group layout tags:

```
<GROUP group1>
  <TABLE table1>
    [colA | colB]
    [colA | colB]
    [colA | colB]
    [colA | colB]
  </TABLE>
</GROUP>
```

Hbox or vbox containers with splitter are automatically created by the form compiler in these conditions:

- **Hbox** is created when two or more stretchable elements are stacked side by side and touch each other (no space between).
- **Vbox** is created when two or more stretchable elements are stacked vertically and touch each other (no space between).

Stretchable elements are containers such as TABLE containers, or form items like IMAGE fields with the STRETCH attribute.

No hbox or vbox object will be created if the elements are in a SCROLLGRID container.

This example defines two tables stacked vertically, generating a VBox with splitter (note that ending tags are omitted):

```
<TABLE table1>
  [colA | colB]
  [colA | colB]
  [colA | colB]
  [colA | colB]
</TABLE>
<TABLE table2>
  [colC | colD]
  [colC | colD]
```

In this example, the layout defines two stretchable TEXTEDIT fields placed side by side which would generate an automatic hbox with splitter. To make both textedits touch you need to use a pipe delimiter in between:

```
[textedit1 | textedit2]
```

The next layout example would make the form compiler create an automatic vbox with splitter to hold table2 and textedit1, plus an hbox with splitter to hold table1 and the first VBox (We must use a pipe character to delimit the end of colB and textedit1 so that both tables can be placed side by side):

```
<TABLE table1>
<TABLE table2>
```
If you want to avoid automatic hbox or vbox with splitter creation, you must add blanks between elements:

< TABLE table1 >  < TABLE table2 >
[ colA | colB ]  [ colC | colD ]
[ colA | colB ]  [ colC | colD ]
[ colA | colB ]  [ colC | colD ]
[ colA | colB ]
[ colA | colB ]  [ textedit1 ]
[ colA | colB ]
[ colA | colB ]

Examples

The typical OK/Cancel window:

LAYOUT
GRID
{
  < GROUP g1 >
  [ com ]
  < >
  [ :bok | bno ]
}
END
END
ATTRIBUTES
LABEL com: comment;
BUTTON bok: accept;
BUTTON bno: cancel;
...

This example shows multiple uses of layout tags:

LAYOUT
GRID
{
  < GROUP g1 >  < GROUP g2 >
  Ident: [ f001 ] [ f002 ] [ text1 ]
  Addr: [ f003 ] [ ]
  < >
  < GROUP g3 >
  [ text2 ]
  [ ]
  < >
  < TABLE t1 >
  Num Name State Value
  [ col1 | col2 | col3 | col4 ]
  [ col1 | col2 | col3 | col4 ]
  [ col1 | col2 | col3 | col4 ]
  [ col1 | col2 | col3 | col4 ]
  <
}
END
END
ATTRIBUTES
**Related concepts**

Form rendering on page 1415
The section explains the layout rules to render forms on graphical front-ends.

Grid-based layout on page 1418
A form file can define a grid-based layout within a tree of layout items.

GRID item type on page 1276
Defines a layout area based on a grid of cells.

Automatic HBoxes and VBoxes on page 1422
Horizontal and vertical boxes are added automatically when stretchable elements are used.

**Item tags**

*Item tags* define the position and size in a grid-based container.

An item tag defines the position and size of a simple form item in a *grid-area* of a GRID or SCROLLGRID container. Form items defined with item tags are leaf nodes in the structure of a form definition, such as a form field (meaning that it is not a container form item).

**Syntax**

```
[identifier [-] [| ... |]]
```

1. *identifier* references a form item definition in the ATTRIBUTES section.
2. The optional – dash defines the real width of the element.
3. The | pipe can be used as item tag separator (equivalent to ] []).

**Usage**

An item tag is delimited by square brackets ([ ]) or pipes (|) and contains an identifier used to reference the description of the form item in the ATTRIBUTES section. In this example, the identifier of the form item is "f01", and the form item type is BUTTONEDIT:

```plaintext
LAYOUT
GRID
{
  ...
  [f01            ]
  ...
}
END
...
ATTRIBUTES
BUTTONEDIT f01 = customer.cust_name, ACTION=zoom;
...
```

Each item tag must be indicated by left and right delimiters to show the length of the item and its position within the container layout. Both delimiters must appear on the same line. You must use left and right square brackets ([ ]) to delimit item tags. The number of characters and the delimiters define the width of the region to be used by the item:

```plaintext
GRID
{
  Name: [f001]
}
```
The form item position starts after the open square brace and the length is defined by the number of characters between the square brackets. The following example defines a form item starting at position 3, with a length of 2:

```
GRID
{ 1234567890  [f1]  }
END
```

By default, the real width of the form item is defined by the number of characters used between the tag delimiters.

For some special items like BUTTONEDIT, COMBOBOX and DATEEDIT, the width of the field is adjusted to include the button. The form compiler computes the width as: $width = nb\text{chars} - 2$ if $nb\text{chars} > 2$:

```
GRID
{ 1234567  [f1]  -- this EDIT gets a width of 7
[f2]  -- this BUTTONEDIT gets a width of 5 (7-2)
}
END
```

If the default width generated by the form compiler does not fit, the – dash symbol can be used to define the real width of the item. In this example, the form item occupies 7 grid cells, but gets a real width of 5 (this means for an EDIT field, you are able to enter 5 characters):

```
GRID
{ 1234567  [f1   - ]  }
END
```

To make two items appear directly next to each other, you can use the pipe symbol (|) to indicate the end of the first item and the beginning of the second item:

```
GRID
{  Info:  [f001    |f002             |f003    ]  }
END
```

If you need the form to support items with a specific height (more than one line), you can specify multiple-segment item tags that occupy several lines of a grid-area. To create a multiple-segment item, repeat the item tag delimiters without the item identifier on successive lines:

```
GRID
{  Multi-segment: [f001
[    ]
[    ]
[    ]
[    ]
}
END
```

The notation applies to the new LAYOUT section only. For backward compatibility (when using a SCREEN section), multiple-segment items can be specified by repeating the identifier in sub-lines.
If the same item tag (that is with the same identifier) appears more than once in the layout, it defines a column of a screen array (also known as "Matrix").

**Note:** While all repeated item-tags apply to a screen array definition, you must distinguish static screen array columns ("Matrix") defined in a GRID container, from TABLE (or TREE) column definitions and SCROLLGRID row templates.

Example:

```plaintext
GRID
{
  Single-line static screen array:
  [f001 ] [f002 ] [f003 ]
  [f001 ] [f002 ] [f003 ]
  [f001 ] [f002 ] [f003 ]
  [f001 ] [f002 ] [f003 ]
}
END
...
ATTRIBUTES
f001 = FORMONLY.field1;
f002 = FORMONLY.field2;
f003 = FORMONLY.field3;
END
INSTRUCTIONS
SCREEN RECORD my_screen_array (FORMONLY.*);
END
```

Multi-line list of fields can be defined as follows:

```plaintext
GRID
{
  Multi-line static screen array:
  [f001 ] [f002]
  [f003 ]
  [f001 ] [f002]
  [f003 ]
  [f001 ] [f002]
  [f003 ]
  [f001 ] [f002]
  [f003 ]
}
END
```

**Related concepts**

**Form rendering** on page 1415
The section explains the layout rules to render forms on graphical front-ends.

**GRID container** on page 1308
Defines a layout area based on a grid of cells.

**ATTRIBUTES section** on page 1330
The ATTRIBUTES section describes properties of grid-based layout elements used in the form.

**Hbox tags**

*Hbox tags* group several item tags within the same horizontal layout box, inside a grid-based container (GRID).

An Hbox tag defines the position and size in a GRID container for an horizontal box containing several leaf form items. The elements in the Hbox tag can use additional alignment rules to get the required visual affect.
Syntax

\[
[ \text{element: \{\ldots\} } ]
\]

where \text{element} can be:

\[
\downarrow \text{identifier} \, \downarrow-\downarrow \downarrow \text{string-list} \downarrow
\]

where \text{string-list} is:

\[
\downarrow \text{string-literal} \, \downarrow \text{spacer} \, \downarrow \{\ldots\}
\]

1. \text{identifier} references a form item definition in the \text{ATTRIBUTES} section.
2. The optional – dash defines the real width of the element.
3. \text{string-list} is a combination of \text{string-literals}.
4. \text{string-literal} is a quoted text that defines a static label.
5. \text{spacer} is one or more blanks that define an invisible element that expands automatically.
6. The colon is the delimiter for Hbox tag elements.

Usage

Hbox tags are provided to control the alignment of form items in a grid. Hbox tags allow you to stack form items horizontally without the elements being influenced by elements above or below. In an Hbox, you can mix form fields, static labels and spacers. A typical use of the hbox is to have zip-code / city form fields side by side with predictable spacing in between.

An hbox tag is delimited by square brackets ([ ]) and must contain at least one \text{string-list} or an \text{identifier} preceded or followed by a colon (:). A \text{string-list} is a combination of \text{string-literals} (quoted text) and \text{spacers} (blank characters). The delimiter for hbox tag elements is the colon.

Hbox tags are not allowed for fields of \text{Screen records / arrays} on page 1251; a form compiler error is generated. The client needs a matrix element directly in a grid or a scrollgrid to perform the necessary positioning calculations for the individual fields.

The following example shows simple hbox tags:

```
GRID
{
    ["Customer info:":    ]
    [f001       :         ]
    [    :f002              ]
    ["Name: "  :f003      ]
}
END
```

In this example:

1. The first hbox tag contains two elements: a static label and a spacer.
2. The second hbox tag contains two elements: a form field and a spacer.
3. The third hbox tag contains two elements: a spacer and a form field.
4. The fourth hbox tag contains two elements: a static label and a form field.

An hbox tag defines the position and width (in grid cells) of several form items grouped inside an horizontal box. The position and width (in grid cells) of the horizontal box is defined by the square brackets ([ ]) delimiting the hbox tag.

When using an \text{identifier}, you define the position of a form item which is described in the \text{ATTRIBUTES} section. When using a \text{string-list}, you can define static labels and/or spacers. The following example defines an hbox tag
generating 7 items (a static label, a spacer, a form item identified by num, a spacer, a static label, a spacer and a form item identified by name):

```
GRID
{
  ["Num:"  :num  :"Name:"  :name        ]
}
END
```

A *spacer* is an invisible element that automatically expands. It can be used to align elements left, right or center in the hbox. The following example defines 3 hboxes with the same width. Each hbox contains one field. The first field is aligned to the left, the second is aligned to the right and third is centered:

```
GRID
{
  [left :              ]
  [         :right     ]
  [     :centered:     ]
}
END

ATTRIBUTES
LABEL left: label1, TEXT="LEFT";
LABEL right: label2, TEXT="RIGHT";
LABEL centered: label3, TEXT="CENTER";
END
```

When you use string literals, the quotes define where the label starts and stops. If there is free space after the quote that ends the label, then it is filled by a spacer. Consider this example:

```
GRID
{
  [    :"Label1"     ]
  [         :"Label2"]
}
END
```

In this example:

1. The first line contains a spacer, followed by the static label, followed by another spacer. The quotation marks end the string literal; a colon is not required to delimit the label from the final spacer.
2. The second line contains a spacer, followed by the static label. Because there is no empty space between the end of the static label and the closing bracket of the hbox Tag ( )

A typical use of hbox tags is to vertically align some form items - that must appear on the same line - with one or more form items that appear on the other lines:

```
GRID
{
  Id:      [num  :"Name:"  :name        ]
  Address: [street                   ]
            [zip-code:city              ]
  Phones:  [phone       :fax           ]
}
END
```

In this example, the form compiler will generate a grid containing 7 elements (3 labels and 4 hboxes):

1. The label "Id:"
2. A first hbox defines 3 cells, where:
   - The field 'num' will occupy the cell (1,1),
• The label "Name:" will occupy the cell (2,1),
• The field 'name' will occupy the cell (3,1).
3. The label "Address:" will occupy cell (1,2),
4. A second hbox defines 1 cell, where:
• The field 'street' will occupy the cell (1,1).
5. A third hbox defines 2 cells, where:
• The field 'zip-code' will occupy the cell (1,1),
• The field 'city' will occupy the cell (2,1).
6. The label "Phones:" will occupy cell (1,4),
7. A fourth hbox defines 2 cells, where:
• The field 'phone' will occupy the cell (1,1),
• The field 'fax' will occupy the cell (2,1).

Inside an hbox tag, the positions and widths of elements are independent of other hboxes. It is not possible to align elements over hboxes. The position of items inside an hbox depends on the spacer and the real size of the elements. The following example does not align the items as you would expect, following the character positions in the layout definition:

```
GRID {
  ["Num:     " :fnum :        ]
  ["Name:    " :fname         ]
}
END
```

A big advantage in using elements in an hbox is that the fields get their real sizes from the .per definition. The following example illustrates the case:

```
GRID {
  MMMMM
  [f1   ]
  [f2 : ]
}
END
```

Here all items will occupy the same number of grid columns (5). The MMMMM static label will have the largest width and define the width of the 5 grid cells. The first field is defined with a normal item tag, and expands to the width of the 5 grid cells. The line 5 defines an hbox that will expand to the size of the 5 grid cells, according to the static label, but its child element - the field f2 - gets a size corresponding to the number of characters used before the 'colon (that is 3 characters).

If the default width generated by the form compiler does not fit, the – dash symbol can be used to define the real width of the item. In this example, the hbox tag occupies 20 grid cells, the first form item gets a width of 5, and the second form item gets a width of 3:

```
GRID {
  12345678901234567890
  [f1   – :f2 – :   ]
}
END
```

The – dash size indicator is especially useful in BUTTONEDIT, DATEEDIT, and COMBOBOX form fields, for which the default width computed by the form compiler may not fit.
In this example, a static label is positioned above a TEXTEDIT field. The label will be centered over the TEXTEDIT field, and will remain centered as the field expands or contracts with the resizing of the window.

```plaintext
GRID
{
  [ :"label": ]
  [textedit   ]
}
END

ATTRIBUTES
  TEXTEDIT textedit = formonly.textedit, STRETCH=BOTH;
END
```

**Related concepts**

- **Grid-based layout** on page 1418
  A form file can define a grid-based layout within a tree of layout items.

- **ATTRIBUTES section** on page 1330
  The ATTRIBUTES section describes properties of grid-based layout elements used in the form.

- **Layout tags** on page 1254
  Layout tags define layout areas for containers inside the frame of a grid-based container.

- **Widget size within hbox tags** on page 1430

### External form inclusion

Form inclusion allows to reuse the same form part in different forms.

In some cases, application forms can become very complex, or can have a common layout part that repeats across forms. In such case, some parts of the form can be defined in an external .per file, that will be included in the final form by using the `FORM clause` inside the LAYOUT section.

Furthermore, the external form parts can be controlled by a declarative dialog instruction that can be attached to any procedural dialog instruction, with the `SUBDIALOG clause` of DIALOG.

```plaintext
LAYOUT
  VBOX
    GRID g1
    {
      Customer information
      Name:  [f001                 ]
      ...
    }
  END
  FORM "orders"
END
END
```

**Related concepts**

- **The SUBDIALOG clause** on page 1600

### Boolean expressions in forms

Some form item definitions can include boolean expressions with a form file specific syntax.

**Syntax**

```
```
where \textit{bool-expr} is:

\begin{verbatim}
[NOT] field-tag
  = expression
  ! = expression
  <= expression
  >= expression
  < expression
  > expression
  IS [NOT] NULL
  [NOT] BETWEEN expression AND expression
  [NOT] MATCHES "string"
  [NOT] LIKE "string"
\end{verbatim}

1. \textit{field-tag} is the name of the current field tag in form line with the attribute definition.
2. \textit{expression} can be the a character string, numeric or date-time literal.

\textbf{Usage}

Some form specification file attributes such as \texttt{COLOR WHERE} require a boolean expression. These boolean expressions are different from the language boolean expressions, and have a limited syntax which is specific to the form files.

When a \textit{field-tag} is used in the boolean expression, the runtime system replaces \textit{field-tag} at runtime with the current value in the screen field and evaluates the expression.

\textbf{Example}

```
EDIT f001 = item.price,
COLOR=RED
  WHERE f001 >= 100 AND f001 < 1000;
```

\textbf{Related concepts}

- \texttt{Literals} on page 325
  Describes the syntax of literals (constant values) to be used in sources.

- \textbf{Form item types}
  The \textit{form item types} defines the purpose of form elements.
  - \textbf{BUTTON item type}
    Defines a push-button that can trigger an action.

  - \textbf{BUTTON item basics}
    The \texttt{BUTTON} form item type defines a standard push button with a label and/or an icon.

  - \textbf{Defining a BUTTON}
    The label of a \texttt{BUTTON} form item is defined with the \texttt{TEXT} attribute. The \texttt{COMMENT} attribute can be used to define a hint for the button. Consider using \texttt{localized strings} for these attributes.

    The picture is defined by the \texttt{IMAGE} attribute. Consider using \texttt{centralized icons} for button images.

```
BUTTON ...
  TEXT = %"common.button.text.ok",
  IMAGE = "accept",
  COMMENT = %"common.button.hint.ok";
```
BUTTON form items can inherit action default attributes, to avoid having to specify the TEXT, COMMENT and IMAGE attributes in all elements bound to the same action. For more details, see Configuring actions on page 1744.

Front-ends support different presentation and behavior options, which can be controlled by a STYLE attribute. For more details, see Style attributes common to all elements on page 1182 and Button style attributes on page 1185.

Detecting BUTTON action

A BUTTON form item acts as an action view for a dialog action, and is bound to the ON_ACTION handler by name. The action name can be prefixed with a sub-dialog identifier and/or a field name, to define a qualified action view:

```
-- Form file (grid layout)
BUTTON b1: print;

-- Program file:
ON ACTION print
    -- Execute code related to the print action
```

**Note:** When controlled by a COMMAND action handler in a DIALOG interactive instruction, form buttons can get the focus and thus be part of the tabbing list (TABINDEX attribute).

For more details, see Binding action views to action handlers on page 1762.

Where to use a BUTTON

A BUTTON form item can be defined in different ways:

1. With an item tag and a BUTTON item definition on page 1333 in a grid-layout container (GRID, SCROLLGRID and TABLE).
2. As a BUTTON stack item on page 1310 in a STACK container.

Related concepts

Item tags on page 1259
Item tags define the position and size in a grid-based container.

ATTRIBUTES section on page 1330
The ATTRIBUTES section describes properties of grid-based layout elements used in the form.

BUTTONEDIT item type

Defines a line-edit with a push-button that can trigger an action.

BUTTONEDIT item basics

The BUTTONEDIT form item defines an edit field that gets user input, with an additional push button that can fire an action.

This type of form field is typically used to open a secondary window, to let the user choose from a large list of items and set the field value.

Defining a BUTTONEDIT

The IMAGE attribute of a BUTTONEDIT form item defines the picture to be displayed on the button.

By default, the text editor of a BUTTONEDIT allows the user to change the field value. Use the NOTEDITABLE attribute to deny text modification. The field still gets the focus, and the action button remains active, if there is a corresponding action handler in the current dialog.

```
BUTTONEDIT ...
    IMAGE = "zoom",
    NOTEDITABLE;
```
The button of buttonedit form items can inherit action default attributes, to avoid having to specify the image attributes in all elements bound to the same action. For more details, see Configuring actions on page 1744.

Most of the attributes described in the edit item type on page 1274 can also be used with the buttonedit.

Front-ends support different presentation and behavior options, which can be controlled by a style attribute. For more details, see Style attributes common to all elements on page 1182 and Buttonedit style attributes on page 1186.

Detecting buttonedit button action

The button of a buttonedit form element acts as an action view for a dialog action, and is bound to the on action handler by the action attribute.

Important: The buttonedit button will be ghosted, if the field is not managed by a current dialog, or when the field is disabled. For example, if a menu defines the on action lookup handler and the buttonedit has action=lookup, the button will NOT be active, because the field is not active during a menu statement. The button is also ghosted when the field is controlled by a non-menu dialog, but is disabled with dialog.setfieldactive() or defined as noentry.

In the form, the action attribute of buttonedit defines the name of the action to be sent to the program when the user clicks on the button. In the program, the action handler is defined with an on action block:

```-- Form file: BUTTONEDIT ...
   ACTION = open_city_list;
-- Program file:
ON ACTION open_city_list
   -- Open the city pick-list```

The button action of a buttonedit can also be prefixed with a sub-dialog identifier and/or field name, to define a qualified action view to be used in conjunction with on action action-name infield field-name. However, for convenience, even if the action is not qualified with a field name, the runtime system considers that action as field-qualified, to make the buttonedit button always active.

For more details, see Binding action views to action handlers on page 1762, Field-specific actions (infield clause) on page 1767.

Where to use a buttonedit

A buttonedit form item can be defined in different ways:

1. With an item tag and a buttonedit item definition on page 1333 in a grid-layout container (grid, scrollgrid and table).
2. As a buttonedit stack item on page 1311 in a stack container.

Defining the widget size

In a grid-based layout, the size of a buttonedit widget is computed by the layout rules as described in widget size within hbox tags on page 1430.

In a stack-based layout, the widget will take the full width available in the parent container.

Field input length

In grid-based layout, the input length in a buttonedit fields is defined by the item tag and may need to get the scroll attribute. For more details, see input length of form fields on page 1716.

Related concepts

Stack container on page 1309
The STACK container holds stack items defining a logical alignment of form items.

GRID container on page 1308
Defines a layout area based on a grid of cells.

Defining action views in forms on page 1759
How to define action views that will fire action events.

CHECKBOX item type
Defines a boolean or three-state checkbox field.

CHECKBOX item basics
The CHECKBOX form item defines a field with a check box and a text label.

Defining a CHECKBOX
The TEXT attribute defines the label to be displayed near the check box. Consider using localized strings for this attribute.

The box shows a check mark when the form field contains the value defined in the VALUECHECKED attribute (for example: "Y"), and shows no check mark if the field value is equal to the value defined by the VALUEUNCHECKED attribute (for example: "N"). If you do not specify the VALUECHECKED or VALUEUNCHECKED attributes, they respectively default to TRUE (integer 1) and FALSE (integer 0).

By default, during an INPUT dialog, a CHECKBOX field can have three states:

- Grayed ( NULL value )
- Checked ( VALUECHECKED value )
- Unchecked ( VALUEUNCHECKED value )

If the field is declared as NOT NULL, the initial state can be grayed if the default value is NULL; once the user has changed the state of the CHECKBOX field, it switches only between checked and unchecked states.

During a CONSTRUCT, a CHECKBOX field always has three possible states (even if the field is NOT NULL), to allow the end user to clear the search condition:

- Grayed (No search condition)
- Checked (Condition column = VALUECHECKED value)
- Unchecked (Condition column = VALUEUNCHECKED value)

Front-ends support different presentation and behavior options, which can be controlled by a STYLE attribute. For more details, see Style attributes common to all elements on page 1182 and CheckBox style attributes on page 1187.

Detecting CHECKBOX modification
To inform the dialog immediately when the value changes, define an ON CHANGE block for the CHECKBOX field. The program can then react immediately to user changes in the field:

```-- Form file (grid layout)
CHECKBOX cb1 = order.ord_valid,
   ITEMS = ... ;
```

```-- Program file:
ON CHANGE ord_valid
   -- The checkbox field has been modified```

For more details, see Reacting to field value changes on page 1724.
Where to use a CHECKBOX

A CHECKBOX form item can be defined in different ways:

1. With an item tag and a CHECKBOX item definition on page 1335 in a grid-layout container (GRID, SCROLLGRID and TABLE).
2. As a CHECKBOX stack item on page 1312 in a STACK container.

COMBOBOX item type

Defines a line-edit with a drop-down list of values.

COMBOBOX item basics

The COMBOBOX form item defines a field that can open a list of possible values the end user can choose from.

Note: The COMBOBOX is best used for a short list of possible values (10 to 50, maximum).

Defining a COMBOBOX

The values of the drop-down list are defined by the `ITEMS` attribute. Define a simple list of values like

```
("A", "B", "C", "D", ... )
```

or a list of key/value pairs like in

```
((1, "Paris"), (2, "Madrid"), (3, "London"))
```

In the second case, the labels (city names) display depending on the key value (the city number) held by the field.

```
COMBOBOX ...
```

Consider using localized strings when defining key/value pairs in the combobox items:

```
COMBOBOX ...
   ITEMS=((1,%"cities.paris"),
          (2,%"cities.madrid"),
          (3,%"cities.london"));
```

The `INITIALIZER` attribute allows you to define an initialization function for the COMBOBOX. This function is invoked at runtime when the form is loaded, to fill the item list dynamically, for example with database records. It is recommended that you use the `TAG` attribute, so you can identify in the program the kind of COMBOBOX form item to be initialized. The initialization function name is converted to lowercase by `fglform`.

```
COMBOBOX ...
   TAG = "city", INITIALIZER = city_module.cmb_init;
```

If neither `ITEMS` nor `INITIALIZER` attributes are specified, the form compiler automatically fills the list of items with the values of the `INCLUDE` attribute, when specified. However, the item list will not automatically be populated with include range values (i.e. values defined using the `TO` keyword). The `INCLUDE` attribute can be specified directly in the form or indirectly in the schema files.

```
COMBOBOX ...
   INCLUDE= ("A", "B", "C", "D", "E");
```

During an INPUT dialog, a COMBOBOX field value can only be one of the values specified in the `ITEMS` attribute. If the field allows NULL values, a NULL item can be placed anywhere in the combobox item list, to satisfy end-user preferences:

```
COMBOBOX ...
   ITEMS=((NULL,"<Undefined>"),
          (1,"Red"),
          (2,"Yellow"),
          (3,"Green"));
```
During an INPUT dialog, if the field allows NULL values, and no NULL item is specified in the item list, a NULL item (with empty label) will be added automatically at the end of the item list, to let the end-user set the field value to null. If the field is defined with NOT NULL, no default NULL item will be added automatically.

During a CONSTRUCT dialog, selecting the list item corresponding to null will be equivalent to the = query operator, which will generate a "colname is null" SQL condition. The COMBOBOX field gets an additional 'empty' item (even if the field is NOT NULL), to let the user clear the search condition. Some front-ends allow to select multiple items to produce the SQL condition "colname in (value1, value2, ... )".

A common practice is to deny nulls with the NOT NULL attribute, and add a special item such as (0,"<Undefined>") to identify a non-specified-value:

```plaintext
COMBOBOX ...
   NOT NULL,
   ITEMS=((0,"<Undefined>"),
          (1,"Red"),
          (2,"Yellow"),
          (3,"Green");
```

Front-ends support different presentation and behavior options, which can be controlled by a STYLE attribute. For more details, see Style attributes common to all elements on page 1182 and ComboBox style attributes on page 1188.

Detecting COMBOBOX item selection

To inform the dialog when the value changes, define an ON CHANGE block for the COMBOBOX field. The program can then react immediately to user changes in the field:

```plaintext
-- Form file (grid layout)
COMBOBOX cb1 = customer.cust_city,
   ITEMS = ...
;

-- Program file:
ON CHANGE cust_city
   -- A new item was selected in the combobox list
```

For more details, see Reacting to field value changes on page 1724.

Where to use a COMBOBOX

A COMBOBOX form item can be defined in different ways:

1. With an item tag and a COMBOBOX item definition on page 1336 in a grid-layout container (GRID, SCROLLGRID and TABLE).
2. As a COMBOBOX stack item on page 1312 in a STACK container.

Defining the widget size

In a grid-based layout, the size of a COMBOBOX widget is computed from the SIZEPOLICY and SAMPLE attributes, and by following the layout rules as described in Widget size within hbox tags on page 1430.

In a stack-based layout, the widget will take the full width available in the parent container.

COMBOBOX on mobile devices

On mobile devices, COMBOBOX form items are best used for a short list of possible values that can be displayed on a single page; for example, 4 to 6 elements. When a list expands to more than one page, it is recommended that you use a BUTTONEDIT with a zoom, which you can improve with a search button to find an exact item or to reduce the list of items to scroll.
Related concepts

Filling a COMBOBOX item list on page 1735
The item list of COMBOBOX fields can be initialized at runtime.

DATEEDIT item type

Defines a line-edit with a calendar widget to pick a date.

DATEEDIT item basics

The DATEEDIT form item defines a field that can open a calendar to ease date input. To store the field value, use a DATE program variable with this form item.

Important: DATEEDIT fields are dedicated for DATE value input. Some front-ends (especially on mobile devices) deny data types different from DATE. If the front-end does not support the data type used for the DATEEDIT field, the runtime system will raise an error and stop the program. Consider testing your application with all types of front-ends.

Defining a DATEEDIT

The DATEEDIT form item type allows the user to edit date values with a specific widget for date input. A DATEEDIT field typically provides a calendar widget, to let the end user pick a date from it.

When using a DATE variable as recommended, with desktop front-ends, the format of DATEEDIT fields is by default defined by the DBDATE environment variable. Specific format can be defined with the FORMAT attribute, but it is recommended to use the default date formatting. On mobile platforms, the date format is defined by the device OS language settings.

On some front-end platforms, the native widget used for DATEEDIT fields may allow only strict year/month/day value input, and therefore cannot be used with a CONSTRUCT instruction, where it must be possible to enter search filters like ">=24/03/2014".

Front-ends support different presentation and behavior options, which can be controlled by a STYLE attribute. For more details, see Style attributes common to all elements on page 1182 and DateEdit style attributes on page 1190.

Detecting DATEEDIT calendar selection

To inform the dialog when a date is picked from the calendar widget, define an ON CHANGE block for the DATEEDIT field. The program can then react immediately to user changes in the field:

```plaintext
-- Form file (grid layout)
DATEEDIT del = order.ord_shipdate,
    NOT NULL;

-- Program file:
ON CHANGE ord_shipdate
    -- A new date value was picked from the calendar
```

For more details, see Reacting to field value changes on page 1724.

Where to use a DATEEDIT

A DATEEDIT form item can be defined in different ways:

1. With an item tag and a DATEEDIT item definition on page 1337 in a grid-layout container (GRID, SCROLLGRID and TABLE).
2. As a DATEEDIT stack item on page 1313 in a STACK container.
Defining the widget size

In a grid-based layout, the size of a DATEEDIT widget is computed by following the layout rules as described in Widget size within hbox tags on page 1430.

In a stack-based layout, the widget will take the full width available in the parent container.

Field input length

The input length in a DATEEDIT fields is defined by the (DATE) program variable. In a grid-based layout, define an item tag with 10 positions, to be able to display dates with 4 year digits. For more details, see Input length of form fields on page 1716.

Calendar configuration

A set of presentation style attributes for the DateEdit class can be used to customize the calendar. For example, you can define the icons of the button to open the calendar, the days off, the first day of the week, and the type of pop-up window for the calendar.

Related concepts

TIMEEDIT item type on page 1290
Defines a line-edit field with a clock widget to pick a time.

DATETIMEEDIT item type
Defines a line-edit with a calendar widget to pick a datetime.

DATETIMEEDIT item basics

The DATETIMEEDIT form item defines a field that can open a calendar to ease date-time input.

To store the field value, use a DATETIME YEAR TO MINUTE or DATETIME YEAR TO SECOND program variable with such form item.

Important: DATETIMEEDIT fields are dedicated for DATETIME value input. Some front-ends (especially on mobile devices) deny data types different from DATETIME. If the front-end does not support the data type used for the DATETIMEEDIT field, the runtime system raises an error and stops the program. Consider testing your application with all types of front-ends.

Defining a DATETIMEEDIT

The DATETIMEEDIT form item type allows the user to edit date-time values with a specific widget for date-time input. A DATETIMEEDIT field typically provides a calendar and clock widget, to let the end user pick a date and time from it.

The display and input precision (time part with or without seconds) of the DATETIMEEDIT widget depends on the front-end. On some platforms, native date-time editors do not handle the seconds. Furthermore, some front-ends (especially on mobile devices) do not allow data types different from DATETIME YEAR TO {_MINUTE|SECOND}_.

On some front-end platforms, the native widget used for DATETIMEEDIT fields may allow only strict date-time value input, and therefore cannot be used with a CONSTRUCT instruction, where it must be possible to enter search filters like ">= 2014-01-23 11:00".

When using a DATETIME variable as recommended, with desktop front-ends, the format of DATETIMEEDIT fields is defined by the DBDATE environment variable. On mobile platforms, the date format is defined by the device OS language settings.

Front-ends support different presentation and behavior options, which can be controlled by a STYLE attribute. For more details, see Style attributes common to all elements on page 1182.
**Detecting DATETIMEEDIT calendar selection**

To inform the dialog when a date-time is picked from the calendar widget, define an `ON CHANGE` block for the `DATETIMEEDIT` field. The program can then react immediately to user changes in the field:

```
-- Form file (grid layout)
DATETIMEEDIT dt1 = order.ord_shipdate,
    NOT NULL;

-- Program file:
ON CHANGE ord_shipdate
    -- A new date-time value was picked from the calendar
```

For more details, see Reacting to field value changes on page 1724.

**Where to use a DATETIMEEDIT**

A `DATETIMEEDIT` form item can be defined in different ways:

1. With an item tag and a `DATETIMEEDIT` item definition on page 1338 in a grid-layout container (`GRID`, `SCROLLGRID` and `TABLE`).
2. As a `DATETIMEEDIT` stack item on page 1313 in a `STACK` container.

**Defining the widget size**

In a grid-based layout, the size of a `DATETIMEEDIT` widget is computed by following the layout rules as described in Widget size within hbox tags on page 1430.

In a stack-based layout, the widget will take the full width available in the parent container.

**Field input length**

The input length in a `DATETIMEEDIT` fields is defined by the `(DATETIME)` program variable. In a grid-based layout, define an item tag with enough positions to be able to display dates with 4 year digits. For more details, see Input length of form fields on page 1716.

**Related concepts**

- **TIMEEDIT item type** on page 1290
  Defines a line-edit field with a clock widget to pick a time.

**EDIT item type**

Defines a simple line-edit field.

**EDIT item basics**

The `EDIT` form item defines a field to enter a single-line of text, for any data type.

This item type is typically used to define character string and numeric form fields.

**Defining an EDIT**

The `EDIT` item type can be used for any data type that can be converted to editable text.

To show a hint to the user when the field has the focus, use the `COMMENT` attribute.

If the field is mandatory for input, combine the `NOT NULL` with the `REQUIRED` attribute.

The value accepted for the field can be limited with the `INCLUDE` attribute.

To provide a default value, define the `DEFAULT` attribute for the field.

Use the `DOWNSHIFT` or `UPSHIFT` attributes to force the letter case during input.
Input can be hidden (for example for password fields), with the `INVISIBLE` attribute.

Especially for mobile devices, use the `KEYBOARDHINT` attribute to get a specific keyboard when entering values into the field.

Input completion proposals can be implemented with the `COMPLETER` attribute.

Front-ends support different presentation and behavior options, which can be controlled by a `STYLE` attribute. For more details, see Style attributes common to all elements on page 1182 and Edit style attributes on page 1194.

**Where to use an EDIT**

An EDIT form item can be defined in different ways:

1. With an item tag and a EDIT item definition on page 1338 in a grid-layout container (GRID, SCROLLGRID and TABLE).
2. As a EDIT stack item on page 1314 in a STACK container.

**Field input length**

In grid-based layout, the input length in an EDIT fields is defined by the item tag and may need to get the `SCROLL` attribute. For more details, see Input length of form fields on page 1716.

**Related concepts**

**BUTTONEDIT item type** on page 1267

Defines a line-edit with a push-button that can trigger an action.

**FOLDER item type**

Defines a layout area to hold folder pages.

**FOLDER item basics**

A FOLDER form item type groups folder pages together. Folder pages are defined with the PAGE form item.

**Defining an FOLDER**

The FOLDER form item is just a container for PAGE items.

```
FOLDER ...
  PAGE ...
  ...
  PAGE ...
  ...
```

**Where to use a FOLDER**

A FOLDER form item can be defined in different ways:

1. In a FOLDER container in a LAYOUT tree, within a grid-based layout.
2. As a FOLDER stack item, inside a STACK container, within a stack-based layout.

**FOLDER rendering**

Front-ends support different presentation and behavior options, which can be controlled by a `STYLE` attribute.

The rendering of the folder tabs can be controlled with the `position` style attribute.

When using the values "top", "left", "right" and "bottom", the folder is rendered as a classic folder/page view, with folder tabs at the required position. Default is "top".

When using the value "accordion", the folder pages can be collapsed and expanded.
Note: The "accordion" position is only supported by the GBC front-end.

When the position style attribute is "accordion", the collapserPosition style attribute can be used to define the position of the collapser icon.

For more details, see Folder style attributes on page 1195.

GRID item type
Defines a layout area based on a grid of cells.

GRID item basics
A GRID form item defines an area in the layout section to place children form items by X,Y position in layout cells.

Defining an GRID
The GRID container declares a formatted text block defining the dimensions and the positions of the form items contained in the grid.

You can specify the position of labels, form fields for data entry or additional interactive objects such as buttons.

A GRID container can hold static text, item tags, field tags, hbox tags, and layout tags to define other containers such as TABLE, TREE and SCROLLGRID.

A GRID can hold form items such as labels, fields, or buttons at a specific position. Form items are located with item tags in the grid layout area. You can use layout tags to place some type of containers inside a grid.

Front-ends support different presentation and behavior options, which can be controlled by a STYLE attribute. For more details, see Style attributes common to all elements on page 1182.

Where to use a GRID
A GRID form item can only be defined as a GRID container in a LAYOUT tree.

GRID layout definition
For more details about grid layout concept, see Grid-based layout on page 1418.

Related concepts
Layout items on page 1240
Layout items are containers with a body that can hold other form items, in a grid-based layout form.

GROUP item type
Defines a layout area to group other layout elements together.

GROUP item basics
A GROUP form item type groups other form items together, typically in a groupbox widget.

Defining an GROUP
The GROUP form item typically gets a TEXT attribute, to define the title of the group. Consider using localized strings for this attribute:

```
GROUP ...
   TEXT="customer.info";
```

Note: Depending on the front-end platform, a GROUP container may get a border, and a title. The border may also be rendered with a shadow effect.
Consider identifying group elements with a name, in order to manipulate the group during program execution. For example use the `ui.Form.setElementHidden()` method to hide or show groups in a form:

```
GROUP g1: g_cust_info, ... ;   -- grid-based layout
GROUP g_cust_info, ... ;       -- stack-based layout
```

Front-ends support different presentation and behavior options, which can be controlled by a `STYLE` attribute. For more details, see Style attributes common to all elements on page 1182 and Group style attributes on page 1196.

**Groups in grid-based layout**

In a `LAYOUT` tree with `GROUP` containers, if you want to include several children in a `GROUP`, you can add a `VBOX` or `HBOX` into the `GROUP`, to define how these form items are aligned.

**Note:** When defining a `GROUP` container, you cannot set the `GRIDCHILDRENINPARENT` attribute. This attribute makes sense only for a group item defined with a layout tag contained in a `GRID` area.

Consider using a group layout tag inside a `GRID` container, this layout specification technique is often more appropriate to define forms:

```
GRID
{
<G g1             ><G g2        >
[11  :f1          ][f4          ]
...<G g3                            >
...                          
```

**Groups in stack-based layout**

In a `STACK` container, `GROUP` form items are one of the base concepts used to put stack items together. For more details see Stacked group rendering on page 1434.

**Collapsible groups**

By default, groups are not collapsible.

Some front-ends (see Group presentation style attributes reference) support the `collapsible` presentation style attribute, to let end users expand/collapse `GROUP` elements in your forms.

**Note:** This feature is only available with the GBC front-end.

When a group is defined as collapsible, the `collapserPosition` style attribute can be used to define the position of the collapsor icon.

The `initiallyCollapsed` style attribute defines if the collapsible group is collapsed or expanded when the form is displayed. Values can be "yes", "no", "never" and "always".

For example:

```
<Style name="Group.mystyle">
    <StyleAttribute name="collapsible" value="yes" />
    <StyleAttribute name="collapserPosition" value="right" />
    <StyleAttribute name="initiallyCollapsed" value="yes" />
</Style>
```

**Note:** These style attributes are supported in both stack and grid layout.

For more details, see Group style attributes on page 1196.
Groups on mobile devices

On mobile devices, groups render depending on the platform standards:

- With GMA/Android™, groups are visualized by a simple separator under the group title. Complex layout construction is supported: groups in groups, groups in a grid, and so on.
- With GMI/iOS, the layout is limited by the platform GUI standards. The only visible grouping container element is a group. Groups within groups are not allowed. GMI enforces each form item as a member in a group. There can be group headers and footers, but no elements in between groups.

Where to use a GROUP

A GROUP form item can be defined in different ways:

1. As a GROUP container in a LAYOUT tree, within a grid-based layout.
2. As a <GROUP> layout tag with a GROUP item definition in the ATTRIBUTES section, within a grid-based layout.
3. As a GROUP stack item, inside a STACK container, within a stack-based layout.

HBOX item type

Defines a layout area to render child elements in horizontal direction.

HBOX item basics

The HBOX container automatically packs the contained elements horizontally from left to right.

Contained elements are packed in the order in which they appear in the LAYOUT section of the form file.

No decoration (border) is added when you use an HBOX container.

By combining VBOX and HBOX containers, you can define any alignment you choose.

Defining an HBOX

An HBOX is defined in a grid-based LAYOUT tree, as a container for other form items. It can for example be combined with a VBOX container, to pack form elements to be displayed in vertical and horizontal directions:

```
LAYOUT
  HBOX
    VBOX
      GROUP
        ...
      END
      TABLE
        ...
      END
    END
  END
END
```

Front-ends support different presentation and behavior options, which can be controlled by a STYLE attribute. For more details, see Style attributes common to all elements on page 1182 and HBox style attributes on page 1197.

Rendering an HBOX as a splitview

Some front-ends (see HBox presentation style attributes reference) allow to render HBOX elements as splitviews, when using the splitViewRendering style attribute:

```xml
<Style name="HBox.splitView"/>
```
The HBOX must contain only two elements such as a GRID and TABLE. If the width of the screen is large enough, both elements are displayed. When the width of the screen becomes too small, the HBOX is converted to a splitview displaying only one of the child elements at a time. The end user can then use platform-specific solutions (with a swipe gesture on mobile devices), to switch between the contained elements.

For more details, see Rendering an HBox as a splitview on page 1907.

Where to use an HBOX
An HBOX form item can only be defined as an HBOX container in a LAYOUT tree, within a grid-based layout.

Note: An HBOX cannot be used inside a STACK container.

Important: Do not confuse the HBOX form item with the concept of Hbox tag, which is similar purpose but only for form item tags like labels and form fields.

IMAGE item type
Defines an area that can display an image resource.

IMAGE item basics
The IMAGE item type defines an area where a picture resource can be displayed.

Defining an IMAGE
An IMAGE form item can be defined as a form field image or as a static image. Use a form field image when the content of the image will change often during program execution (for example, to display images from the database). Use a static image if the image remains the same during program execution.

Front-ends support different presentation and behavior options, which can be controlled by a STYLE attribute. For more details, see Style attributes common to all elements on page 1182 and Image style attributes on page 1198.

Form field IMAGE item
Use a form field image item to display values that change often during program execution, for example if the image is stored in the database.

The picture resource is defined by the value of the field.

The value can be changed by the program using the DISPLAY BY NAME / DISPLAY TO instruction, or just by changing the value of the program variable bound to the image field when using the UNBUFFERED mode in an interactive instruction.

When defining the IMAGE item in the form, use a field name to identify the element in programs:

```
-- Grid-based layout (ATTRIBUTES item definition)
IMAGE f001 = cars.picture, SIZEPOLICY=FIXED, STRETCH=BOTH;

-- Stack-based layout (STACK item)
IMAGE cars.picture, SIZEPOLICY=FIXED, Autoscale;
```

Static IMAGE item
Use a static image item to display an image that does not change during program execution, such as form decoration pictures and logos.
The resource of the image is defined by the IMAGE attribute; the item is not a form field. This kind of item is not affected by instructions such as CLEAR FORM or the DISPLAY TO instruction.

```
-- Grid-based layout (ATTRIBUTES item definition)
IMAGE img1: logo, IMAGE="fourjs.png", SIZEPOLICY=FIXED, STRETCH=BOTH;

-- Stack-based layout (STACK item)
IMAGE : logo, IMAGE="fourjs.png", SIZEPOLICY=FIXED, AUTOSCALE;
```

**Providing the image resource**

To display an image, the front-end needs the image data, which can be provided in different ways.

For example, you can specify an URL, a mapped icon, or a plain image file (centralized on the application server).

For more details about image resource specification, see [Providing the image resource](#) on page 1149.

**Detecting IMAGE clicks**

To inform the dialog immediately when an image was clicked, define the ACTION attribute in the IMAGE item, and implement the corresponding ON ACTION handler in the dialog:

```
-- Form file (grid layout)
IMAGE : logo, IMAGE="fourjs.png",
     ACTION=show_about;

-- Program file:
ON ACTION show_about
     -- The image was clicked
```

The program can then react immediately when the user selects the image element.

**Where to use IMAGE**

An IMAGE form item can be defined in different ways:

1. With an item tag and an IMAGE item definition on page 1340 in a grid-layout container (GRID, SCROLLGRID and TABLE).
2. As an IMAGE stack item on page 1316 in a STACK container.

**Defining the widget size**

The size of an IMAGE widget can be controlled in grid-based or stack-based layout by several attributes such as SIZEPOLICY, AUTOSCALE and STRETCH.

For more details about image sizing, see [Controlling the image layout](#) on page 1148.

**Related concepts**

- **Using images** on page 1147
  Describes how to use pictures in the forms of your application.

- **LABEL item type**
  Defines a simple text area to display a read-only value.

- **LABEL item basics**
  The LABEL form item defines a read-only text area.
Defining a LABEL

A LABEL form item can be defined as a form field image or as a static label. Use a form field label when the text changes often during program execution (for example, to display text from the database). Use a static label if the text remains the same during program execution.

Front-ends support different presentation and behavior options, which can be controlled by a STYLE attribute. For more details, see Style attributes common to all elements on page 1182 and Label style attributes on page 1198.

Form field LABEL item

Use a form field label item to display values that change often during program execution, for example if the text is stored in the database.

The label text is defined by the value of the field.

The value can be changed by the program using the DISPLAY BY NAME / DISPLAY TO instruction, or just by changing the value of the program variable bound to the label field when using the UNBUFFERED mode in an interactive instruction.

When defining the LABEL item in the form, use a field name to identify the element in programs:

```
-- Grid-based layout (ATTRIBUTES item definition)
LABEL f001 = cars.description;

-- Stack-based layout (STACK item)
LABEL cars.description;
```

Static LABEL item

Use a static label item to display text that does not change during program execution.

This kind of item is not affected by instructions such as CLEAR FORM or the DISPLAY TO instruction.

```
-- Grid-based layout (ATTRIBUTES item definition)
LABEL lab1: label1, TEXT="Name:";

-- Stack-based layout (STACK item)
LABEL : label1, TEXT="Name:";
```

Consider using localized strings to ease application internationalization:

```
LABEL ... 
    TEXT = %"label.customer.name";
```

Static labels display only character text values, and therefore do not follow any justification rule as form field labels.

Multi-line text in LABELs

In order to display label text on several lines, the text must contain \n line-feed characters:

```
LABEL lab1: label1, 
    TEXT="First line.\nSecond line."
```

Where to use a LABEL

A LABEL form item can be defined in different ways:

1. With an item tag and a LABEL item definition on page 1341 in a grid-layout container (GRID, SCROLLGRID and TABLE).
2. As a LABEL stack item on page 1317 in a STACK container.
Defining the widget size
The size of a LABEL widget can be controlled in grid-based or stack-based layout by using the SIZEPOLICY attribute.

By default (SIZEPOLICY=INITIAL), labels adapt their width to the initial text displayed by the element.

With static labels, the initial text is usually defined in the form file, or with a localized string, and the size does not need to adapt once the label is displayed.

When using form field labels, the same default rule applies. However, if the initial text displayed in the form field is NULL or smaller than other texts that will be displayed, the size of the label element will not adapt after the initial text is displayed; the label stays at the size of the initial displayed text.

Important: When using a form field label, make sure that the size of the label will be large enough to display all possible values. To control the label size, use the SIZEPOLICY=DYNAMIC attribute, or use SIZEPOLICY=FIXED and define the form item with a sufficient size in the LAYOUT section.

PAGE item type
Defines the content of a folder page.

PAGE item basics
A PAGE form item type groups other form elements together, to define a folder page of a parent FOLDER form item.

Defining an PAGE
A PAGE form item can only be a child of a FOLDER form item.

By default, PAGE form items are used to group elements for decoration only.

Note: Use the TABINDEX attribute of form fields inside the folder page, to define which field gets the focus when a folder page is selected.

The TEXT attributes defines the label of the folder page. Consider using localized strings for this attribute.

The IMAGE attribute can be used to specify which image to use as an icon.

Detecting folder page selection
If needed, you can use the ACTION attribute to bind an action to a folder page. When the page is selected, the program gets the corresponding action event.

Note: Binding a folder page to an ON ACTION handler with the ACTION attribute is supported to implement different singular dialog statements (INPUT) to handle each folder page. This was the old method to control folder pages. It is now recommended that you use a DIALOG statement, to control all folder pages simultaneously.

Bring a folder page to the top
To bring a folder page to the top, use the NEXT FIELD program instruction to give the focus to one of the active fields of the page, or use the ui.Form.ensureFieldVisible() method if the fields are disabled/unused, or use the ui.Form.ensureElementVisible() method if the page does not contain focusable form items.

For more details, see Giving the focus to a form element on page 1731.

Where to use a PAGE
A PAGE form item can be defined in different ways:

1. As a PAGE container in a LAYOUT tree, within a grid-based layout.
2. As a PAGE stack item, inside a STACK container, within a stack-based layout.
**PROGRESSBAR item type**
Defines a progress indicator field.

**PROGRESSBAR item basics**
The PROGRESSBAR form item defines a field that shows a progress indicator.

**Note:** Use a SMALLINT or INTEGER variable with a PROGRESSBAR form item. Larger types like BIGINT or DECIMAL are not supported.

**Defining a PROGRESSBAR**
The `VALUEMIN` and `VALUEMAX` attributes define respectively the lower and upper integer limit of the progress information. Any value outside this range will not be displayed. Default values are `VALUEMIN=0` and `VALUEMAX=100`.

Front-ends support different presentation and behavior options, which can be controlled by a `STYLE` attribute. For more details, see [Style attributes common to all elements](#) on page 1182 and [ProgressBar style attributes](#) on page 1201.

**Displaying PROGRESSBAR values**
The position of the progress bar indicator is defined by the value of the corresponding form field. The value can be changed by the program using the `DISPLAY TO` instruction, to set the value of the field, or by changing the program variable bound to the field when using the UNBUFFERED dialog mode.

Progress information is typically displayed during non-interactive program code. To show changes to the end user in this context, you need to use the `ui.Interface.refresh()` method to force a refresh. To provide the best feedback to the user, consider calling the `refresh()` method regularly but not too often, otherwise you will overload the network traffic and bring down the front-end component.

For example, if you have to process 1000 rows, define `VALUEMIN=0` and `VALUEMAX=1000` in the PROGRESSBAR item, and perform a refresh every 50 rows:

```plaintext
FOR row=1 TO 1000
    ... 
    IF (row MOD 50) == 0 THEN
        LET myprogbar = row
        CALL ui.Interface.refresh()
    END IF
END FOR
```

**Where to use a PROGRESSBAR**
A PROGRESSBAR form item can be defined in different ways:

1. With an item tag and a PROGRESSBAR item definition on page 1342 in a grid-layout container (GRID, SCROLLGRID and TABLE).
2. As a PROGRESSBAR stack item on page 1319 in a STACK container.

**Related concepts**
The buffered and unbuffered modes on page 1720
The buffered and unbuffered mode control the synchronization of program variables and form fields.

**RADIOGROUP item type**
Defines a mutual exclusive set of options field.

**RADIOGROUP item basics**
The RADIOGROUP form item defines a field that provides several options that the user can make a selection from. Checking one radio button unchecks any previously checked button within the same group.

**Defining a RADIOGROUP**
A RADIOGROUP defines a set of radio buttons where each button is associated with a value defined in the ITEMS attribute.

The text associated with each item value will be used as the label of the corresponding radio button, for example: ITEMS=((1,"Beginner"), (2,"Normal"), (3,"Expert")) will create three radio buttons with the texts Beginner, Normal and Expert, respectively.

```plaintext
RADIOGROUP ...
    ITEMS=((1,"Beginner"), (2,"Normal"), (3,"Expert"));
```

Consider using localized strings when defining key/value pairs in the radio group items:

```plaintext
RADIOGROUP ...
    ITEMS=((1, "skills.beginner"), (2, "skills.normal"), (3, "skills.expert"));
```

If the ITEMS attribute is not specified, the form compiler automatically fills the list of items with the values of the INCLUDE attribute, when specified. However, the item list will not automatically be populated with include range values (that is values defined using the TO keyword). The INCLUDE attribute can be specified directly in the form or indirectly in the schema files.

During an INPUT, a RADIOGROUP field value can only be one of the values specified in the ITEMS attribute. During a CONSTRUCT, a RADIOGROUP field allows all items to be unchecked (even if the field is NOT NULL), to let the user clear the search condition.

If one of the items is explicitly defined with NULL and the NOT NULL attribute is omitted, in INPUT, selecting the corresponding radio button sets the field value to null. In CONSTRUCT, selecting the radio button corresponding to null will be equivalent to the equals (=) query operator, which will generate a "colname is null" SQL condition.

Use the ORIENTATION attribute to define if the radio group must be displayed vertically or horizontally:

```plaintext
RADIOGROUP ...
    ITEMS=..., 
    ORIENTATION = HORIZONTAL;
```

Front-ends support different presentation and behavior options, which can be controlled by a STYLE attribute. For more details, see Style attributes common to all elements on page 1182 and RadioGroup style attributes on page 1201.

**Detecting RADIOGROUP item selection**
To inform the dialog when a value change, define an ONCHANGE block for the RADIOGROUP field. The program can then react immediately to user changes in the field:

```plaintext
-- Form file (grid layout)
RADIOGROUP rgl = user.user_skill,
```
ITEMS = ... ;

-- Program file:
ON CHANGE user_skill
   -- An new item was selected in the radiogroup

For more details, see Reacting to field value changes on page 1724.

**Where to use a RADIOGROUP**

A RADIOGROUP form item can be defined in different ways:

1. With an item tag and a RADIOGROUP item definition on page 1343 in a grid-layout container (GRID, SCROLLGRID and TABLE).
2. As a RADIOGROUP stack item on page 1320 in a STACK container.

**SCROLLGRID item type**

Defines a scrollable grid view widget.

*Note:* This topic covers the SCROLLGRID item type definition in form files. See also the chapter dedicated to scrollgrid view programming.

**SCROLLGRID item basics**

A SCROLLGRID form item type defines a grid to show a scrolling list of data records in a set of positioned form fields.

The SCROLLGRID basically acts like a TABLE container, in the sense of its function as a record list container.

By default, scrollgrids display a fixed number of rows, and can be configured to be resizable in height.

A SCROLLGRID is controlled by a DISPLAY ARRAY or INPUT ARRAY dialog, see List dialogs on page 1831 for more details about this type of dialogs.

**Defining a SCROLLGRID**

The SCROLLGRID form item defines a formatted list view to show a structured set of data records. It is bound to a screen array grouping form fields which define the list fields of the scrollgrid.

The screen array definition must have exactly the same number of fields as the SCROLLGRID form item. Use PHANTOM fields, if the number of record members in the program array exceeds the number of fields to be displayed in the SCROLLGRID container.

The SCROLLGRID form item declares a formatted text block, defining the dimensions and the position of the logical elements of a screen for a multi-record presentation.

A SCROLLGRID is similar to the GRID, that repeats on several "row-templates", in order to design a view for multiple-records that display with a vertical scrollbar.

Inside a SCROLLGRID, the same layout rules apply as in a GRID container.

Static labels can be used as in a regular GRID container. However, this prevents you from localizing the label text. Consider using LABEL fields with a TEXT attribute, instead of static text labels.

Front-ends support different presentation and behavior options, which can be controlled by a STYLE attribute. For more details, see Style attributes common to all elements on page 1182 and ScrollGrid style attributes on page 1202.

**Where to use a SCROLLGRID?**

Within a grid-based layout, a SCROLLGRID form item can be defined in different ways:

1. As a SCROLLGRID container in a LAYOUT tree.
2. As a <SCROLLGRID > layout tag with a SCROLLGRID item definition in the ATTRIBUTES section.
Defining the scrollgrid size and layout

The global size of a SCROLLGRID element is defined by its content, and the number of record rows displayed.

**Note:** When using a SCROLLGRID container, you cannot set the GRIDCHILDRENINPARENT attribute. This attribute makes sense only for a scrollgrid defined with a layout tag, within a GRID area.

By default, a scrollgrid is not resizable in height: The number of visible rows is defined by the number of row templates in the form layout. Use the WANTFIXEDPAGESIZE=NO attribute, to allow the scrollgrid to resize.

When using WANTFIXEDPAGESIZE=NO, a resizable scrollgrid is displayed with one single row, if no other element in the form layout makes the scrollgrid stretch implicitly. To force a default number of rows in stretchable scrollgrids, use the INITIALPAGESIZE attribute.

A usual pattern on the web is to render information as a responsive tile list, using tiles displayed in pages. Each tile will flow, depending on the container size.

To render stretchable scrollgrids as a paged responsive tile list, define the customWidget presentation style attribute to pagedScrollGrid. With this attribute, each scrollgrid row will be rendered as a tile (the page size of the scrollgrid defines the number of tiles in each page):

```xml
<Style name="ScrollGrid.paged" >
  <StyleAttribute name= "customWidget" value= "pagedScrollGrid" />
</Style>
```

For more details, see also Controlling scrollgrid rendering on page 1885.

**Related concepts**

**TABLE item type** on page 1288

Defines a list view widget.

**SLIDER item type**

Defines a slider form item.

**SLIDER item basics**

The SLIDER form item defines a field where the user can set a value in a given range, such as a typical audio volume control widget where you can grab the slider handle to change the value.

Use a SMALLINT or INTEGER variable with a SLIDER form item, larger types like BIGINT or DECIMAL are not supported.

**Defining a SLIDER**

A SLIDER field allows the user to move a handle along a horizontal or vertical groove and translates the handle's position into a value within the legal range.

The VALUEMIN and VALUEMAX attributes define respectively the lower and upper integer limit of the slider information. Any value outside this range will not be displayed. The step between two marks is defined by the STEP attribute. If VALUEMIN and/or VALUEMAX are not specified, they default respectively to 0 (zero) and 5.

The ORIENTATION attribute defines whether the SLIDER is displayed vertically or horizontally.

This item type is not designed for CONSTRUCT, as the user can only select one value.

Front-ends support different presentation and behavior options, which can be controlled by a STYLE attribute. For more details, see Style attributes common to all elements on page 1182.
Detecting SLIDER item selection

To inform the dialog when a value changes, define an ON CHANGE block for the SLIDER field. The program can then react immediately to user changes in the field:

```
-- Form file (grid layout)
SLIDER s1 = options.opts_volume,
    VALUemin=0, VALUemax=100;

-- Program file:
ON CHANGE opts_volume
    -- A value changed in the slider
```

For more details, see Reacting to field value changes on page 1724.

Where to use a SLIDER

A SLIDER form item can be defined in different ways:

1. With an item tag and a SLIDER item definition on page 1345 in a grid-layout container (GRID, SCROLLGRID and TABLE).
2. As a SLIDER stack item on page 1320 in a STACK container.

Related concepts

Binding variables to form fields on page 1715
Some dialogs need program variables to store form field values.

SPINEDIT item type

Defines a spin box widget to enter integer values.

SPINEDIT item basics

The SPINEDIT form item defines a field dedicated to numeric values. Depending on the front-end platform, the widget may provide buttons to increment/decrement the field value.

Use a SMALLINT or INTEGER variable with a SPINEDIT form item. Larger types like BIGINT or DECIMAL are not supported.

Defining a SPINEDIT

The increment between two values is defined by the STEP attribute:

```
SPINEDIT ...
    STEP = 5;
```

The VALUemin and VALUemax attributes define respectively the lower and upper integer limit of the spin-edit range. There is no default minimum or maximum value for the SPINEDIT widget.

Note: This widget is not designed for CONSTRUCT, as you can only enter an integer value.

Front-ends support different presentation and behavior options, which can be controlled by a STYLE attribute. For more details, see Style attributes common to all elements on page 1182.

Detecting SPINEDIT modification

To inform the dialog when a value changes, define an ON CHANGE block for the SPINEDIT field. The program can then react immediately to user changes in the field:

```
-- Form file (grid layout)
SPINEDIT s1 = options.opts_rate,
    VALUemin=0, VALUemax=100, STEP=5;
```
For more details, see Reacting to field value changes on page 1724.

Where to use a SPINEDIT

A SPINEDIT form item can be defined in different ways:

1. With an item tag and a SPINEDIT item definition on page 1345 in a grid-layout container (GRID, SCROLLGRID and TABLE).
2. As a SPINEDIT stack item on page 1321 in a STACK container.

Related concepts

Binding variables to form fields on page 1715
Some dialogs need program variables to store form field values.

TABLE item type

Defines a list view widget.

Note: This topic covers the TABLE item type definition in form files. See also the chapter dedicated to table view programming.

TABLE item basics

A TABLE form item type defines a list view to show a scrolling list of data records in a set of columns.

A TABLE is controlled by a DISPLAY ARRAY or INPUT ARRAY dialog, see List dialogs on page 1831 for more details about this type of dialogs.

Defining a TABLE

The TABLE form item defines a list view widget to show a set of data records. It is bound to a screen array grouping form fields which define the columns of the table.

The screen array definition must have exactly the same number of columns as the TABLE form item. Use PHANTOM fields, if the number of record members in the program array exceeds the number of columns to be displayed in the TABLE container.

Column titles can be defined with static labels in the TABLE layout. However, this prevents you from localizing the column text. Consider using the TITLE attribute in the form field definition corresponding to the table columns.

A TABLE container can define AGGREGATE fields to display summary information for columns.

Front-ends support different presentation and behavior options, which can be controlled by a STYLE attribute. For more details, see Style attributes common to all elements on page 1182 and Table style attributes on page 1204.

Where to use a TABLE

A TABLE form item can be defined in different ways:

1. As a TABLE container in a LAYOUT tree, within a grid-based layout.
2. As a <TABLE > layout tag with a TABLE item definition in the ATTRIBUTES section, within a grid-based layout.
3. As a TABLE stack item, inside a STACK container, within a stack-based layout.

Defining the TABLE layout and size

The number of field columns composing the TABLE container in the form layout defines the initial width of the table view.
The width of a TABLE container is by default defined by the number of visible columns in its layout, and the initial height is defined by the number of rows. To specify explicitly the width and height of a table, use the WIDTH and HEIGHT attributes.

By default, a table is resizable in width and height. To force a TABLE to keep the height defined in the form file, use the WANTFIXEDPAGESIZE attribute.

Table columns can be moved around, their width can be adapted, they can be hidden/shown, and can be selected to sort the record list automatically. To turn off these features, use respectively the UNMOVABLECOLUMNS, UNSIZABLECOLUMNS, UNHIDABLECOLUMNS and UNSORTABLECOLUMNS attributes.

Related concepts
Controlling table rendering on page 1858
Table rendering can be controlled by the use of presentation styles and table attributes.
SCROLLGRID item type on page 1285
Defines a scrollable grid view widget.
ATTRIBUTES section on page 1330
The ATTRIBUTES section describes properties of grid-based layout elements used in the form.

TEXTEDIT item type
Defines a multi-line edit field.

TEXTEDIT item basics
The TEXTEDIT form item defines a text input field with multiple lines. This type of element is typically used to handle large text values such as comments or addresses that would not fit in a single-line edit field.

Use a VARCHAR(N) or STRING variable to hold the data for a TEXTEDIT form item.

Defining a TEXTEDIT
Use the SCROLLBARS attribute to define vertical and/or horizontal scrollbars for the TEXTEDIT form field. By default, when not specifying this attribute, TEXTEDIT fields get a vertical scrollbar.

The STRETCH attribute can be used to force the TEXTEDIT field to stretch when the parent container is re-sized. Values can be NONE, X, Y or BOTH. By default, this attribute is set to NONE for TEXTEDIT fields.

Front-ends support different presentation and behavior options, which can be controlled by a STYLE attribute. For more details, see Style attributes common to all elements on page 1182 and TextEdit style attributes on page 1212.

TAB and RETURN
By default, when the focus is in a TEXTEDIT field, the Tab key moves to the next field, while the Return key adds a newline (ASCII 10) character in the text.

To control the user input when the Tab and Return keys are pressed, specify the WANTTABS and WANTNORETURNS attributes.

With WANTTABS, the Tab key is consumed by the TEXTEDIT field, and a Tab character (ASCII 9) is added to the text. The user can still jump out of the field with the Shift-Tab combination.

With WANTNORETURNS, the Return key is not intercepted or consumed by the TEXTEDIT field, and the action corresponding to the Return key is triggered. The user can still enter a newline character with Shift-Return or Ctrl-Return.

Where to use a TEXTEDIT
A TEXTEDIT form item can be defined in different ways:
1. With an item tag and a TEXTEDIT item definition on page 1347 in a grid-layout container (GRID, SCROLLGRID and TABLE).
2. As a **TEXTEDIT** stack item on page 1323 in a **STACK** container.

**Defining the widget size**

In a grid-based layout, the rendering of the **TEXTEDIT** widget can be controlled with the **SIZEPOLICY**, **STRETCH** and **SCROLLBARS** attributes. GDC and GBC desktop front-ends implement only the **SIZEPOLICY=FIXED** behavior, where the size of the **TEXTEDIT** is defined by the height of the element in the form. With GMI and GMA mobile front-ends, the default is **SIZEPOLICY=DYNAMIC**, to adapt the **TEXTEDIT** size to its content. Use **SIZEPOLICY=FIXED** with GMA/GMI, to keep the width/height defined in the form file.

In a stack-based layout, the **TEXTEDIT** widget always adapts its height to the content/value of the field. You can control the minimum height of the **TEXTEDIT** widget with the **HEIGHT** attribute. If the field content is null and the **HEIGHT** attribute is not defined, the minimum size defaults to one line.

**Field input length**

By default, the input length in **TEXTEDIT** fields is defined by the program variable.

There is no need to define the **SCROLL** attribute, except you explicitly specify **SCROLLBARS=NONE** (in a grid-based layout).

**Note:** When specifying **SCROLLBARS=NONE** (in a grid-based layout), the **TEXTEDIT** field will limit the maximum input length to the number of cells defined by the **screen item tag**.

For more details about the **SCROLL** attribute, see **Input length of form fields** on page 1716.

**Making the **TEXTEDIT** read-only**

Use the **NOTEDITABLE** attribute to prevent text modification by the user. This attribute is typically used to display a large piece of text that the user is not required to modify (for example, to show the content of a log file). Yet, the focus can still go to the field, if it is enabled.

**Rich Text HTML support**

Some front-ends can also support different text formatting based on a style attribute. You can for example display and input **HTML** content in a **TEXTEDIT** with the Genero Desktop Client. When this feature is enabled, the **TEXTEDIT** supports rich text editing. Depending on the front-end, different formatting options are available (bold, font size, and so on) and can be controlled using an integrated toolbox.

**Note:**

- Each front-end uses its own technology to provide **HTML** support in **TEXTEDIT** fields. The **HTML** representation may vary between front-ends. As a result, the same **HTML** content may display in a different way on another front-end.
- When using rich text, the **FGL_DIALOG_SETCURSOR()** and **FGL_DIALOG_SETSELECTION()** functions must be called carefully. Because of the rich text format, having a corresponding cursor position / selection between displayed text and HTML representation may be difficult, especially in the case of hidden parts.

**Related concepts**

The **fglrichtext web component** on page 1972
The **fglrichtext** built-in web component implements an HTML text editor.

**TIMEEDIT item type**

Defines a line-edit field with a clock widget to pick a time.

**TIMEEDIT item basics**

The **TIMEEDIT** form item defines a field that allows the user to edit 24H time values, or time duration (intervals), with a specific clock widget for time input.
To store TIMEEDIT field values, consider using the appropriate DATETIME HOUR TO MINUTE or DATETIME HOUR TO SECOND data type depending on the target front-end.

**Important:**

The display and input precision (with or without seconds) of the TIMEEDIT widget depends on the front-end. On some platforms, native time editors do not handle the seconds. Furthermore, some front-ends (especially on mobile devices) do not allow data types different from DATETIME HOUR TO \{MINUTE|SECOND\}. If the front-end does not support the data type used for the TIMEEDIT field, the runtime system will raise an error and stop the program. Consider testing your application with all types of front-ends.

On some front-ends, TIMEEDIT fields can also be used to handle INTERVAL values of the class HOUR TO \{MINUTE|SECOND\}, in order to input a time duration. Note however that in most cases the time interval pickers are limited to 24H hours and allow only positive values. As result, not all values allowed in an INTERVAL HOUR TO MINUTE variable (such as -86 hours 23 minutes) can be displayed by such widgets.

**Defining a TIMEEDIT**

No specific attribute is needed to define the rendering and behavior of a TIMEEDIT field. Common data validation attributes such NOT NULL, REQUIRED, DEFAULT are allowed.

**Note:** The time display format is automatically taken from the front-end platform settings. For example, time values can display in the 0-12 hour clock format (with AM/PM indicators), or in the 0-24 hour clock format.

On some front-end platforms, the native widget used for TIMEEDIT fields usually allows only strict hh:mm[:ss] time value input, and therefore cannot be used with a CONSTRUCT instruction, where it must be possible to enter search filters like ">=11:00".

Front-ends support different presentation and behavior options, which can be controlled by a STYLE attribute. For more details, see Style attributes common to all elements on page 1182.

**Detecting TIMEEDIT modification**

To inform the dialog when a date is picked from the clock widget, define an ON CHANGE block for the TIMEEDIT field. The program can then react immediately to user changes in the field:

```plaintext
-- Form file (grid layout)
TIMEEDIT del1 = order.ord_shiptime,
    NOT NULL;

-- Program file:
ON CHANGE ord_shiptime
    -- A new time value was picked from the clock widget
```

For more details, see Reacting to field value changes on page 1724.

**Where to use a TIMEEDIT**

A TIMEEDIT form item can be defined in different ways:

1. With an item tag and a TIMEEDIT item definition on page 1348 in a grid-layout container (GRID, SCROLLGRID and TABLE).
2. As a TIMEEDIT stack item on page 1323 in a STACK container.

**Field input length**

The input length in a TIMEEDIT fields is defined by the (DATETIME) program variable. In a grid-based layout, define an item tag wide enough to fit all time value digits (5 positions for HH:MM, 7 positions for HH:MM:SS). For more details, see Input length of form fields on page 1716.

**Related concepts**

DATEEDIT item type on page 1272
Defines a line-edit with a calendar widget to pick a date.

**DATETIMEEDIT item type** on page 1273
Defines a line-edit with a calendar widget to pick a datetime.

**TREE item type**
Defines a tree view widget.

**Note:** This topic covers the TREE item type definition in form files. See also the chapter dedicated to tree view programming.

**TREE item basics**
A TREE form item type defines a tree view to show a structured tree of data records with an optional set of columns. The TREE basically acts like a TABLE container, in the sense of its function as a record list container. A TREE container renders as a tree view widget, with regular table columns on the right of the tree view.

A TREE is controlled by a **DISPLAY ARRAY** dialog, see List dialogs on page 1831 for more details about this type of dialog.

**Defining a TREE**
The TREE form item defines a tree view widget to show a structured set of data records. It is bound to a screen array grouping form fields which define the columns of the tree view.

The screen array definition must have exactly the same number of columns as the TREE form item. Use **PHANTOM fields**, if the number of record members in the program array exceeds the number of columns to be displayed in the TREE container.

TREE container definitions are very similar to regular TABLE containers; before reading further about tree views, you may need to familiarize yourself with TABLE containers.

The first column in the TREE must be the field defining the text of the tree view nodes.

Column titles can be defined with static labels in the TREE layout. However, this prevents you from localizing the column text. Consider using the **TITLE** attribute in the form field definition corresponding to the table columns.

Front-ends support different presentation and behavior options, which can be controlled by a **STYLE** attribute. For more details, see Style attributes common to all elements on page 1182 and Tree style attributes on page 1209.

**Where to use a TREE**
In a grid-based layout, a TREE form item can be defined in different ways:

1. As a **TREE container** in a LAYOUT tree.
2. As a **<TREE >** layout tag with a TREE item definition in the ATTRIBUTES section.

**Defining the TREE layout and size**
The number of field columns composing the TREE container in the form layout defines the initial width of the tree view.

The width of a TREE container is by default defined by the number of visible columns in its layout, and the initial height is defined by the number of rows. To specify explicitly the width and height of a tree view, use the **WIDTH** and **HEIGHT** attributes.

By default, a tree view is resizable in width and height. To force a TREE to keep the height defined in the form file, use the **WANTFIXEDPAGESIZE** attribute.

Tree view columns can be moved around, their width can be adapted, they can be hidden/shown, and can be selected to sort the record list automatically. To turn off these features, use respectively the **UNMOVABLECOLUMNS**, **UNSIZABLECOLUMNS**, **UNHIDABLECOLUMNS** and **UNSORTABLECOLUMNS** attributes.
For more details, see also Controlling table rendering on page 1858.

**VBOX item type**
Defines a layout area to render child elements in vertical direction.

**VBOX item basics**
The VBOX container automatically packs the contained elements vertically from top to bottom.

- Contained elements are packed in the order in which they appear in the LAYOUT section of the form file.
- No decoration (border) is added when you use a VBOX container.
- By combining VBOX and HBOX containers, you can define any alignment you choose.

**Defining an VBOX**
A VBOX is defined in a grid-based LAYOUT tree, as a container for other form items. It can for example be combined with an HBOX container to pack form elements to be displayed in vertical and horizontal directions:

```
LAYOUT
  HBOX
    VBOX
      GROUP
      ...
    END
    TABLE
      ...
    END
  END
END TABLE
END VBOX
END
```

Front-ends support different presentation and behavior options, which can be controlled by a STYLE attribute. For more details, see Style attributes common to all elements on page 1182.

**Where to use a VBOX**
A VBOX form item can only be defined as a VBOX container in a LAYOUT tree, within a grid-based layout.

**Note**: A VBOX cannot be used inside a STACK container.

**WEBCOMPONENT item type**
Defines a specialized form item that holds an external component.

**WEBCOMPONENT item basics**
The WEBCOMPONENT form item defines a form field that will hold an external component, implemented with a front-end plug-in mechanism.

This topic describes the WEBCOMPONENT item type in form definition files. For more details see the chapter dedicated to web component programming.

**Defining a WEBCOMPONENT**
The COMPONENTTYPE attribute identifies gICAPI external objects to be used for the field. The PROPERTIES attribute is typically used to define attributes that are specific to a given gICAPI-based web component. For example, a chart component might have properties to define x-axis and y-axis labels. For more details, see Using a gICAPI web component on page 1932.
If the COMPONENTTYPE attribute is not used, the web component will be a URL-based web component. For more details, see Using a URL-based web component on page 1928.

Front-ends support different presentation and behavior options, which can be controlled by a STYLE attribute. For more details, see Style attributes common to all elements on page 1182.

**Where to use a WEBCOMPONENT**

A WEBCOMPONENT form item can be defined in different ways:

1. With an item tag and a WEBCOMPONENT item definition on page 1349 in a grid-layout container (GRID, SCROLLGRID and TABLE).
2. As a WEBCOMPONENT stack item on page 1324 in a STACK container.

**Built-in Web Components**

Genero BDL provides a set of ready-to-use web components, that are deployed by default.

For more details, see Built-in web components on page 1972.

**Defining the widget size**

The size of a WEBCOMPONENT widget can be controlled in grid-based or stack-based layout, based on several attributes such as SIZEPOLICY and STRETCH.

For more details about image sizing, see Controlling the web component layout on page 1924.

**Related concepts**

SIZEPOLICY attribute on page 1391

The SIZEPOLICY attribute is a sizing directive based on the content of a form item.

**Form file structure**

A form specification file is defined by a set of sections.

The sections of a form specification file must appear in the following order:

1. SCHEMA section on page 1294
2. ACTION DEFAULTS section on page 1296
3. TOPMENU section on page 1297
4. TOOLBAR section on page 1298
5. TABLES section on page 1329
6. LAYOUT section on page 1301 (or SCREEN section on page 1300 for TUI mode)
7. ATTRIBUTES section on page 1330
8. INSTRUCTIONS section on page 1350

Each section must begin with the keyword for which it is named, only the LAYOUT section is mandatory.

**Related concepts**

Examples on page 1414

Form definition (.per) examples.

**SCHEMA section**

Defines the database schema file to be used to compile the form.

Each form specification file can begin with a SCHEMA section identifying the database schema (if any) on which the form is based. This can be any database schema that is defined with a database schema file. Form field data types can be automatically extracted from the schema file if you specify the table and column name in the form field definition (see ATTRIBUTES section).
Syntax 1

```
SCHEMA database@dbserver string FORMONLY
```

1. `database` is the name of the database schema to be used for the form compilation.
2. `dbserver` identifies the Informix® database server (INFORMIXSERVER).
3. `string` can be a string literal containing the database name.

Syntax 2: (supported for backward compatibility)

```
DATABASE database@dbserver string FORMONLY WITHOUT NULL INPUT
```

The DATABASE syntax is supported for compatibility with Informix® 4gl; using SCHEMA is recommended.

1. `database` is the name of the database schema to be used for the form compilation.
2. `dbserver` identifies the Informix® database server (INFORMIXSERVER)
3. `string` can be a string literal containing the database name.

**Usage**

The SCHEMA (or DATABASE) defines the database schema to be used to resolve data types for database column-based fields.

**Note:** The DATABASE instruction is supported for backward compatibility, we recommend using SCHEMA instead.

The SCHEMA section must appear in the sequence described in form file structure.

The SCHEMA section is optional; if you do not specify it, database schema specification defaults to SCHEMA FORMONLY.

You can create a form that is not related to any database schema by using the FORMONLY keyword after SCHEMA/DATABASE. When using this option, you must omit the TABLES section and define field data types explicitly in the ATTRIBUTES section.

The `database` and `dbserver` specifications are supported (but ignored) for backward compatibility with Informix® form specifications.

When using a specific database schema, the field data types are taken from the schema file during compilation. Make sure that the database schema file of the development database corresponds to the production database; otherwise the form fields defined in the compiled version of your forms will not match the table structures of the production database.

The use of the WITHOUT NULL INPUT option in the DATABASE syntax is supported for backward compatibility, but is ignored.

**Example**

```
SCHEMA stores
LAYOUT
...
```

**Related concepts**

- [Database schema on page 467](#)
  Defines database table structures with column type information to be reused in program variable definitions.
- [TABLES section on page 1329](#)
  Defines the list of database tables referenced by form field definitions.
- [ATTRIBUTES section on page 1330](#)
The **ATTRIBUTES** section describes properties of grid-based layout elements used in the form.

**ACTION DEFAULTS** section
The **ACTION DEFAULTS** section defines local action view default attributes for the form elements.

**Syntax**

```
ACTION DEFAULTS
  ACTION action-identifier ( action-attribute [, ...] )
  END
```

1. **action-identifier** defines the name of the action.
2. **action-attribute** defines an attribute for the action.

**Form attributes**

`ACCELERATOR, ACCELERATOR2, ACCELERATOR3, ACCELERATOR4, DEFAUTVIEW, COMMENT, CONTEXTMENU, IMAGE, TEXT, VALIDATE`.

**Style attributes**

Not applicable.

**Usage**

The **ACTION DEFAULTS** section centralizes action view attributes (text, comment, image, accelerators) at the form level.

The **ACTION DEFAULTS** section must appear in the sequence described in **form file structure**.

The **ACTION DEFAULTS** section is optional.

The section holds a list of **ACTION** elements that specify attributes for each action. The action is identified by the name following the **ACTION** keyword, and attributes are specified in a list between parenthesis.

The attributes defined in this section are applied to form action views like buttons, toolbar buttons, or topmenu options, if the individual action views do not explicitly define their own attributes.

Action attributes can be defined at different levels, see **action configuration** for more details.

**Example**

```
ACTION DEFAULTS
  ACTION accept ( COMMENT="Commit order record changes", CONTEXTMENU=NO )
  ACTION cancel ( TEXT="Stop", IMAGE="stop", ACCELERATOR=SHIFT-F2, VALIDATE=NO )
  ACTION print ( COMMENT="Print order information", ACCELERATOR=CONTROL-P, ACCELERATOR2=F5 )
  ACTION zoom1 ( COMMENT="Open items list", VALIDATE=NO )
  ACTION zoom2 ( COMMENT="Open customers list", VALIDATE=NO )
END
```

**Related concepts**

- **Action handling basics** on page 1739
  This topic describes the basics of action views, action events, and action handlers.

  **ON ACTION block** on page 1478
**TOPMENU section**

The TOPMENU section defines a pull-down menu with options that are bound to actions.

**Syntax**

```
TOPMENU [menu-identifier] ( menu-attribute [, ...] )
group
  [ ... ]
END
```

where `group` is:

```
GROUP [group-identifier] ( group-attribute [, ...] )
  [ command ]
  [ group ]
  [ separator ]
  [ ... ]
END
```

where `command` is:

```
COMMAND [command-identifier] ( command-attribute [, ...] )
```

and `separator` is:

```
SEPARATOR [separator-identifier] ( separator-attribute [, ...] )
```

1. `menu-identifier` defines the name of the top menu (optional).
2. `group-identifier` defines the name of the group.
3. `command-identifier` defines the name of the action to bind to.
4. `separator-identifier` defines the name of the separator (optional).
5. `menu-attribute` can be: STYLE, TAG.
6. `group-attribute` is one of: STYLE, TEXT, IMAGE, COMMENT, TAG, HIDDEN.
7. `command-attribute` is one of: STYLE, TEXT, IMAGE, COMMENT, TAG, HIDDEN, ACCELERATOR.
8. `separator-attribute` is one of: STYLE, TAG, HIDDEN.

**Form attributes**

ACCELERATOR, COMMENT, HIDDEN, IMAGE, STYLE, TEXT, TAG.

**Style attributes**

Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.

Class-specific: none.

**Usage**

The TOPMENU section is used to define a pull-down menu in a form.

The TOPMENU section must appear in the sequence described in form file structure.

The TOPMENU section is optional.

In a TOPMENU section, you build a tree of GROUP elements to design the pull-down menu. A GROUP can contain COMMAND, SEPARATOR or GROUP children. A COMMAND defines a pull-down menu option that triggers an action when it is selected. In the topmenu specification, command-identifier defines which action a menu option is bound to.
For example, if you define a topmenu option as "COMMAND zoom", it can be controlled by an "ON ACTION zoom" clause in an interactive instruction.

The topmenu commands are enabled depending on the actions defined by the current interactive instruction. For example, you can define a topmenu option with the action name "cancel" to bind the pull-down item to this predefined dialog action.

An accelerator name can be defined for a topmenu command; this accelerator name will be used for display in the command item. You must define the same accelerator in the action defaults section for the action name of the topmenu command.

TOPMENU elements can get a STYLE attribute in order to use a specific rendering/decoration based on presentation style definitions.

Example

```
TOPMENU tm ( STYLE="mystyle" )
  GROUP form (TEXT="Form")
    COMMAND help (TEXT="Help", IMAGE="quest")
    COMMAND quit (TEXT="Quit")
  END
  GROUP edit (TEXT="Edit")
    COMMAND accept (TEXT="Validate", IMAGE="ok", TAG="acceptMenu")
    COMMAND cancel (TEXT="Cancel", IMAGE="cancel")
  SEPARATOR
    COMMAND editcut   -- Gets its decoration from action defaults
    COMMAND editcopy  -- Gets its decoration from action defaults
    COMMAND editpaste -- Gets its decoration from action defaults
  END
  GROUP records (TEXT="Records")
    COMMAND append (TEXT="Add", IMAGE="plus")
    COMMAND delete (TEXT="Remove", IMAGE="minus")
    COMMAND update (TEXT="Modify", IMAGE="accept")
  SEPARATOR (TAG="lastSeparator")
    COMMAND search (TEXT="Search", IMAGE="find")
  END
END
```

Related concepts

Binding action views to action handlers on page 1762
How are action views of the forms bound to action handlers in the program code?

Topmenus on page 1444
Topmenus define typical pull-down menus that appear at the top of application forms.

ACTION DEFAULTS section on page 1296
The ACTION DEFAULTS section defines local action view default attributes for the form elements.

TOOLBAR section
The TOOLBAR section defines a toolbar with buttons that are bound to actions.

Syntax

```
TOOLBAR [ toolbar-identifier ] ( toolbar-attribute )
  { ITEM item-identifier ( item-attribute )
    | SEPARATOR separator-identifier ( separator-attribute )
  }...
END
```

1. toolbar-identifier defines the name of the toolbar (optional).
2. item-identifier defines the name of the action to bind to.
3. separator-identifier defines the name of the separator (optional).
4. toolbar-attribute is one of: STYLE, TAG, BUTTONTEXTHIDDEN.
5. item-attribute is one of: STYLE, TAG, TEXT, IMAGE, COMMENT, HIDDEN.
6. separator-attribute is one of: STYLE, TAG, HIDDEN.

Form attributes
BUTTONTEXTHIDDEN, COMMENT, HIDDEN, IMAGE, STYLE, TEXT, TAG.

Style attributes
Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.
Class-specific: scaleIcon, toolBarTextPosition.

Usage
The TOOLBAR section defines a toolbar in a form.

The TOOLBAR section must appear in the sequence described in form file structure.

The TOOLBAR section is optional.

A TOOLBAR section defines a set of ITEM elements that can be grouped by using a SEPARATOR element. Each ITEM defines a toolbar button associated with an action by name. The SEPARATOR keyword specifies a vertical line.

The toolbar buttons are enabled depending on the actions defined by the current interactive instruction. For example, you can define a toolbar button with the action name "cancel" to bind the toolbar item to this predefined dialog action.

Toolbar button labels are visible by default. The TOOLBAR supports the BUTTONTEXTHIDDEN attribute to hide the labels of buttons.

TOOLBAR elements can get a STYLE attribute in order to use a specific rendering/decoration following presentation style definitions.

Example

```
TOOLBAR tb ( STYLE="mystyle" )
   ITEM accept ( TEXT="Ok", IMAGE="ok" )
   ITEM cancel ( TEXT="Cancel", IMAGE="cancel" )
SEPARATOR
   ITEM editcut   -- Gets its decoration from action defaults
   ITEM editcopy  -- Gets its decoration from action defaults
   ITEM editpaste -- Gets its decoration from action defaults
SEPARATOR ( TAG="lastSeparator")
   ITEM append ( TEXT="Append", IMAGE="add" )
   ITEM update ( TEXT="Update", IMAGE="modify" )
   ITEM delete ( TEXT="Delete", IMAGE="del" )
   ITEM search ( TEXT="Search", IMAGE="find" )
END
```

Related concepts
Binding action views to action handlers on page 1762
How are action views of the forms bound to action handlers in the program code?

Toolbars on page 1436
Toolbars define a bar of buttons that appears at the top of application forms.

ACTION DEFAULTS section on page 1296
The **ACTION DEFAULTS** section defines local action view default attributes for the form elements.

**SCREEN section**
The **SCREEN** section defines the form layout for TUI mode forms.

**Syntax**

```
SCREEN
  [[ SIZE lines [ BY chars ] ]
  [ TITLE "title" ]
  [ TAG "tag-string" ]
  {
    [ text ] [ item-tag [ item-tag ] [. . . ] ]
  }
[ END ]
```

1. *lines* is the number of characters the form can display vertically. The default is 24.
2. *chars* is the number of characters the form can display horizontally. The default is the maximum number of characters in any line of the screen definition.
3. *title* is the title for the top window.
4. *tag-string* is a user-defined string.
5. *item-tag* and *text* define form elements in the layout.

**Usage**
The **SCREEN** section must be used to design TUI mode screens. For a GUI mode application, use a **LAYOUT** or **STACKED LAYOUT** section instead.

The **SCREEN** section must appear in the sequence described in **form file structure**.

The **SCREEN** section is mandatory, unless you use a **LAYOUT** section.

The **END** keyword is optional.

The **SIZE lines [ BY chars ]** clause is supported for backward compatibility. A good practice is to omit this clause, and let the form compiler compute the size of the form, based on the content of the curly brackets.

The **TAG** attribute can be used to specify a string that will help to identify the form at runtime. For more details about this attribute, see **TAG**.

Inside the **SCREEN** section, you can define the position of text labels and form fields in the area delimited by the {} curly brackets.

Between the curly brackets, horizontal lines can be specified with a sequence of dash characters (-----).

**Note:** Avoid Tab characters (ASCII 9) inside the area delimited by the curly brackets. If used, Tab characters will be replaced by 8 blank spaces by fglform.

You can include **graphics characters defined for your terminal** between the curly brackets to place rectangles in a screen form. Use \g to start and end graphics mode. For example, \g \g defines a vertical segment. The following characters can be used to indicate the borders of rectangles:

- \p upper-left corner.
- \q upper-right corner.
- \b left-lower corner.
- \d left-right corner.
- - (hyphen) to mark horizontal segments.
- | (pipe) to mark vertical segments.
Example

```
SCREEN TITLE "Customer info" TAG "regular"
{
  CustId : [f001] Name: [f002]
  Address: [f003]
  [f003]
}------------------------------------------------
END
```

Related concepts

**LAYOUT section** on page 1301
The LAYOUT section defines the graphical alignment of the form by using a tree of layout containers.

**Item tags** on page 1259
*Item tags* define the position and size in a grid-based container.

**LAYOUT section**
The LAYOUT section defines the graphical alignment of the form by using a tree of layout containers.

**Syntax**

```
LAYOUT [ ( layout-attribute [, ...] ) ]
  root-container
    child-container
      [ , ... ]
  END
[END]
```

1. *layout-attribute* is an attribute for the whole form.
2. *root-container* is the first container that holds *child-containers*.

**Form attributes**

*IMAGE, MINHEIGHT, MINWIDTH, SPACING, STYLE, TEXT, TAG, VERSION, WINDOWSTYLE.*

**Style attributes (form)**

Common: *backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.*

Class-specific: *resetFormSize.*

See also: *Window style attributes* on page 1220, *UserInterface style attributes* on page 1219.

**Can hold**

*FORM, VBOX, HBOX, GROUP, FOLDER, GRID, SCROLLGRID, STACK, TABLE, TREE.*

**Usage**

The LAYOUT section is used to define a tree of layout containers, it can mix grid-based layout containers (GRID), with stack-based layout containers (STACK).

The LAYOUT section must appear in the sequence described in *form file structure*.

This section is mandatory, unless you use a SCREEN section.

Indentation is supported in the LAYOUT section.

The END keyword is optional.
The layout tree of the form is defined by associating layout containers. Different kinds of layout containers are provided, each of them having a specific role. Some containers such as VBOX, HBOX and FOLDER can hold children containers, while others such as GRID and TABLE define a screen area. Containers using a screen area define a formatted region containing static text labels, item tags and layout tags. External form files can be included in the current layout with the FORM clause.

```
LAYOUT (VERSION="12", STYLE="regular")
  VBOX
    GRID grid1
    grid-area
    END
    GROUP group1
    HBOX
      GRID grid2
      grid-area
      END
      TABLE table1
      table-area
      END
    END
  END
END
```

The definition would result in a layout tree that looks like this:

```
-- VBOX
  | +-- GRID grid1
  |     | +-- GROUP group1
  |     |     | +-- HBOX
  |     |     |     | +-- GRID grid2
  |     |     |     |     | +-- TABLE table1
```

The layout section can also contain a simple GRID container (equivalent to a V3 SCREEN definition):

```
LAYOUT
  GRID
  grid-area
  END
END
```

**Description of LAYOUT attributes**

The VERSION attribute can be used to specify a version for the form. This allows you to indicate that the form content has changed. Typically used to avoid having the front-end reload the saved window settings.

The MINHEIGHT, MINWIDTH attributes can be used to specify a minimum width and height for the form. You typically use these attributes to force the form to have a bigger size than the default when it is first rendered. If the front-end stores window sizes, these attributes will only be significant the first time the form is opened, or each time the VERSION attribute is changed.

The IMAGE attribute can be used to define the icon of the window that will display the form. This attribute is automatically applied to the parent window node when a form is loaded.

The TEXT attribute can be used to define the title of the window that will display the form. This attribute is automatically applied to the parent window node when a form is loaded.
The **SPACING** attribute can be used to give a hint to the front-end to define the gap between form elements.

The **STYLE** attribute defines the presentation style for form elements, you can for example define a font property for all form elements.

With the **WINDOWSTYLE** attribute, you can define the window type and decoration. This attribute is automatically applied to the parent window when a form is loaded. For backward compatibility, the **STYLE** attribute is used as the default **WINDOWSTYLE** if this attribute is not used.

**Related concepts**

- **Stack-based layout** on page 1431
  A form file can define a stack-based layout within a tree of stack items.

- **SCREEN section** on page 1300
  The **SCREEN** section defines the form layout for TUI mode forms.

- **ATTRIBUTES section** on page 1330
  The **ATTRIBUTES** section describes properties of grid-based layout elements used in the form.

**FORM clause**

Reuse the definition of a form in the current form.

**Syntax**

```plaintext
FORM "form-file"
```

1. **form-file** is the form to be included (without .per extension).

**Usage**

The **FORM** clause includes an external form at the current layout position, enforcing form re-usability, or to solve form complexity when using a **DIALOG** instruction; for example to define a common form header for several application forms.

Wherever a layout container can be specified, the layout of an external form can be merged into the layout of the current form, with the **FORM** clause. See **External form inclusion** on page 1265.

The .per source of the included form must be readable. If the compiled version (.42f) does not exist, or is older than the .per source, fglform will automatically compile the included form. The included forms can be located in a different directory as the main form.

The form compiler searches for the external form relative to the path of the current compiled form. For example, with fglform dir1/dir2/main.per, when the main form includes an external form with FORM "./otherdir/subform", fglform will include the form file located in dir1/otherdir/subform.per.

The form compiler performs an up-to-date test of the compiled form. Error -6842 is thrown if the up-to-date test fails.

If the external form contains a **TOOLBAR** or a **TOPMENU** section, error -6841 is thrown.

The external form must not define a **SCREEN RECORD** or use a **TABLE** already defined in the current form, otherwise error -2024 is thrown. Consider using the **table alias syntax** to avoid duplicate table names in merged forms.

The external form can define its own **ACTION DEFAULTS** section. The action defaults of the external file will be merged into the action defaults of the current form.

The **TABINDEX** attributes of the elements of the result form will be adjusted. As the result tabbing (**OPTIONS FIELD ORDER FORM** in programs) keeps the visual order of the layout.

**Example**

```plaintext
LAYOUT
  FOLDER
  PAGE page1 (TEXT = "Customer")
    FORM "customer"
```
Related concepts

Screen records / arrays on page 1251
Form fields can be grouped in a screen record or screen array definition.

TOOLBAR section on page 1298
The TOOLBAR section defines a toolbar with buttons that are bound to actions.

TOPMENU section on page 1297
The TOPMENU section defines a pull-down menu with options that are bound to actions.

The SUBDIALOG clause on page 1600

HBOX container
Packs child layout elements horizontally.

Syntax

HBOX [[identifier] [ ( attribute [ , ... ] ) ] ]
   layout-container [...]
END

1. identifier defines the name of the element.
2. attribute is an attribute for the element.
3. layout-container is another child container.

Form attributes

COMMENT, FONTPITCH, HIDDEN, STYLE, SPLITTER, TAG.

Style attributes

Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight,
localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.
Class-specific: splitViewRendering.

Can hold

VBOX, HBOX, GROUP, FOLDER, GRID, SCROLLGRID, TABLE, TREE.

Usage

An HBOX container packs other form items together, in the horizontal direction.
For more details about this item type, see HBOX item type on page 1278.

Example

HBOX
   GROUP ( TEXT = "Customer" )
   ...
END
TABLE ...

**Related concepts**

**Grid-based layout** on page 1418
A form file can define a grid-based layout within a tree of layout items.

**Using hbox tags to align form items** on page 1428
The hbox tag concept has been introduced to bypass the limitations of the character-based grid in forms.

**Automatic HBoxes and VBoxes** on page 1422
Horizontal and vertical boxes are added automatically when stretchable elements are used.

**VBOX container**
Packs child layout elements vertically.

**Syntax**

```
VBOX [ identifier ] [ ( attribute [, ...] ) ]
    layout-container
    [, ...]
END
```

1. *identifier* defines the name of the element.
2. *attribute* is an attribute for the element.
3. *layout-container* is another child container.

**Form attributes**

- `COMMENT`, `FONTPITCH`, `HIDDEN`, `STYLE`, `SPLITTER`, `TAG`.

**Style attributes**

Common: `backgroundColor`, `border`, `fontFamily`, `fontSize`, `fontStyle`, `fontWeight`, `localAccelerators`, `showAcceleratorInToolTip`, `textColor`, `textDecoration`.

Class-specific: none.

**Can hold**

- `VBOX`, `HBOX`, `GROUP`, `FOLDER`, `GRID`, `SCROLLGRID`, `TABLE`, `TREE`.

**Usage**

A VBOX container packs other form items together, in the vertical direction.

For more details about this item type, see **VBOX item type** on page 1293.

**Example**

```
VBOX
    GROUP ( TEXT = "Customer" )
    ...
END
TABLE
    ...
END
END
```

**Related concepts**

**Grid-based layout** on page 1418
A form file can define a grid-based layout within a tree of layout items.

**Item tags** on page 1259

Item tags define the position and size in a grid-based container.

**Layout tags** on page 1254

Layout tags define layout areas for containers inside the frame of a grid-based container.

**Automatic HBoxes and VBoxes** on page 1422

Horizontal and vertical boxes are added automatically when stretchable elements are used.

**GROUP container**

Defines a layout area to group other layout elements together, in a grid-based layout.

**Syntax**

```
GROUP [identifier] ([ attribute [, ..] ] )
[layout-container]
[..]
END
```

1. `identifier` defines the name of the element.
2. `attribute` is an attribute for the element.
3. `layout-container` is another child container.

**Form attributes**

`COMMENT`, `FONTPITCH`, `STYLE`, `TAG`, `HIDDEN`, `TEXT`.

**Style attributes**

Common: `backgroundColor`, `border`, `fontFamily`, `fontSize`, `fontStyle`, `fontWeight`, `localAccelerators`, `showAcceleratorInToolTip`, `textColor`, `textDecoration`.

Class-specific: `collapsible`, `collapserPosition`, `initiallyCollapsed`.

**Can hold**

`VBOX`, `HBOX`, `GROUP`, `FOLDER`, `GRID`, `SCROLLGRID`, `TABLE`, `TREE`.

**Usage**

In a `LAYOUT` tree definition, use a `GROUP` container to hold other containers such as a `VBOX` with children, or a `GRID` container.

**Note:** When defining a `GROUP` container, you cannot set the `GRIDCHILDRENINPARENT` attribute. This attribute makes sense only for a group item defined with a layout tag contained in a `GRID` area.

For more details about this item type, see **GROUP item type** on page 1276.

**Example**

```
GROUP ( TEXT = "Customer" )
VBOX
  GRID
  {
    ...
  }
END
TABLE
  {
    ...
  }
```
Related concepts

Stacked group rendering on page 1434
Groups render in a native way on front-ends supporting the stacked layout.

Layout tags on page 1254
Layout tags define layout areas for containers inside the frame of a grid-based container.

FOLDER container
Defines the parent container for folder pages, in a grid-based layout.

Syntax

FOLDER \( \text{identifier} \) \( \{ \text{attribute} \{ \}, ..., \} \) \( \text{folder-page} \)
\( \{ \} \)
END

1. \textit{identifier} defines the name of the element.
2. \textit{attribute} is an attribute for the element.
3. \textit{folder-page} defines a folder page that contains other form elements.

Form attributes

COMMENT, FONTPITCH, STYLE, TAG, HIDDEN.

Style attributes

Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.
Class-specific: collapserPosition, position.

Can hold

PAGE

Usage

A FOLDER container including PAGE elements defines a folder tab widget.
Define each folder page with a PAGE container inside the FOLDER container.
For more details about this item type, see FOLDER item type on page 1275.

PAGE container
Defines the content of a folder page, in a grid-based layout.

Syntax

PAGE \( \text{identifier} \) \( \{ \text{attribute} \{ \}, ..., \} \) \( \text{layout-container} \)
\( \{ \} \)
END

1. \textit{identifier} defines the name of the element.
2. *attribute* is an attribute for the element.
3. *layout-container* is another child container.

**Form attributes**

*ACTION, COMMENT, HIDDEN, IMAGE, STYLE, TAG, TEXT.*

**Style attributes**

Common: *backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.*

Class-specific: none.

**Can hold**

*VBOX, HBOX, GROUP, FOLDER, GRID, SCROLLGRID, TABLE, TREE.*

**Usage**

In a *LAYOUT* tree definition, use a *PAGE* container to define a folder page that holds other containers such as a *VBOX* with children, or a *GRID* container.

A *PAGE* container always belongs to a parent *FOLDER* container.

For more details about this item type, see *PAGE item type* on page 1282.

**Example**

```plaintext
FOLDER
  PAGE p1 ( TEXT="Global info" )
  GRID
    { ...
    }
  END
END
PAGE p2 ( IMAGE="list" )
  TABLE
    { ...
    }
  END
END
END

*GRID container*

Defines a layout area based on a grid of cells.

**Syntax**

```plaintext
GRID [identifier] [ ( attribute [,..] ) ]
{
  [ text
    [ item-tag
    [ hbox-tag
    [ layout-tag
    [ horizontal-line ] ]
  [ ] ] ]
}
END
```
1. **text** is literal text that will appear in the form as a static label.
2. **item-tag** defines the position and length of a form item.
3. **hbox-tag** defines the position and length of several form items inside an horizontal box.
4. **layout-tag** defines the position and length of a layout tag.
5. **horizontal-line** is a set of dash characters defining a horizontal line.

**Form attributes**

COMMENT, FONTPITCH, HIDDEN, STYLE, TAG.

**Style attributes**

Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.

Class-specific: none.

**Usage**

The GRID container declares a formatted text block, defining the dimensions and the positions of children form items.

**Note:** Avoid Tab characters (ASCII 9) inside the curly-brace delimited area. If used, Tab characters will be replaced by 8 blanks by fglform.

For more details about this item type, see GRID item type on page 1276.

**Example**

```
GRID
{
 <GROUP g1                                   >
  Id:  [f1] Name: [f2                      
  Addr: [f3                                 
 <                                           >
  }                                           
END
```

**Related concepts**

Grid-based layout on page 1418

A form file can define a grid-based layout within a tree of layout items.

**STACK container**

The STACK container holds stack items defining a logical alignment of form items.

**Important:** This feature is experimental, the syntax/name and semantics/behavior may change in a future version.

**Syntax**

```
STACK
{ scalable-item
  | container-list
  ...
END
```

where *container-list* is:

```
grouping-item
  leaf-item
  [...]
END
```
1. **scalable-item** is a leaf element of the stacked layout, for widgets with a scalable width and height.
2. **grouping-item** is a stacked layout grouping element that holds a list of **leaf-items**.
3. **leaf-item** is a leaf element of the stacked layout, for widgets with a fixed size (non-scalable).

**Can hold**

Scalable stack items: **IMAGE, TEXTEDIT, WEBCOMPONENT**.

or:

Grouping stack items: **FOLDER, GROUP, TABLE**.

**Usage**

The **STACK** container is used to define a **stack-based layout**.

**Note:** Unlike grid-based containers (**GRID**) where element definition is split in the LAYOUT and ATTRIBUTES sections, the items in a **STACK** container define both the position and attributes.

The **STACK** container must appear immediately after the LAYOUT keyword.

The **STACK** container typically defines a list of elements (such as **GROUP, FOLDER, TABLE**), grouping leaf stack items (such as form fields) together:

```
LAYOUT
STACK
  GROUP custinfo (TEXT="Customer info")
    EDIT customer.cust_num, TITLE="Num:", NOENTRY;
    EDIT customer.cust_name, TITLE="Name:", SCROLL;
    ...
  END
  TABLE cust_orders (STYLE="compact_list", DOUBLECLICK=select)
    LABEL orders.ord_num, TITLE="Num";
    LABEL orders.ord_ship, TITLE="Ship date";
    LABEL orders.ord_value, TITLE="Value";
    ...
  END
  ...
END
```

A stack container can also define a single scalable stack item, such as an **IMAGE, TEXTEDIT, or WEBCOMPONENT**:

```
STACK
  IMAGE FORMONLY.picture;
END
```

**Related concepts**

- Stack-based layout on page 1431
- A form file can define a stack-based layout within a tree of stack items.

**BUTTON stack item**

Defines a push-button that can trigger an action in a stack-based layout.

**Syntax**

```
BUTTON item-name \, attribute-list \;
```

1. **item-name** defines the form item name and the action name.
2. *attribute-list* defines the aspect and behavior of the form item.

**Form attributes**

*COMMENT, DISCLOSUREINDICATOR, FONTPITCH, HIDDEN, IMAGE, SAMPLE, SIZEPOLICY, STYLE, TABINDEX, TAG, TEXT.*

**Style attributes**

Common: *backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.*

Class-specific: *alignment, buttonType, scaleIcon.*

**Usage**

Define the rendering and behavior of a button stack item, with a *BUTTON* element inside a *STACK* container.

For more details about this item type, see *BUTTON item type* on page 1266.

**Example**

`BUTTON print, TEXT="Print Report", IMAGE="printer";`

**BUTTONEDIT stack item**

Defines a line-edit with a push-button that can trigger an action in a stack-based layout.

**Syntax**

`BUTTONEDIT field-name [, attribute-list ];`

1. *field-name* identifies the name of the screen record field.
2. *attribute-list* defines the aspect and behavior of the form item.

**Form attributes**

*ACTION, AUTONEXT, CENTURY, COLOR, COMPLETER, COLOR WHERE, COMMENT, DEFAULT, DISPLAY LIKE, DOWNSHIFT, FONTPITCH, HIDDEN, FORMAT, IMAGE, INCLUDE, INVISIBLE, JUSTIFY, KEY, KEYBOARDHINT, NOT NULL, NOTEDITABLE, NOENTRY, PICTURE, PLACEHOLDER, PROGRAM, REVERSE, SAMPLE, SCROLL, STYLE, REQUIRED, TAG, TITLE, TABINDEX, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, UPSHIFT, VALIDATE LIKE, VERIFY.*

**Style attributes**

Common: *backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.*

Class-specific: *scaleIcon.*

**Usage**

Define the rendering and behavior of a buttonedit stack item, with a *BUTTONEDIT* element inside a *STACK* container.

For more details about this item type, see *BUTTONEDIT item type* on page 1267.

**Example**

`BUTTONEDIT customer.state, REQUIRED, IMAGE="smiley", ACTION=zoom;`
CHECKBOX stack item
Defines a boolean or three-state checkbox field in a stack-based layout.

Syntax

```
CHECKBOX  field-name [, attribute-list ] ;
```

1. `field-name` identifies the name of the screen record field.
2. `attribute-list` defines the aspect and behavior of the form item.

Form attributes

```
COLOR, COLOR WHERE, COMMENT, DEFAULT, FONTPITCH, HIDDEN, INCLUDE, JUSTIFY, KEY, NOT NULL, NOENTRY, REQUIRED, SAMPLE, SIZEPOLICY, STYLE, TAG, TABINDEX, TEXT, TITLE, VALIDATE LIKE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, VALUECHECKED, VALUEUNCHECKED.
```

Style attributes

```
Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.
```

Usage

Define the rendering and behavior of a checkbox stack item, with a CHECKBOX element inside a STACK container.

For more details about this item type, see CHECKBOX item type on page 1269.

Example

```
CHECKBOX customer.active,
   REQUIRED, TEXT="Active",
   VALUECHECKED="Y", VALUEUNCHECKED="N";
```

COMBOBOX stack item
Defines a line-edit with a drop-down list of values in a stack-based layout.

Syntax

```
COMBOBOX  field-name [, attribute-list ] ;
```

1. `field-name` identifies the name of the screen record field.
2. `attribute-list` defines the aspect and behavior of the form item.

Form attributes

```
COLOR, COLOR WHERE, COMMENT, DEFAULT, DOWNSHIFT, FONTPITCH, HIDDEN, KEY, INCLUDE, INITIALIZER, ITEMS, JUSTIFY, NOT NULL, NOENTRY, PLACEHOLDER, QUERYEDITABLE, REQUIRED, SAMPLE, SCROLL, SIZEPOLICY, STYLE, UPSHIFT, TAG, TABINDEX, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, TITLE, VALIDATE LIKE.
```

Style attributes

```
Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.
```

Usage

Define the rendering and behavior of a combobox stack item, with a COMBOBOX element inside a STACK container.

For more details about this item type, see COMBOBOX item type on page 1270.

Example

```
COMBOBOX customer.active,
   REQUIRED, TEXT="Active",
   VALUECHECKED="Y", VALUEUNCHECKED="N";
```
**Usage**

Define the rendering and behavior of a combobox stack item, with a COMBOBOX element inside a STACK container.

For more details about this item type, see COMBOBOX item type on page 1270.

**Example**

```plaintext
COMBOBOX customer.city,
  ITEMS=((1,"Paris"),
         (2,"Madrid"),
         (3,"London"));
COMBOBOX customer.sector,
  REQUIRED,
  ITEMS=("SA","SB","SC");
COMBOBOX customer.state,
  NOT NULL,
  INITIALIZER=myinit;
```

DATEEDIT stack item

Defines a line-edit with a calendar widget to pick a date in a stack-based layout.

**Syntax**

```plaintext
DATEEDIT field-name \[, attribute-list \] ;
```

1. `field-name` identifies the name of the screen record field.
2. `attribute-list` defines the aspect and behavior of the form item.

**Form attributes**

AUTONEXT, CENTURY, COLOR, COLOR WHERE, COMMENT, DEFAULT, FONTPITCH, FORMAT, HIDDEN, IMAGECOLUMN, INCLUDE, JUSTIFY, KEY, NOT NULL, NOENTRY, PLACEHOLDER, REQUIRED, SAMPLE, STYLE, TAG, TABINDEX, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, VALIDATE LIKE.

**Style attributes**

Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.

Class-specific: buttonIcon, calendarType, daysOff, firstDayOfWeek, showCurrentMonthOnly, showGrid, showWeekNumber.

**Usage**

Define the rendering and behavior of a date edit stack item, with a DATEEDIT element inside a STACK container.

For more details about this item type, see DATEEDIT item type on page 1272.

**Example**

```plaintext
DATEEDIT order.shipdate;
```

DATETIMEEDIT stack item

Defines a line-edit with a calendar widget for selecting a datetime in a stack-based layout.

**Syntax**

```plaintext
DATETIMEEDIT field-name \[, attribute-list \] ;
```

1. `field-name` identifies the name of the screen record field.
2. *attribute-list* defines the aspect and behavior of the form item.

**Form attributes**

AUTONEXT, COLOR, COLOR WHERE, COMMENT, DEFAULT, FONTPITCH, HIDDEN, INCLUDE, IMAGECOLUMN, JUSTIFY, NOT NULL, NOENTRY, PLACEMARKER, REQUIRED, SAMPLE, STYLE, TABINDEX, TAG, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, VALIDATE LIKE.

**Style attributes**

Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.

Class-specific: buttonIcon, calendarType, daysOff, firstDayOfWeek, showCurrentMonthOnly, showGrid, showWeekNumber.

**Usage**

Define the rendering and behavior of a date-time edit stack item, with a DATETIMEEDIT element inside a STACK container.

For more details about this item type, see DATETIMEEDIT item type on page 1273.

**Example**

```
DATETIMEEDIT package.modts;
```

EDIT stack item

Defines an element to enter a single-line text in a stack-based layout.

**Syntax**

```
EDIT [identifier] [, attribute-list ]
```

1. *field-name* identifies the name of the screen record field.
2. *attribute-list* defines the aspect and behavior of the form item.

**Form attributes**

AUTONEXT, CENTURY, COLOR, COLOR WHERE, COMMENT, COMPLETER, DEFAULT, DISPLAY LIKE, DOWNSHIFT, FONTPITCH, FORMAT, HIDDEN, IMAGECOLUMN, INCLUDE, INVISIBLE, JUSTIFY, KEY, KEYBOARDHINT, NOT NULL, NOENTRY, PICTURE, PLACEMARKER, PROGRAM, REQUIRED, REVERSE, SAMPLE, STYLE, SCROLL, TAG, TABINDEX, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, UPSHIFT, VALIDATE LIKE, VERIFY.

**Style attributes**

Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.

Class-specific: dataTypeHint, spellCheck.

**Usage**

Define the rendering and behavior of an edit stack item, with an EDIT element inside a STACK container.

For more details about this item type, see EDIT item type on page 1274.
Example

```
EDIT customer.cust_name, NOT NULL;
```

FOLDER stack item

Defines a stack area to hold a set of folder pages, in a stack-based layout.

Syntax

```
FOLDER [identifier] [ ( attribute-list ) ]
  folder-page
  [...]
END
```

1. `identifier` defines the name of the element.
2. `attribute-list` defines the aspect and behavior of the form item.
3. `folder-page` is a page element in the folder definition.

Form attributes

- `COMMENT`, `FONTPITCH`, `STYLE`, `TAG`, `HIDDEN`.

Style attributes

Common: `backgroundColor`, `border`, `fontFamily`, `fontSize`, `fontStyle`, `fontWeight`, `localAccelerators`, `showAcceleratorInToolTip`, `textColor`, `textDecoration`.

Class-specific: `collapserPosition`, `position`.

Can hold

- `PAGE`.

Usage

Use a `FOLDER` stack layout element to define a set of folder pages with a folder tab widget.

Define each folder page with a `PAGE` stack item inside the `FOLDER` container.

For more details about this item type, see `FOLDER item type` on page 1275.

For a code example, see `PAGE stack item` on page 1318.

GROUP stack item

Defines a stack area to group other layout elements together, in a stack-based layout.

Syntax

```
GROUP [identifier] [ ( attribute-list ) ]
  stack-item
  [...]
END
```

1. `identifier` defines the name of the element.
2. `attribute-list` defines the aspect and behavior of the form item.
3. `stack-item` is child element in the stack container.

Form attributes

- `COMMENT`, `FONTPITCH`, `HIDDEN`, `STYLE`, `TAG`, `TEXT`. 

Style attributes

Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.

Class-specific: collapsible, collapserPosition, initiallyCollapsed.

Can hold

BUTTON, BUTTONEDIT, CHECKBOX, COMBOBOX, DATEEDIT, DATETIMEEDIT, EDIT, IMAGE, LABEL, PROGRESSBAR, PHANTOM, SLIDER, SPINEDIT, TEXTEDIT, TABLE, TIMEEDIT, RADIOGROUP, WEBCOMPONENT.

Usage

Use a GROUP stack layout element to group other stack items together.

For more details about this item type, see GROUP item type on page 1276.

Example

```
GROUP group1 (TEXT="Customer info")
  EDIT ...
  BUTTONEDIT ...
  ...
END
```

Related concepts

Stacked group rendering on page 1434
Groups render in a native way on front-ends supporting the stacked layout.

Layout tags on page 1254

Layout tags define layout areas for containers inside the frame of a grid-based container.

IMAGE stack item

Defines an element to display an image resource in a stack-based layout.

Syntax 1: Defining a form field image

```
IMAGE field-name \[, attribute-list \]
```

Syntax 2: Defining a static image

```
IMAGE : item-name, IMAGE="resource" \[, attribute-list \]
```

1. `field-name` identifies the name of the screen record field.
2. `item-name` identifies the form element (name attribute in .42f)
3. `resource` is the image resource, see Providing the image resource on page 1149.
4. `attribute-list` defines the aspect and behavior of the form item.

Form attributes

ACTION, AUTOSCALE, COMMENT, HEIGHT, HIDDEN, SIZEPOLICY, STYLE, STRETCH, TAG, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, WIDTH.

Image field only: JUSTIFY, SAMPLE.

Static image only: IMAGE.
Style attributes

Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.

Class-specific: alignment, imageContainerType.

Usage

Define the rendering and behavior of an image stack item, with a IMAGE element inside a STACK container.

Note: The IMAGE stack item can be used inside a stack container like a group, or as root element of the STACK container. When used directly under the STACK container, the IMAGE stack item must be the only element in the container. It will be rendered a scalable form item that can stretch to fit the front-end screen size.

For more details about this item type, see IMAGE item type on page 1279.

Example

```
IMAGE cars.picture, COMMENT="Picture of the car";
```

LABEL stack item

Defines a simple text area to display a read-only value in a stack-based layout.

Syntax 1: Defining a form field label

```
LABEL field-name \, attribute-list \;
```

Syntax 2: Defining a static label

```
LABEL : item-name \, TEXT = \^[\]"string" \, attribute-list \;
```

1. field-name identifies the name of the screen record field.
2. item-name identifies the form element (name attribute in .42f).
3. string defines the label text, with the % prefix it is a localized string.
4. attribute-list defines the aspect and behavior of the form item.

Form attributes

COLOR, COLOR WHERE, COMMENT, FONTPITCH, HIDDEN, IMAGECOLUMN, JUSTIFY, REVERSE, SIZEPOLICY, STYLE, TAG, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE.

Form field label only: FORMAT, SAMPLE.

Static label only: TEXT.

Style attributes

Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.

Class-specific: textFormat.

Usage

Define the rendering and behavior of a label stack item, with a LABEL element inside a STACK container.

For more details about this item type, see LABEL item type on page 1280.
PAGE stack item

Defines the content of a folder page stack item.

Syntax

```
PAGE [identifier] [ ( attribute-list ) ]
  ↓ scalable-item
  ↓ grouping-item
    leaf-item
      [ ... ]
    END
  ↓ [ ... ]
END
```

1. `identifier` defines the name of the element.
2. `attribute-list` defines the aspect and behavior of the form item.
3. `scalable-item` is a stacked layout items that can grow and shrink.
4. `grouping-item` is a stacked layout grouping element that holds a list of `stack-items`.
5. `leaf-item` is a leaf element of the stacked layout, for widgets with a fixed size (non-scalable).

Form attributes

```
ACTION, COMMENT, HIDDEN, IMAGE, STYLE, TAG, TEXT.
```

Style attributes

```
Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.
Class-specific: none.
```

Can hold

```
FOLDER, GROUP, TABLE.
```

Usage

Use a PAGE stack layout element to group other stack items together.

A PAGE stack item always belongs to a parent FOLDER stack item.

For more details about this item type, see PAGE item type on page 1282.

Example

```
LAYOUT
STACK
  FOLDER folder1 ( STYLE="common" )
    PAGE page1 ( TEXT="Customer info" )
      GROUP
        EDIT FORMONLY.cust_num;
        EDIT FORMONLY.cust_name;
      END
    END
  PAGE page2 ( TEXT="Picture" )
    GROUP
      IMAGE FORMONLY.cust_pic;
```
Related concepts

**FOLDER stack item** on page 1315
Defines a stack area to hold a set of folder pages, in a stack-based layout.

**PHANTOM stack item**
Defines a form field in a stack-based container, that must not be displayed to the end user.

**Syntax**

```
PHANTOM [field-name];
```

1. `field-name` identifies the name of the screen record field.

**Usage**

Define a PHANTOM leaf element in a stack container, to declare a form field to be used by a dialog, without being displayed to the user.

For more details, see Phantom fields on page 1246.

**Example**

```
PHANTOM customer.cust_name;
```

**Related concepts**

**STACK container** on page 1309
The STACK container holds stack items defining a logical alignment of form items.

**PHANTOM item definition** on page 1332
Defines a form field in a grid-based container, that must not be displayed to the end user.

**PROGRESSBAR stack item**
Defines a progress indicator field in a stack-based layout.

**Syntax**

```
PROGRESSBAR field-name [attribute-list];
```

1. `field-name` identifies the name of the screen record field.
2. `attribute-list` defines the aspect and behavior of the form item.

**Form attributes**

COLOR, COLOR WHERE, COMMENT, FONTPITCH, HIDDEN, JUSTIFY, VALUemin, VALUemax, SAMPLE, STYLE, TAG, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE.
Style attributes

Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.

Class-specific: percentageVisible.

Usage

Define the rendering and behavior of a progress bar stack item, with a PROGRESSBAR element inside a STACK container.

For more details about this item type, see PROGRESSBAR item type on page 1283.

Example

```progresobar
workstate.position,
VALUEMIN=-100, VALUEMAX=+100;
```

RADIOGROUP stack item

Defines a mutual exclusive set of option fields, in a stack-based layout.

Syntax

```radioigroup
field-name [ , attribute-list ] ;
```

1. `field-name` identifies the name of the screen record field.
2. `attribute-list` defines the aspect and behavior of the form item.

Form attributes

COLOR, COLOR WHERE, COMMENT, DEFAULT, FONTPITCH, HIDDEN, INCLUDE, ITEMS, JUSTIFY, KEY, NOT NULL, NOENTRY, ORIENTATION, REQUIRED, SAMPLE, SIZEPOLICY, STYLE, TAG, TABINDEX, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, VALIDATE LIKE.

Style attributes

Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.

Class-specific: autoSelectionStart, completionTimeout.

Usage

Define the rendering and behavior of a radio group stack item, with an EDIT element inside a STACK container.

For more details about this item type, see RADIOGROUP item type on page 1284.

Example

```radioigroup
player.level,
ITEMS=((1,"Beginner"),
(2,"Normal"),
(3,"Expert"));
```

SLIDER stack item
Defines a slider element in a stack-based layout.

**Syntax**

```
SLIDER field-name [ , attribute-list ] ;
```

1. `field-name` identifies the name of the screen record field.
2. `attribute-list` defines the aspect and behavior of the form item.

**Form attributes**

COLOR, COLOR WHERE, COMMENT, DEFAULT, FONTPITCH, HIDDEN, INCLUDE, JUSTIFY, NOENTRY, ORIENTATION, SAMPLE, STEP, STYLE, TABINDEX, TAG, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, VALIDATE LIKE, VALUEMIN, VALUEMAX.

**Style attributes**

Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.

Class-specific: none.

**Usage**

Define the rendering and behavior of a slider stack item, with a SLIDER element inside a STACK container.

For more details about this item type, see SLIDER item type on page 1286.

**Example**

```
SLIDER workstate.duration,
  VALUEMIN=0, VALUEMAX=5,
  STEP=1;
```

**SPINEDIT stack item**

Defines a spin box widget to enter integer values in a stack-based layout.

**Syntax**

```
SPINEDIT field-name [ , attribute-list ] ;
```

1. `field-name` identifies the name of the screen record field.
2. `attribute-list` defines the aspect and behavior of the form item.

**Form attributes**

AUTONEXT, COLOR, COLOR WHERE, COMMENT, DEFAULT, FONTPITCH, HIDDEN, IMAGECOLUMN, INCLUDE, JUSTIFY, NOT NULL, NOENTRY, PLACEHOLDER, REQUIRED, SAMPLE, STEP, STYLE, TABINDEX, TAG, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, VALIDATE LIKE, VALUEMIN, VALUEMAX.

**Style attributes**

Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.

Class-specific: none.
Usage
Define the rendering and behavior of a spin edit stack item, with a SPINEDIT element inside a STACK container.
For more details about this item type, see SPINEDIT item type on page 1287.

Example

SPINEDIT command.nbitems, STEP=5;

TABLE stack item
Defines a re-sizable table designed to display a list of records in a stack-based layout.

Syntax

TABLE identifier \(\{\) attribute-list \(\}\)

stack-item

\[\ldots\]\nEND

1. identifier defines the name of the element.
2. attribute-list defines the aspect and behavior of the form item.
3. stack-item is child element in the stack container defining a column in the table.

Form attributes

AGGREGATETEXT, COMMENT, DOUBLECLICK, HIDDEN, FONTPITCH, STYLE, TAG, UNHIDABLECOLUMNS, UNMOVABLECOLUMNS, UNSIZABLECOLUMNS, UNSORTABLECOLUMNS, WANTFIXEDPAGESIZE, WIDTH, HEIGHT.

Style attributes

Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.

Class-specific: allowWebSelection, forceDefaultSettings, headerAlignment, headerHidden, highlightColor, highlightCurrentCell, highlightCurrentRow, highlightTextColor, resizeFillsEmptySpace, reduceFilter, rightFrozenColumns, rowActionTrigger, showGrid, summaryLineAlwaysAtBottom, tableType.

Can hold

BUTTONEDIT, CHECKBOX, COMBOBOX, DATEEDIT, DATETIMEEDIT, EDIT, IMAGE, LABEL, PROGRESSBAR, PHANTOM, SLIDER, SPINEDIT, TIMEEDIT, RADIOGROUP.

Usage
The TABLE stack layout element defines a list view element in a stack-based layout.
To create a table view in a stacked layout, define the following elements in the form file:

1. The layout of the list, with a TABLE stack item.
2. The columns definitions as stack items inside the TABLE item.

Note: The TABLE item must have an identifier, that will be used as screen-array in list dialogs.
For more details about table view programming, see Table views on page 1853

Example

TABLE custlist (STYLE="regular")
EDIT ...
TEXTEDIT stack item
Defines a multi-line edit field in a stack-based layout.

Syntax
```
TEXTEDIT [identifier], attribute-list;
```

1. `field-name` identifies the name of the screen record field.
2. `attribute-list` defines the aspect and behavior of the form item.

Form attributes
```
COLOR, COLOR WHERE, COMMENT, DEFAULT, DOWNSHIFT, FONTPITCH, HEIGHT, HIDDEN, INCLUDE,
JUSTIFY, KEY, NOT NULL, NOENTRY, PLACEHOLDER, PROGRAM, REQUIRED, SAMPLE, SCROLL,
SCROLLBARS, STYLE, STRETCH, TAG, TABINDEX, TITLE, UPSHIFT, VALIDATE LIKE, WANTTABS,
WANTNORETURNS.
```

Style attributes

Common: `backgroundColor`, `border`, `fontFamily`, `fontSize`, `fontStyle`, `fontWeight`,
`localAccelerators`, `showAcceleratorInToolTip`, `textColor`, `textDecoration`.

Class-specific: `integratedSearch`, `showEditToolBox`, `spellCheck`, `textFormat`, `wrapPolicy`.

Usage
Define the rendering and behavior of a text edit stack item, with a `TEXTEDIT` element inside a `STACK` container.

**Note:** The `TEXTEDIT` stack item can be used inside a stack container like a group, or as root element of the `STACK` container. When used directly under the `STACK` container, the `TEXTEDIT` stack item must be the only element in the container. It is rendered as a scalable form item that can stretch to fit the front-end screen size.

For more details about this item type, see `TEXTEDIT item type` on page 1289.

Example
```
TEXTEDIT customer.cust_address, HEIGHT=3, REQUIRED;
```

Related concepts

`STACK container` on page 1309
The `STACK` container holds stack items defining a logical alignment of form items.

`TEXTEDIT item definition` on page 1347
Defines a multi-line edit field, in a grid-based layout.

TIMEEDIT stack item
Defines a line-edit with a clock widget to pick a time, in a stack-based layout.

Syntax
```
TIMEEDIT field-name, attribute-list;
```

1. `field-name` identifies the name of the screen record field.
2. `attribute-list` defines the aspect and behavior of the form item.
Form attributes
AUTONEXT, COLOR, COLOR WHERE, COMMENT, DEFAULT, FONTPITCH, HIDDEN, IMAGECOLUMN, INCLUDE, JUSTIFY, NOT NULL, NOENTRY, PLACEHOLDER, REQUIRED, SAMPLE, STYLE, TABINDEX, TAG, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, VALIDATE LIKE.

Style attributes
Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.
Class-specific: none.

Usage
Define the rendering and behavior of a time edit stack item, with a TIMEEDIT element inside a STACK container.
For more details about this item type, see TIMEEDIT item type on page 1290.

Example
TIMEEDIT package.arrtime;

WEBCOMPONENT stack item
Defines a generic form field that can receive an external widget, in a stack-based layout.

Syntax
WEBCOMPONENT [identifier] [, attribute-list] ;

1. field-name identifies the name of the screen record field.
2. attribute-list defines the aspect and behavior of the form item.

Form attributes
COLOR, COLOR WHERE, COMPONENTTYPE, COMMENT, DEFAULT, FONTPITCH, HEIGHT, HIDDEN, INCLUDE, JUSTIFY, NOT NULL, NOENTRY, PROPERTIES, REQUIRED, SCROLLBARS, SIZEPOLICY, STYLE, STRETCH, TAG, TABINDEX, TITLE, VALIDATE LIKE, WIDTH.

Style attributes
Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.
Class-specific: none.

Usage
Define the rendering and behavior of a web component stack item, with a WEBCOMPONENT element inside a STACK container.

Note: The WEBCOMPONENT stack item can be used inside a stack container like a group, or as root element of the STACK container: When used directly under the STACK container, the WEBCOMPONENT stack item must be the only element in the container. It will be rendered as a scalable form item that can stretch to fit the front-end screen size.
For more details about this item type, see WEBCOMPONENT item type on page 1293.

Example
-- URL-based web component (recommended):
WEBCOMPONENT FORMONLY.mymap;
Related concepts
Web component in stack-based layout on page 1925

SCROLLGRID container
Defines a scrollable grid view widget, in a grid-based layout.

Syntax

SCROLLGRID [identifier] (attribute [, ...]) {\n  row-template [...].
} END

where row-template is a text block containing:

  text  item-tag [...]

1. text is literal text that will appear in the form as a static label.
2. item-tag defines the position and length of a form item.

Form attributes

COMMENT, DOUBLECLICK, FONTPITCH, STYLE, TAG, HIDDEN, WANTFIXEDPAGESIZE.

Style attributes

Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.
Class-specific: customWidget, highlightColor, highlightCurrentCell, highlightCurrentRow, highlightTextColor, rowActionTrigger, itemsAlignment.

Usage

The SCROLLGRID container declares a formatted text block defining the dimensions and the position of the logical elements of a screen for a multi-record presentation.

Note: Avoid Tab characters (ASCII 9) inside the curly-brace delimited area. If used, Tab characters will be replaced by 8 blanks by fglform.

For more details about this item type, see SCROLLGRID item type on page 1285.

Example 1: Resizable scrollgrid (using WANTFIXEDPAGESIZE=NO):

SCROLLGRID (WANTFIXEDPAGESIZE=NO)
{\n  Id: [f001] Name: [f002] \n  Address: [f003] \n} END
Example 2: Scrollgrid with fixed page size, using four rows:

```plaintext
SCROLLGRID
{
  Id: [f001]  Name: [f002]
  Address: [f003]

  Id: [f001]  Name: [f002]
  Address: [f003]

  Id: [f001]  Name: [f002]
  Address: [f003]

  Id: [f001]  Name: [f002]
  Address: [f003]
}
END
```

Related concepts

Grid-based layout on page 1418
A form file can define a grid-based layout within a tree of layout items.

GRID container on page 1308
Defines a layout area based on a grid of cells.

TABLE container
Defines a re-sizable table designed to display a list of records.

Syntax

```plaintext
TABLE [identifier] (attribute [, ]...
{
  title [....]  
  [col-name [, ]... ]
[....]
  [aggr-name [, ]... ]
}
END
```

1. `identifier` defines the name of the element.
2. `attribute` is an attribute for the element.
3. `title` is the text to be displayed as column title.
4. `col-name` is an identifier that references a form field.
5. `aggr-name` is an identifier that references an aggregate Field.

Form attributes

AGGREGATETEXT, COMMENT, DOUBLECLICK, HIDDEN, FONTPITCH, STYLE, TAG, UNHIDABLECOLUMNS, UNMOVABLECOLUMNS, UNSIZABLECOLUMNS, UNSORTABLECOLUMNS, WANTFIXEDPAGESIZE, WIDTH, HEIGHT.

Style attributes

Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.

Class-specific: allowWebSelection, forceDefaultSettings, headerAlignment, headerHidden, highlightColor, highlightCurrentCell, highlightCurrentRow, highlightTextColor,
Usage:
The TABLE container defines a list view element in a grid-based layout.

To create a table view in a grid layout, define the following elements in the form file:

1. The layout of the list, with a TABLE container in the LAYOUT section.
2. The column data types and field properties, in the ATTRIBUTES section.
3. The field list definition to group form fields together with a screen array, in the INSTRUCTIONS section.

For more details about this item type, see TABLE item type on page 1288.

Example

```plaintext
SCHEMA videolab
LAYOUT ( TEXT="Customer list" )
TABLE ( TAG="normal" )
{
    [c1 | c2 | c3 | c4 ]
    [c1 | c2 | c3 | c4 ]
    [c1 | c2 | c3 | c4 ]
    [c1 | c2 | c3 | c4 ]
}
END
END
TABLES
    customer
END
ATTRIBUTES
    EDIT c1 = customer.cust_num, TITLE="Num";
    EDIT c2 = customer.cust_name, TITLE="Customer name";
    EDIT c3 = customer.cust_cdate, TITLE="Date";
    CHECKBOX c4 = customer.cust_status, TITLE="Status";
END
INSTRUCTIONS
    SCREEN RECORD custlist( cust_num, cust_name, cust_cdate, cust_status )
END
```

Related concepts

Grid-based layout on page 1418
A form file can define a grid-based layout within a tree of layout items.

Layout tags on page 1254
Layout tags define layout areas for containers inside the frame of a grid-based container.

Item tags on page 1259
Item tags define the position and size in a grid-based container.

TREE container
The TREE container defines the presentation of a list of ordered records in a tree-view widget.

Syntax

```plaintext
TREE |identifier| [ ( attribute [, ...] ) ] 
{ 
    title [....]
    [name_column |identifier [, ...] ]
    [....]
}
```
1. **identifier** defines the name of the element.
2. **attribute** is an attribute for the element.
3. **title** is the text to be displayed as column title.
4. **name_column** is a mandatory column referencing a form item defining the node text.
5. **identifier** references a form item.

**Form attributes**

`COMMENT, DOUBLECLICK, HIDDEN, FONTPITCH, STYLE, TAG, UNHIDABLECOLUMNS, UNMOVABLECOLUMNS, UNSIZABLECOLUMNS, UNSORTABLECOLUMNS, WANTFIXEDPAGESIZE, WIDTH, HEIGHT, PARENTIDCOLUMN, IDCOLUMN, EXPANDEDCOLUMN, ISNODECOLUMN, IMGAGEEXPANDED, IMAGECOLLAPSED, IMAGELEAF`.

**Style attributes**

**Common:** `backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration`.

**Class-specific:** `allowWebSelection, forceDefaultSettings, headerAlignment, headerHidden, highlightColor, highlightCurrentRow, highlightTextColor, resizeFillsEmptySpace, rightFrozenColumns, rowActionTrigger, showGrid, tableType`.

**Usage**

To create a tree view in a grid-based layout, you must define the following elements in the form file:

1. The layout of the tree-view, with a `TREE` container in the LAYOUT section.
2. The column data types and field properties, in the ATTRIBUTES section.
3. The field list definition to group form fields together with a screen array, in the INSTRUCTIONS section.

For more details about this item type, see [TREE item type](#) on page 1292.

**Example**

```
LAYOUT
GRID
{
<Tree t1>
Name       Index
[c1 |c2 ]
[c1 |c2 ]
[c1 |c2 ]
[c1 |c2 ]
END
END

ATTRIBUTES
LABEL c1 = FORMONLY.name;
LABEL c2 = FORMONLY.idx;
PHANTOM FORMONLY.pid;
PHANTOM FORMONLY.id;
TREE t1: tree1
    PARENTIDCOLUMN = pid,
    IDCOLUMN = id;
END

INSTRUCTIONS
SCREEN RECORD sr_tree(name, pid, id, idx);
```
Related concepts

Grid-based layout on page 1418
A form file can define a grid-based layout within a tree of layout items.

Layout tags on page 1254
Layout tags define layout areas for containers inside the frame of a grid-based container.

Item tags on page 1259
Item tags define the position and size in a grid-based container.

TABLES section
Defines the list of database tables referenced by form field definitions.

Syntax

```
TABLES
  [ alias = [database[@dbserver]:]owner. ] table [, . . . ]
[END]
```

1. `alias` represents an alias name for the given table.
2. `table` is the name of the database table.
3. `database` is the name of the database of the table (see warnings).
4. `dbserver` identifies the Informix® database server (INFORMIXSERVER)
5. `owner` is the name of the table owner (see warnings).

Usage

The TABLES section lists every database table or view referenced in the form specification file. This section is mandatory when form fields reference database columns defined in the database schema file.

The TABLE section must appear in the sequence described in form file structure.

The END keyword is optional.

The SCHEMA section must also exist to define the database schema.

Field identifiers in programs or in other sections of the form specification file can reference screen fields as `column`, `alias.column`, or `table.column`.

The same `alias` must also appear in screen interaction statements of programs that reference screen fields linked to the columns of a table that has an `alias`.

If a table requires the name of an `owner` or of a `database` as a qualifier, the TABLES section must also declare an alias for the table. The `alias` can be the same identifier as `table`.

For backward compatibility with the Informix® form specification, the comma separator is optional and the `database`, `dbserver` and `owner` specifications are ignored.

Example

```
SCHEMA stores
LAYOUT
GRID
{
   ...
}
END
TABLES
customer, orders
END
```
ATTRIBUTES

... 
END

Related concepts
Form fields on page 1242

Form fields are form elements designed for data input and/or data display.

ATTRIBUTES section
The ATTRIBUTES section describes properties of grid-based layout elements used in the form.

Syntax

<table>
<thead>
<tr>
<th>ATTRIBUTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>form-field-definition</td>
</tr>
<tr>
<td>phantom-field-definition</td>
</tr>
<tr>
<td>form-item-definition</td>
</tr>
</tbody>
</table>

[END]

where form-field-definition is:

| item-type item-tag = field-name ↓, attribute-list ↓; |

where phantom-field-definition is:

| PHANTOM field-name; |

where form-item-definition is:

| item-type item-tag: item-name ↓, attribute-list ↓; |

1. item-type defines the type of the Form Item.
2. item-tag is the name of the screen element used in the LAYOUT section.
3. field-name defines the name of the screen record field.
4. item-name identifies the form item that is not a form field containing data.
5. attribute-list defines the aspect and behavior of the form item.

where attribute-list is:

| attribute ↓, ... |

1. The attribute list is a comma-separated list of attributes.

where attribute is:

| attribute-name ↓ = ↓ value ↓ value-list ↓ ↓ |

1. attribute identifies the attribute of the form item.

where value-list is:

| ( ↓ value ↓ sub-value-list ↓ ↓, ...) |

1. value is a string, date or numeric literal, or predefined constant like TODAY.
2. sub-value-list is a set of values separated by comma, to support subset definitions as in "(1, (21, 22), (31, 32, 33))".
Usage

The **ATTRIBUTES** section is required to define the attributes for the form items used in grid-based containers of the **LAYOUT** section.

The **ATTRIBUTES** section must appear in the sequence described in **form file structure**.

The **END** keyword is optional.

Every item-tag used in the **LAYOUT** section must get an item definition in the **ATTRIBUTES** section.

A form item definition is associated by name to an item tag or layout tag defined in the grid-based container.

In order to define a form field, the form item definition must use the equal sign notation to associate a screen record field with the form item. If the form item is not associated with a screen record field (for example, a push button), you must use the colon notation.

To match the complete structure of a database table record, additional fields can be defined as phantom fields, when no corresponding item tag is used in the layout.

Form item definitions can optionally include an **attribute-list** to specify the appearance and behavior of the item. For example, you can define acceptable input values, on-screen comments, and default values for fields.

When no screen record is defined in the **INSTRUCTION** section, a default screen record is built for each set of form items declared with the same table name.

The order in which you list the form items determines the order of fields in the default screen records that the form compiler creates for each table.

To define form items as form fields, you are not required to specify **table** unless the name **column** is not unique within the form specification. However, it is recommended that you always specify **table.column** rather than the unqualified **column** name. As you can refer to field names collectively through a screen record built upon all the fields linked to the same table, your forms might be easier to work with if you specify **table** for each field.

When used in a table, some widgets are rendered only when the user enters in the field. For example **RadioGroup**, **CheckBox**, **ComboBox**, **ProgressBar**.

Example

```
SCHEMA game
LAYOUT
GRID
{
  ...
}
END
TABLES
player
END
ATTRIBUTES
EDIT f001 = player.name, REQUIRED,
  COMMENT="Enter player's name";
EDIT f002 = player.ident, NOENTRY;
COMBOBOX f003 = player.level, NOT NULL,
  ITEMS=((1,"Beginner"), (2,"Normal"),(3,"Expert"));
CHECKBOX f004 = FORMONLY.winner,
  VALUECHECKED=1, VALUEUNCHECKED=0,
  TEXT="Winner";
BUTTON b1: print, TEXT="Print Report";
GROUP g1: print, TEXT="Description";
END
```

Related concepts

*Form items* on page 1238
The concept of *form item* includes all elements used in the definition of a form.

**AGGREGATE item definition**
Defines screen-record fields that hold computed values to be displayed as footer cells in a TABLE container.

**Syntax**

```
AGGREGATE item-tag = field-name [ , attribute-list ];
```

1. *item-tag* is an identifier that defines the name of the item tag in the layout section.
2. *field-name* identifies the name of the screen record field.
3. *attribute-list* defines the aspect and behavior of the form item.

**Attributes**

`AGGREGATETEXT, AGGREGATETYPE`.

**Usage**

Aggregate fields must be declared with an AGGREGATE element in the ATTRIBUTES section.

**Important:** This feature is not supported on mobile platforms.

For more details see Aggregate fields on page 1248.

**Example**

```
AGGREGATE total = FORMONLY.o_total,
                   AGGREGATETEXT = "Total:",
                   AGGREGATETYPE = SUM;
```

**Related concepts**

Summary lines in tables on page 1866
Table views can display a summary line, to show aggregate values for columns.

**PHANTOM item definition**
Defines a form field in a grid-based container, that must not be displayed to the end user.

**Syntax**

```
PHANTOM [field-name] ;
```

1. *field-name* identifies the name of the screen record field.

**Usage**

Define a phantom form field (that will be used by a dialog, but not displayed in the form layout), with a PHANTOM element in the ATTRIBUTES section.

For more details, see Phantom fields on page 1246.

**Example**

```
PHANTOM customer.cust_name;
```

**Related concepts**

Database column fields on page 1243
Form fields defined with a table and column name get data type from the database schema file.

Formonly fields on page 1245
FORMONLY form fields define their data type explicitly, with or without referencing a database columns.

**PHANTOM stack item** on page 1319
Defines a form field in a stack-based container, that must not be displayed to the end user.

**BUTTON item definition**
Defines a push-button field that can trigger an action, in a grid-based layout.

**Syntax**

```
BUTTON item-tag: item-name [ attribute-list ] ;
```

1. `item-tag` is an identifier that defines the name of the item tag in the layout section.
2. `item-name` defines the form item name and the action name.
3. `attribute-list` defines the aspect and behavior of the form item.

**Form attributes**

`COMMENT, DISCLOSUREINDICATOR, FONTPITCH, HIDDEN, IMAGE, SAMPLE, SIZEPOLICY, STYLE, TABINDEX, TAG, TEXT.`

**Style attributes**

Common: `backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration`.

Class-specific: `alignment, buttonType, scaleIcon`.

**Usage**

Defines the rendering and behavior of a button item tag, with a `BUTTON` element in the ATTRIBUTES section.

For more details about this item type, see **BUTTON item type** on page 1266.

**Example**

```
LAYOUT GRID
{
[btn1 ]
...
}
END END

ATTRIBUTES
BUTTON btn1: print, TEXT="Print Report", IMAGE="printer";
...
```

**BUTTONEDIT item definition**

Defines a line-edit field with a push-button that can trigger an action, in a grid-based layout.

**Syntax**

```
BUTTONEDIT item-tag = field-name [ attribute-list ] ;
```

1. `item-tag` is an identifier that defines the name of the item tag in the layout section.
2. `field-name` identifies the name of the screen record field.
3. `attribute-list` defines the aspect and behavior of the form item.
Form attributes

ACTION, AUTONEXT, CENTURY, COLOR, COMPLETER, COLOR WHERE, COMMENT, DEFAULT, DISPLAY LIKE, DOWNSHIFT, FONTPITCH, HIDDEN, FORMAT, IMAGE, INCLUDE, INVISIBLE, JUSTIFY, KEY, KEYBOARDHINT, NOT NULL, NOTEDITABLE, NOENTRY, PICTURE, PLACEHOLDER, PROGRAM, REVERSE, SAMPLE, SCROLL, STYLE, REQUIRED, TAG, TITLE, TABINDEX, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, UPSHIFT, VALIDATE LIKE, VERIFY.

Style attributes

Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.

Class-specific: scaleIcon.

Usage

Define the rendering and behavior of a buttonedit item tag, with a BUTTONEDIT element in the ATTRIBUTES section.

For more details about this item type, see BUTTONEDIT item type on page 1267.

Example

```
LAYOUT
GRID
{
    [f1]
    ...
}
END
END

ATTRIBUTES
BUTTONEDIT f1 = customer.state,
    REQUIRED, IMAGE="smiley", ACTION=zoom;
    ...
```

CANVAS item definition

The CANVAS form item defines an area in which you can draw shapes, in a grid-based layout.

Syntax

Important: This feature is deprecated, its use is discouraged although not prohibited.

```
CANVAS item-tag: item-name [, attribute-list ] ;
```

1. item-tag is an identifier that defines the name of the item tag in the layout section.
2. item-name identifies the form item.
3. attribute-list defines the aspect and behavior of the form item.

Form attributes

COMMENT, HIDDEN, TAG.

Style attributes

Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.
Class-specific: none.

**Usage**

Define the rendering and behavior of a canvas drawing area item tag, with a `CANVAS` element in the `ATTRIBUTES` section.

**Note:** The `CANVAS` feature is deprecated, consider using the built-in `fglsvgcanvas WEBCOMPONENT`.

**Example**

```
LAYOUT
GRID
{
  [cvs1]
  [   ]
  [   ]
  ...
}
END
END

ATTRIBUTES
CANVAS cvs1: canvas1;
...
```

**Related concepts**

- **Canvases** on page 2013
  Canvases are form drawing areas.

- **CHECKBOX item definition**
  Defines a boolean or three-state checkbox field, in a grid-based layout.

**Syntax**

```
CHECKBOX item-tag = field-name [, attribute-list ] ;
```

1. `item-tag` is an identifier that defines the name of the item tag in the layout section.
2. `field-name` identifies the name of the screen record field.
3. `attribute-list` defines the aspect and behavior of the form item.

**Form attributes**

```
COLOR, COLOR WHERE, COMMENT, DEFAULT, FONTPITCH, HIDDEN, INCLUDE, JUSTIFY, KEY, NOT NULL, NOENTRY, REQUIRED, SAMPLE, SIZEPOLICY, STYLE, TAG, TABINDEX, TEXT, TITLE, VALIDATE LIKE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, VALUECHECKED, VALUEUNCHECKED.
```

**Style attributes**

```
Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.
```

Class-specific: `customWidget`, `iosCheckBoxOnTintColor`.

**Usage**

Define the rendering and behavior of a checkbox item tag, with a `CHECKBOX` element in the `ATTRIBUTES` section.

For more details about this item type, see **CHECKBOX item type** on page 1269.
Example

```
LAYOUT
GRID
{
    [f1         ]
    ...
}
END
END
```

**COMBOBOX item definition**

Defines a COMBOBOX item in a grid-based layout, in a grid-based layout.

**Syntax**

```
COMBOBOX item-tag = field-name [, attribute-list ] ;
```

1. `item-tag` is an identifier that defines the name of the item tag in the layout section.
2. `field-name` identifies the name of the screen record field.
3. `attribute-list` defines the aspect and behavior of the form item.

**Form attributes**

- COLOR, COLOR WHERE, COMMENT, DEFAULT, DOWNSHIFT, FONTPITCH, HIDDEN, KEY, INCLUDE, INITIALIZER, ITEMS, JUSTIFY, NOT NULL, NOENTRY, PLACEHOLDER, QUERYEDITABLE, REQUIRED, SAMPLE, SCROLL, SIZEPOLICY, STYLE, UPSHIFT, TAG, TABINDEX, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, TITLE, VALIDATE LIKE.

**Style attributes**

Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.

Class-specific: autoSelectionStart, comboboxCompleter, completionTimeout.

**Usage**

Define the rendering and behavior of a combobox item, with a COMBOBOX element in the ATTRIBUTES section.

For more details about this item type, see COMBOBOX item type on page 1270.

**Example**

```
LAYOUT
GRID
{
    [f1         ]
    ...
}
END
END
```
Related concepts

Filling a COMBOBOX item list on page 1735

The item list of COMBOBOX fields can be initialized at runtime.

DATEEDIT item definition

Defines a line-edit field with a calendar widget to pick a date, in a grid-based layout.

Syntax

```
DATEEDIT item-tag = field-name [ , attribute-list ] ;
```

1. `item-tag` is an identifier that defines the name of the item tag in the layout section.
2. `field-name` identifies the name of the screen record field.
3. `attribute-list` defines the aspect and behavior of the form item.

Form attributes

```
AUTONEXT, CENTURY, COLOR, COLOR WHERE, COMMENT, DEFAULT, FONTPITCH, FORMAT, HIDDEN,
IMAGECOLUMN, INCLUDE, JUSTIFY, KEY, NOT NULL, NOENTRY, PLACEDHOLDER, REQUIRED, SAMPLE,
STYLE, TAG, TABINDEX, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, VALIDATE LIKE.
```

Style attributes

Common: `backgroundColor`, `border`, `fontFamily`, `fontSize`, `fontStyle`, `fontWeight`,
`localAccelerators`, `showAcceleratorInToolTip`, `textColor`, `textDecoration`.

Class-specific: `buttonIcon`, `calendarType`, `daysOff`, `firstDayOfWeek`, `showCurrentMonthOnly`,
`showGrid`, `showWeekNumber`.

Usage

Define the rendering and behavior of a date edit item tag, with a DATEEDIT element in the ATTRIBUTES section.

For more details about this item type, see DATEEDIT item type on page 1272.

Example

```
LAYOUT
GRID
{
    [f1 ]
    ...
}
END
END

ATTRIBUTES
DATEEDIT f1 = order.shipdate;
...
```
**DATETIMEEDIT item definition**
Defines a line-edit field with a calendar widget to pick a datetime, in a grid-based layout.

**Syntax**

```
DATETIMEEDIT item-tag = field-name [ , attribute-list ];
```

1. *item-tag* is an identifier that defines the name of the item tag in the layout section.
2. *field-name* identifies the name of the screen record field.
3. *attribute-list* defines the aspect and behavior of the form item.

**Form attributes**

AUTONEXT, COLOR, COLOR WHERE, COMMENT, DEFAULT, FONTPITCH, HIDDEN, INCLUDE, IMAGECOLUMN, JUSTIFY, NOT NULL, NOENTRY, PLACEHOLDER, REQUIRED, SAMPLE, STYLE, TABINDEX, TAG, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, VALIDATE LIKE.

**Style attributes**

Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.

Class-specific: buttonIcon, calendarType, daysOff, enableCalendar, firstDayOfWeek, showCurrentMonthOnly, showGrid, showWeekNumber.

**Usage**

Define the rendering and behavior of a date edit *item tag*, with a DATETIMEEDIT element in the ATTRIBUTES section.

For more details about this item type, see DATETIMEEDIT item type on page 1273.

**Example**

```
LAYOUT
GRID
{
    [f1       ]
    ... 
}
END
END

ATTRIBUTES
DATETIMEEDIT f1 = package.modts;
...
```

**EDIT item definition**
Defines a simple line-edit field, in a grid-based layout.

**Syntax**

```
EDIT item-tag = field-name [ , attribute-list ];
```

1. *item-tag* is an identifier that defines the name of the item tag in the layout section.
2. *field-name* identifies the name of the screen record field.
3. *attribute-list* defines the aspect and behavior of the form item.
Form attributes

AUTONEXT, CENTURY, COLOR, COLOR WHERE, COMMENT, COMPLETER, DEFAULT, DISPLAY LIKE, DOWNSHIFT, FONTPITCH, FORMAT, HIDDEN, IMAGECOLUMN, INCLUDE, INVISIBLE, JUSTIFY, KEY, KEYBOARDHINT, NOT NULL, NOENTRY, PICTURE, PLACED, PLACEHOLDER, PROGRAM, REQUIRED, REVERSE, SAMPLE, STYLE, SCROLL, TAG, TABINDEX, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, UPSHIFT, VALIDATE LIKE, VERIFY.

Style attributes

Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.

Class-specific: dataTypeHint, spellCheck.

Usage

Define the rendering and behavior of an edit item tag, with an EDIT element in the ATTRIBUTES section.

For more details about this item type, see EDIT item type on page 1274.

Example

```
LAYOUT
GRID
{
 f1
 }
...
END
END

ATTRIBUTES
EDIT f1 = customer.cust_state,
   REQUIRED,
   COMMENT = "customer.cust_state.comment",
   INCLUDE=(0,1,2);
...
```

GROUP item definition

Defines a group-box layout tag, in a grid-based layout.

Syntax

```
GROUP layout-tag: item-name __ , attribute-list __ ;
```

1. layout-tag is an identifier that defines the name of the layout tag.
2. item-name identifies the form item.
3. attribute-list defines the aspect and behavior of the form item.

Form attributes

COMMENT, FONTPITCH, GRIDCHILDRENINPARENT, HIDDEN, STYLE, TAG, TEXT.

Style attributes

Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.

Class-specific: collapsible, collapserPosition, initiallyCollapsed.
Usage
Define the rendering and behavior of a group layout tag, with a GROUP element in the ATTRIBUTES section.
For more details about this item type, see GROUP item type on page 1276.

Example

```
LAYOUT
GRID
{
 <GROUP g1     >
   Num: [f001    ]
   ...
   }
END
END

ATTRIBUTES
GROUP g1: group1,
   TEXT="Description",
   GRIDCHILDRENINPARENT;
   ...
```

Related concepts
GROUP container on page 1306
Defines a layout area to group other layout elements together, in a grid-based layout.

GROUP stack item on page 1315
Defines a stack area to group other layout elements together, in a stack-based layout.

Examples on page 1414
Form definition (.per) examples.

IMAGE item definition
Defines an area that can display an image resource, in a grid-based layout.

Syntax 1: Defining a form field image

```
IMAGE item-tag = field-name ↓, attribute-list ↓;
```

Syntax 2: Defining a static image

```
IMAGE item-tag: item-name ↓, attribute-list ↓;
```

1. item-tag is an identifier that defines the name of the item tag in the layout section.
2. field-name identifies the name of the screen record field.
3. item-name identifies the form item for a static image.
4. attribute-list defines the aspect and behavior of the form item.

Form attributes

ACTION, AUTOSCALE, COMMENT, HEIGHT, HIDDEN, SIZEPOLICY, STYLE, STRETCH, TAG, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, WIDTH.

Image field only: JUSTIFY, SAMPLE.

Static image only: IMAGE.
Style attributes

Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.

Class-specific: alignment, imageContainerType.

Usage

Define the rendering and behavior of an image item tag, with an IMAGE element in the ATTRIBUTES section.

For more details about this item type, see IMAGE item type on page 1279.

Example

```
LAYOUT
GRID
{
  [f1                     
  [                      
  [                      
  [                      
  ... 
  ]
}
END
END

ATTRIBUTES
IMAGE f1 = cars.picture, SIZEPOLICY=FIXED, AUTOSCALE, COMMENT="Picture of the car";
...
```

LABEL item definition

Defines a simple text area to display a read-only value, in a grid-based layout.

Syntax 1: Defining a form field label

```
LABEL item-tag = field-name [, attribute-list ] ;
```

Syntax 2: Defining a static label

```
LABEL item-tag: item-name [, attribute-list ] ;
```

1. item-tag is an identifier that defines the name of the item tag in the layout section.
2. field-name identifies the name of the screen record field.
3. item-name identifies the form element (name attribute in .42f) of a static label.
4. attribute-list defines the aspect and behavior of the form item.

Attributes

COLOR, COLOR WHERE, COMMENT, FONTPITCH, HIDDEN, IMAGECOLUMN, JUSTIFY, REVERSE, SIZEPOLICY, STYLE, TAG, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE.

Form field label only: FORMAT, SAMPLE.

Static label only: TEXT.
Style attributes

Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.

Class-specific: textFormat.

Usage

Define the rendering and behavior of an label item tag, with an LABEL element in the ATTRIBUTES section.

For more details about this item type, see LABEL item type on page 1280.

Example

LAYOUT
GRID
{
[11 : f1]
...
}
END
END

ATTRIBUTES
LABEL 11: label1, TEXT="Desc:"; -- This is a static label
LABEL f1 = vehicle.description; -- This is a form field label
...

Related concepts

Hbox tags on page 1261
Hbox tags group several item tags within the same horizontal layout box, inside a grid-based container (GRID).

PROGRESSBAR item definition

Defines a progress indicator field, in a grid-based layout.

Syntax

PROGRESSBAR item-tag = field-name [ , attribute-list ] ;

1. item-tag is an identifier that defines the name of the item tag in the layout section.
2. field-name identifies the name of the screen record field.
3. attribute-list defines the aspect and behavior of the form item.

Form attributes

COLOR, COLOR WHERE, COMMENT, FONTPITCH, HIDDEN, JUSTIFY, VALUemin, VALUemax, SAMPLE, STYLE, TAG, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE.

Style attributes

Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.

Class-specific: percentageVisible.

Usage

Define the rendering and behavior of a progress bar item tag, with a PROGRESSBAR element in the ATTRIBUTES section.
For more details about this item type, see PROGRESSBAR item type on page 1283.

**Example**

```plaintext
LAYOUT
GRID
{
    [f1     ]
    ...  
}
END
END

ATTRIBUTES
PROGRESSBAR f1 = workstate.position,
    VALUEMIN=-100, VALUEMAX=+100;
...
```

**RADIOGROUP item definition**

Defines a mutually-exclusive set of option fields, in a grid-based layout.

**Syntax**

```plaintext
RADIOGROUP item-tag = field-name \[attribute-list] ;
```

1. *item-tag* is an identifier that defines the name of the item tag in the layout section.
2. *field-name* identifies the name of the screen record field.
3. *attribute-list* defines the aspect and behavior of the form item.

**Form attributes**

COLOR, COLOR WHERE, COMMENT, DEFAULT, FONTPITCH, HIDDEN, INCLUDE, ITEMS, JUSTIFY, KEY, NOT NULL, NOENTRY, ORIENTATION, REQUIRED, SAMPLE, SIZEPOLICY, STYLE, TAG, TABINDEX, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, VALIDATE LIKE.

**Style attributes**

Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.

Class-specific: autoSelectionStart, completionTimeout.

**Usage**

Define the rendering and behavior of a radio group *item tag*, with a RADIOGROUP element in the ATTRIBUTES section.

For more details about this item type, see RADIOGROUP item type on page 1284.

**Example**

```plaintext
LAYOUT
GRID
{
    [f1     ]
    ...  
}
END
END
```
ATTRIBUTES
RADIOGROUP f1 = player.level,
   ITEMS=((1,"Beginner"),
      (2,"Normal"),
      (3,"Expert"));
...

SCROLLGRID item definition
Defines a scrollgrid layout tag in a grid-based layout.

Syntax

SCROLLGRID layout-tag: item-name [, attribute-list ];

1. layout-tag is an identifier that defines the name of the layout tag.
2. item-name identifies the form item.
3. attribute-list defines the aspect and behavior of the form item.

Form attributes

COMMENT, DOUBLECLICK, FONTPITCH, GRIDCHILDRENINPARENT, HIDDEN, INITIALPAGESIZE, STYLE, TAG.

Style attributes

Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.
Class-specific: customWidget, highlightColor, highlightCurrentCell, highlightCurrentRow, highlightTextColor, rowActionTrigger, itemsAlignment.

Usage

The SCROLLGRID form item type to specify the attributes of a scrollgrid container defined with a layout tag.

For more details about this item type, see SCROLLGRID item type on page 1285.

Example

LAYOUT
GRID
{
   <SCROLLGRID sg1>
      [f001] 
   ...

   }
END
END

ATTRIBUTES
SCROLLGRID sg1: scrollgrid1,
   GRIDCHILDRENINPARENT;
**SLIDER item definition**
Defines a slider element, in a grid-based layout.

**Syntax**

```
SLIDER item-tag = field-name [, attribute-list ] ;
```

1. *item-tag* is an identifier that defines the name of the item tag in the layout section.
2. *field-name* identifies the name of the screen record field.
3. *attribute-list* defines the aspect and behavior of the form item.

**Form attributes**

COLOR, COLOR WHERE, COMMENT, DEFAULT, FONTPITCH, HIDDEN, INCLUDE, JUSTIFY, NOENTRY, ORIENTATION, SAMPLE, STEP, STYLE, TABINDEX, TAG, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, VALIDATE LIKE, VALUEMIN, VALUEMAX.

**Style attributes**

Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.

Class-specific: none.

**Usage**

Define the rendering and behavior of a slider *item tag*, with an SLIDER element in the ATTRIBUTES section.

For more details about this item type, see SLIDER item type on page 1286.

**Example**

```
LAYOUT
GRID
{
  [f1       ]
  ...
}
END
END

ATTRIBUTES
SLIDER f1 = workstate.duration,
  VALUEMIN=0, VALUEMAX=50,
  STEP=1;
  ...
```

**SPINEDIT item definition**
Defines a spin box widget to enter integer values, in a grid-based layout.

**Syntax**

```
SPINEDIT item-tag = field-name [, attribute-list ] ;
```

1. *item-tag* is an identifier that defines the name of the item tag in the layout section.
2. *field-name* identifies the name of the screen record field.
3. *attribute-list* defines the aspect and behavior of the form item.
### Form attributes

`AUTONEXT, COLOR, COLOR WHERE, COMMENT, DEFAULT, FONTPITCH, HIDDEN, IMAGECOLUMN, INCLUDE, JUSTIFY, NOT NULL, NOENTRY, PLACEHOLDER, REQUIRED, SAMPLE, STEP, STYLE, TABINDEX, TAG, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, VALIDATE LIKE, VALUEMIN, VALUEMAX`.

### Style attributes

Common: `backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration`.

Class-specific: none.

### Usage

Define the rendering and behavior of a spin edit item tag, with a SPINEDIT element in the ATTRIBUTES section.

For more details about this item type, see SPINEDIT item type on page 1287.

### Example

```xml
LAYOUT
GRID
{
    [f1  ]
    ...
}
END
END

ATTRIBUTES
SPINEDIT f1 = command.nbitems, STEP=5;
...
```

**TABLE item definition**

Defines attributes for a table layout tag, in a grid-based layout.

### Syntax

```xml
TABLE layout-tag: item-name ↓ , attribute-list ↓ ;
```

1. `layout-tag` is an identifier that defines the name of the layout tag.
2. `item-name` identifies the form item.
3. `attribute-list` defines the aspect and behavior of the form item.

### Form attributes

`AGGREGATETEXT, COMMENT, DOUBLECLICK, FONTPITCH, HEIGHT, HIDDEN, STYLE, TAG, UNHIDABLECOLUMNS, UNMOVABLECOLUMNS, UNSIZABLECOLUMNS, UNSORTABLECOLUMNS, WANTFIXEDPAGESIZE, WIDTH`.

### Style attributes

Common: `backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration`.

Class-specific: `allowWebSelection, forceDefaultSettings, headerAlignment, headerHidden, highlightColor, highlightCurrentCell, highlightCurrentRow, highlightTextColor, resizeFillsEmptySpace, reduceFilter, rightFrozenColumns, rowActionTrigger, showGrid, summaryLineAlwaysAtBottom, tableType`. 
Usage
Define a TABLE element in the ATTRIBUTES section, to configure a table layouted with a <TABLE > layout tag.

For more details about this item type, see TABLE item type on page 1288.

Example

```plaintext
LAYOUT
GRID
{
    <TABLE t1 >
    [ c1 | c2 | c3 ]
    [ c1 | c2 | c3 ]
    ...

    }
END
END

ATTRIBUTES
TABLE t1: table1, UNSORTABLECOLUMNS;
...
```

**TEXTEDIT item definition**
Defines a multi-line edit field, in a grid-based layout.

**Syntax**

```
TEXTEDIT item-tag = field-name [ , attribute-list ] ;
```

1. `item-tag` is an identifier that defines the name of the item tag in the layout section.
2. `field-name` identifies the name of the screen record field.
3. `attribute-list` defines the aspect and behavior of the form item.

**Form attributes**

COLOR, COLOR WHERE, COMMENT, DEFAULT, DOWNSHIFT, FONTPITCH, HIDDEN, INCLUDE, JUSTIFY, KEY, NOT NULL, NOENTRY, PLACEHOLDER, PROGRAM, REQUIRED, SAMPLE, SCROLL, SIZEPOLICY, SCROLLBARS, STYLE, STRETCH, TAG, TITLE, TABINDEX, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, UPSHIFT, VALIDATE LIKE, WANTTABS, WANTNORETURNS.

**Style attributes**

Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.

Class-specific: integratedSearch, showEditToolBox, spellCheck, textFormat, wrapPolicy.

Usage
Define the rendering and behavior of a text edit item tag, with a TEXTEDIT element in the ATTRIBUTES section.

For more details about this item type, see TEXTEDIT item type on page 1289.

Example

```plaintext
LAYOUT
GRID
{
    [f1 ]
    [ ]
```
TIMEEDIT item definition
Defines a line-edit with a clock widget to pick a time, in a grid-based layout.

Syntax

```
TIMEEDIT item-tag = field-name [ , attribute-list ] ;
```

1. `item-tag` is an identifier that defines the name of the item tag in the layout section.
2. `field-name` identifies the name of the screen record field.
3. `attribute-list` defines the aspect and behavior of the form item.

Form attributes

```
AUTONEXT, COLOR, COLOR WHERE, COMMENT, DEFAULT, FONTPITCH, HIDDEN, IMAGECOLUMN, INCLUDE,
JUSTIFY, NOT NULL, NOENTRY, PLACEHOLDER, REQUIRED, SAMPLE, STYLE, TABINDEX, TAG, TITLE,
UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, VALIDATE LIKE.
```

Style attributes

Common: `backgroundColor`, `border`, `fontFamily`, `fontSize`, `fontStyle`, `fontWeight`,
`localAccelerators`, `showAcceleratorInToolTip`, `textColor`, `textDecoration`.

Class-specific: none.

Usage

Define the rendering and behavior of a time edit `item tag`, with a `TIMEEDIT` element in the `ATTRIBUTES` section.

For more details about this item type, see `TIMEEDIT item type` on page 1290.

Example

```
LAYOUT
GRID
{[f1]
  [ ]
  ...
}
END
END

ATTRIBUTES
TIMEEDIT f1 = package.arrtime;
...```
**TREE item definition**
Defines attributes for a tree layout tag, in a grid-based layout.

**Syntax**
```
TREE layout-tag: item-name [, attribute-list ] ;
```
1. `layout-tag` is an identifier that defines the name of the layout tag.
2. `item-name` identifies the form item.
3. `attribute-list` defines the aspect and behavior of the form item.

**Form attributes**
```
COMMENT, DOUBLECLICK, HIDDEN, FONTPITCH, STYLE, TAG, UNHIDABLECOLUMNS,
UNMOVABLECOLUMNS, UNSORTABLECOLUMNS, WANTFIXEDPAGESIZE, WIDTH,
HEIGHT, PARENTIDCOLUMN, IDCOLUMN, EXPANDEDCOLUMN, ISNODECOLUMN, IMAGEEXPANDED,
IMAGECOLLAPSED, IMAGELEAF.
```

**Style attributes**
Common: `backgroundColor`, `border`, `fontFamily`, `fontSize`, `fontStyle`, `fontWeight`,
`localAccelerators`, `showAcceleratorInToolTip`, `textColor`, `textDecoration`.
Class-specific: `allowWebSelection`, `forceDefaultSettings`, `headerAlignment`, `headerHidden`,
`highlightColor`, `highlightCurrentRow`, `highlightTextColor`, `resizeFillsEmptySpace`,
`rightFrozenColumns`, `rowActionTrigger`, `showGrid`, `tableType`.

**Usage**
The TREE form item type can be used to specify the attributes of a tree container defined with a layout tag.

For more details about this item type, see TREE item type on page 1292.

**WEBCOMPONENT item definition**
Defines a generic form field that can receive an external widget, in a grid-based layout.

**Syntax**
```
WEBCOMPONENT item-tag = field-name [, attribute-list ] ;
```
1. `item-tag` is an identifier that defines the name of the item tag in the layout section.
2. `field-name` identifies the name of the screen record field.
3. `attribute-list` defines the aspect and behavior of the form item.

**Form attributes**
```
COLOR, COLOR WHERE, COMPONENTTYPE, COMMENT, DEFAULT, FONTPITCH, HEIGHT, HIDDEN, INCLUDE,
JUSTIFY, NOT NULL, NOENTRY, PROPERTIES, REQUIRED, SCROLLBARS, SIZEPOLICY, STYLE,
STRETCH, TAG, TABINDEX, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, VALIDATE LIKE,
WIDTH.
```

**Style attributes**
Common: `backgroundColor`, `border`, `fontFamily`, `fontSize`, `fontStyle`, `fontWeight`,
`localAccelerators`, `showAcceleratorInToolTip`, `textColor`, `textDecoration`.
Class-specific: none.
Usage
Define the rendering and behavior of a web component item tag, with a WEBCOMPONENT element in the ATTRIBUTES section.

For more details about this item type, see WEBCOMPONENT item type on page 1293.

Example

```
LAYOUT
GRID
{
  [f1
   ]
  [
   ]
  [
  ...
  [f2
   ]
  [
   ]
  [
  ...
  }
END
END

ATTRIBUTES

-- URL-based web component (recommended):
WEBCOMPONENT f1 = FORMONLY.mymap,
  STRETCH=BOTH;

-- qICAPI web component:
WEBCOMPONENT f2 = FORMONLY.mycal,
  COMPONENTTYPE="Calendar", -- lookup "Calendar.html",
  PROPERTIES = ( type = "gregorian",
                  week_start = 2,
                  days_off = ( 1, 7 )
               ),
  STRETCH=BOTH, STYLE="regular";
```

INSTRUCTIONS section
The INSTRUCTIONS section is used to define screen arrays, non-default screen records, and global form properties.

Syntax

```
INSTRUCTIONS
  [ screen-record-definition ]...[ ]
  [ DELIMITERS AB ]...[ ]
  [ DEFAULT SAMPLE = "string" ]
[END]
```

1. screen-record-definition is the definition of a screen record or screen array.
2. A and B define the opening and closing field delimiters for character based terminals.

Usage
The INSTRUCTIONS section must appear in the sequence described in form file structure.

The INSTRUCTIONS section is optional in a form definition.

The END keyword is optional.
This section is mainly used to define screen records, to group fields using tables, tree views, scrollgrids, or traditional static field arrays.

**Screen records (or screen arrays)**

A *screen record* is a named group of form fields.

See [Screen records / arrays](#) on page 1251 for more details.

**Field delimiters**

Use the DELIMITER keyword to specify the characters to be displayed as field delimiters on the screen.

This option is especially used for TUI mode applications.

**Default sample**

The `DEFAULT SAMPLE` directive defines the default sample text for all fields.

```
DEFAULT SAMPLE = "MMM"
```

See [SAMPLE attribute](#) on page 1389 for more details.

**Example**

```
SCHEMA stores
LAYOUT GRID
{ ...
} END
TABLES stock, items END
ATTRIBUTES ...
END
INSTRUCTIONS
SCREEN RECORD s_items[10]
( stock.*,
  items.quantity,
  FORMONLY.total_price )
DELIMITERS "[]"
END
```

**Related concepts**

- [Binding tables to arrays in dialogs](#) on page 1857
  Program arrays act as data model that are bound to form tables, when implementing list dialogs.

- [Stack-based layout](#) on page 1431
  A form file can define a stack-based layout within a tree of stack items.

- [Grid-based layout](#) on page 1418
  A form file can define a grid-based layout within a tree of layout items.

**KEYS section**

The `KEYS` section can be used to define default key labels for the current form.

**Syntax**

```
KEYS
```
1. *key-name* is the name of a key (like F10, Control-z).
2. *label* is the text to be displayed in the button corresponding to the key.

**Usage**

The **KEYS** section can be used to define default key labels at the form level.

The **KEYS** section must appear in the sequence described in form file structure.

The **KEYS** section is optional in a form definition.

The **END** keyword is optional.

**Note:** This feature is supported for backward compatibility. Consider using action attributes to define accelerator keys and decorate actions.

**Example**

```
KEYS
    F10 = "City list"
    F11 = "State list"
    F15 = "Validate"
END
```

**Related concepts**

Setting action key labels on page 1775
Labels can be defined to decorate buttons controlled by **ON KEY** / **COMMAND KEY** action handlers.

**Form item attributes**

The form item attributes reference.

**ACCELERATOR attribute**

The **ACCELERATOR** attribute is an action attribute defining the primary accelerator key for an action.

**Syntax**

```
ACCELERATOR = key
```

1. *key* defines the accelerator key.

**Usage**

This attribute is an action attribute that can be specified in form **ACTION DEFAULTS**, for more details, see **ACCELERATOR action attribute** on page 1750.

**Related concepts**

Defining keyboard accelerators for actions on page 1749

**ACCELERATOR2 attribute**

The **ACCELERATOR2** attribute is an action attribute defining the secondary accelerator key for an action.

**Syntax**

```
ACCELERATOR2 = key
```

1. *key* defines the accelerator key.
Usage
This attribute is an action attribute that can be specified in form ACTION DEFAULTS, for more details, see ACCELERATOR2 action attribute on page 1751.

Related concepts
Defining keyboard accelerators for actions on page 1749

ACCELERATOR3 attribute
The ACCELERATOR3 is an action attribute defining the third accelerator key for an action.

Syntax

ACCELERATOR3 = key

1. key defines the accelerator key.

Usage
This attribute is an action attribute that can be specified in form ACTION DEFAULTS, for more details, see ACCELERATOR3 action attribute on page 1752.

Related concepts
Defining keyboard accelerators for actions on page 1749

ACCELERATOR4 attribute
The ACCELERATOR4 is an action attribute defining the fourth accelerator key for an action.

Syntax

ACCELERATOR4 = key

1. key defines the accelerator key.

Usage
This attribute is an action attribute that can be specified in form ACTION DEFAULTS, for more details, see ACCELERATOR4 action attribute on page 1752.

Related concepts
Defining keyboard accelerators for actions on page 1749

ACTION attribute
The ACTION attribute defines the action associated with the form item.

Syntax

ACTION = action-name

1. action-name is an identifier that defines the name of the action to be sent.

Usage
The ACTION attribute defines the name of the action to be sent to the program when the user activates the form item. This attribute can for example be used in a BUTTONEDIT field to identify the corresponding action handle to be executed in the program when the button is pressed. The Dialog actions on page 1739 chapter describes how actions can be implemented in dialogs.
The action name can be prefixed with a sub-dialog identifier and/or field name, to define a qualified action view (see Binding action views to action handlers on page 1762 for more details).

Example

```
BUTTONEDIT f001 = customer.state, ACTION = print;
```

Related concepts

Configuring actions on page 1744
Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with action attributes.

**AGGREGATETEXT attribute**
The AGGREGATETEXT attribute defines a label to be displayed for aggregate fields.

**Syntax**

```
AGGREGATETEXT = [%]"string"
```

1. *string* defines the label to be associated with the aggregate cell, with the % prefix it is a localized string.

**Usage**
The AGGREGATETEXT attribute can be specified at the AGGREGATE field level, or globally at the TABLE level, to define a label for the whole summary line.

When defining the AGGREGATETEXT attribute at the aggregate field level, the text will be anchored to the value cell.

If the AGGREGATETEXT attribute is specified at the TABLE level, the label will appear on the left in the summary line.

When an aggregate text is defined at both levels, the global aggregate text of the table will be ignored.

For more details, see Summary lines in tables on page 1866.

Example

```
AGGREGATE tot = FORMONLY.total, AGGREGATETEXT="Total:"
```

**Related concepts**

Aggregate fields on page 1248
An AGGREGATE field defines a screen-record field to display summary information for a TABLE column.

Localized strings on page 538
Localized strings provide a means of writing applications in which the text of strings can be customized on site.

**AGGREGATETYPE attribute**
The AGGREGATETYPE attribute defines how the aggregate field value is computed.

**Syntax**

```
AGGREGATETYPE = { PROGRAM | SUM | AVG | MIN | MAX | COUNT }
```

**Usage**

PROGRAM specifies that the aggregate value will be computed and displayed by the program code.

An aggregate type different from PROGRAM specifies that the aggregate value is computed automatically:

- SUM computes the total of all values of the corresponding numeric column.
- AVG computes the average of all values of the corresponding numeric column.
• **MIN** displays the minimum value of the corresponding numeric column.
• **MAX** displays the maximum value of the corresponding numeric column.
• **COUNT** computes the number of rows.

The **SUM** and **AVG** aggregate types apply to data types that can be used as operand for an addition, such as **INTEGER**, **DECIMAL**, **INTERVAL**.

The **MIN** and **MAX** aggregate types apply to data types that can be compared, such as **INTEGER**, **DECIMAL**, **INTERVAL**, **CHAR**, **DATETIME**.

For more details, see Summary lines in tables on page 1866.

**Example**

```
AGGREGATE tot = FORMONLY.total, AGGREGATETYPE=PROGRAM;
```

**Related concepts**

**AGGREGATE item definition** on page 1332
Defines screen-record fields that hold computed values to be displayed as footer cells in a **TABLE** container.

**TABLE item definition** on page 1346
Defines attributes for a table layout tag, in a grid-based layout.

**AUTOSCALE attribute**
The AUTOSCALE attribute causes the form element contents to automatically scale to the size given to the item.

**Syntax**

```
AUTOSCALE
```

**Usage**

For images, this attribute forces the image to be stretched to fit in the area reserved for the image.

See Controlling the image layout on page 1148 for more details.

**Related concepts**

**IMAGE item type** on page 1279
Defines an area that can display an image resource.

**AUTONEXT attribute**
The AUTONEXT attribute forces the focus to automatically leave the current field when completed.

**Syntax**

```
AUTONEXT
```

**Usage**

When using the AUTONEXT attribute in a form item type that has a text editor, the focus goes automatically to the next field, when the user enters a character that completely fills the field. The maximum number of characters is defined by the **field input length**.

In form item types with value pickers such as calendars or drop down lists, the focus also goes automatically to the next field, when a value is selected in the value picker.

If the value entered in the field does not meet the requirements of other field attributes like **INCLUDE**, the focus stays in the current field, and an error message displays.
AUTONEXT is particularly useful with character fields in which the input data is of a standard length, such as postal codes. It is also useful if a character field has a length of 1, as only one keystroke is required to enter data and move to the next field.

The next field is defined by the field input order, see TABINDEX.

**Related concepts**

EDIT item type on page 1274
Defines a simple line-edit field.

**BUTTONTEXTTHIDDEN attribute**
The BUTTONTEXTTHIDDEN attribute indicates that the button labels for an element are not to be displayed.

**Syntax**

```plaintext
BUTTONTEXTTHIDDEN
```

**Usage**

Use BUTTONTEXTTHIDDEN in a TOOLBAR definition to hide the labels of toolbar buttons.

**Note:** On front-ends where the toolbar button texts can be hidden with an option (context menu), the stored settings take precedence over the BUTTONTEXTTHIDDEN attribute.

For more details about toolbar implementation and configuration, see Toolbars on page 1436.

**Related concepts**

TOOLBAR section on page 1298
The TOOLBAR section defines a toolbar with buttons that are bound to actions.

**CENTURY attribute**
The CENTURY attribute defines expansion of the year in a DATE or DATETIME field.

**Syntax**

```plaintext
CENTURY = "R" | "C" | "F" | "P"
```

**Usage**

The CENTURY attribute specifies how to expand abbreviated one- and two-digit year specifications in a DATE and DATETIME field.

Century expansion is based on this attribute and on the current year defined by the system clock.

The CENTURY attribute can specify any of four algorithms to expand abbreviated years into four-digit year values that end with the same digits (or digit) that the user has entered.

CENTURY supports the same settings as the DBCENTURY environment variable, but with a scope that is restricted to a single field.

If the CENTURY and DBCENTURY settings are different, CENTURY takes precedence.

Unlike DBCENTURY, the CENTURY attribute is not case sensitive. However, we recommend that you use uppercase letters in the attribute.
**CLASS attribute**
The **CLASS** attribute defines the behavior of a field defined with the **WIDGET** attribute.

**Syntax**

**Important:** This feature is deprecated, its use is discouraged although not prohibited.

```
CLASS = "identifier"
```

1. **identifier** is a predefined keyword defining the class of the field.

**Usage**
The **CLASS** attribute can only be used with the **WIDGET** attribute. It is ignored if **WIDGET** is not used.

**Note:** The **CLASS** attribute is deprecated, use new form item types instead.

**Table 352: Supported field classes**

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY</td>
<td>Field is used to trigger a keystroke instead of being a normal input field. Only supported with **WIDGET=&quot;BMP&quot;</td>
</tr>
<tr>
<td>PASSWORD</td>
<td>Field input is masked by replacing normal character echo with stars.</td>
</tr>
</tbody>
</table>

**Related concepts**

**WIDGET attribute** on page 1408
The **WIDGET** attribute specifies the type of graphical widget to be used for a field.

**COLOR attribute**
The **COLOR** attribute defines the foreground color of the text displayed by a form element.

**Syntax**

```
COLOR = color-spec
```

1. **color-spec** can be: BLACK, BLUE, CYAN, GREEN, MAGENTA, RED, WHITE, and YELLOW.
2. **color-spec** can also be one of: REVERSE, LEFT, BLINK, and UNDERLINE, and it can be combined with a color name (COLOR = RED REVERSE).

**Usage**
The **COLOR** attribute defines the logical color of a value displayed in a field.

The fglform compiler ignores COLOR=WHITE and COLOR=BLACK: These colors can be specified in the .per file, but will not be written in the .42f file.

**Tip:** In GUI mode, for more precise decoration options, use presentation styles.

A color name like RED, BLUE can be combined with an secondary keyword that must be one of: REVERSE, LEFT, BLINK, and UNDERLINE. The secondary keyword can also be used without a color name.

**Note:** For backward compatibility, the color can be specified as a number: 0=WHITE, 1=YELLOW, 2=MAGENTA, 3=RED, 4=CYAN, 5=GREEN, 6=BLUE, 7=BLACK. Note that fglform ignores COLOR=0 or COLOR=7: like COLOR=WHITE and COLOR=BLACK.
Example

```
EDIT f001 = customer.name, COLOR = RED;
```

Related concepts

COLOR WHERE Attribute on page 1358

The COLOR WHERE attribute defines a condition to set the foreground color dynamically.

COLOR WHERE Attribute

The COLOR WHERE attribute defines a condition to set the foreground color dynamically.

Syntax

```
COLOR = color-spec [...] WHERE bool-expr
```

1. `color-spec` can be: BLACK, BLUE, CYAN, GREEN, MAGENTA, RED, WHITE, and YELLOW.
2. `color-spec` can also be one of: REVERSE, LEFT, BLINK, and UNDERLINE, and it can be combined with a color name (COLOR = RED REVERSE).
3. `bool-expr` defines a boolean expression with a restricted syntax. This expression can only reference the current form field item tag.

Usage

The attribute COLOR WHERE defines a conditional color. The color will be applied if the condition is true.

The fglform compiler ignores COLOR=WHITE and COLOR=BLACK: These colors can be specified in the .per file, but will not be written in the .42f file.

A color name like RED, BLUE can be combined with an secondary keyword that must be one of: REVERSE, LEFT, BLINK, and UNDERLINE. The secondary keyword can also be used without a color name.

Note: For backward compatibility, the color can be specified as a number: 0=WHITE, 1=YELLOW, 2=MAGENTA, 3=RED, 4=CYAN, 5=GREEN, 6=BLUE, 7=BLACK. Note that fglform ignores COLOR=0 or COLOR=7: like COLOR=WHITE and COLOR=BLACK.

The condition in COLOR WHERE can only reference the field for which the attribute is set, using its item tag. See Boolean expressions in forms on page 1265 for more details.

Note: With form fields such as EDIT, BUTTONEDIT the color will be applied when leaving the field. The color will not change while editing the value.

Example

```
EDIT f001 = item.price, COLOR = RED WHERE f001 < 0;
```

Related concepts

COLOR attribute on page 1357

The COLOR attribute defines the foreground color of the text displayed by a form element.

CONFIG Attribute

The CONFIG attribute defines the behavior and decoration of a field defined with the WIDGET attribute.

Syntax

Important: This feature is deprecated, its use is discouraged although not prohibited.

```
CONFIG = "parameter [...]"
```

1. `parameter` is the value of a configuration parameter.
Usage

The CONFIG attribute can only be used with the WIDGET attribute. It is ignored if WIDGET is not used.

Configuration parameters are separated by blanks.

If a configuration parameter holds blank characters, you must use { } curly brackets to delimit the parameter value.

Note: The CONFIG attribute is deprecated, use new form item types instead.

Related concepts

WIDGET attribute on page 1408

The WIDGET attribute specifies the type of graphical widget to be used for a field.

CONTEXTMENU attribute

The CONTEXTMENU attribute defines whether a context menu option must be displayed for an action.

Syntax

| CONTEXTMENU = AUTO YES NO |

Usage

This attribute is an action attribute that can be specified in form ACTION DEFAULTS

CONTEXTMENU is an action attribute defining whether the context menu option must be displayed for an action.

Important: Actions to be displayed in a context menu must have a TEXT attribute. If the TEXT attribute is not defined or is empty, the action will not be shown in the context menu.

Possible values for CONTEXTMENU are:

1. NO indicates that no context menu option must be displayed for this action.
2. YES indicates that a context menu option must always be displayed for this action, if the action is visible.
3. AUTO means that the context menu option is displayed if no explicit action view is used for that action and the action is visible.

The default is YES.

Note: With some front-ends, the CONTEXTMENU=AUTO attribute will be interpreted as CONTEXTMENU=YES. Consider using CONTEXTMENU=NO or leave the default (CONTEXTMENU=YES).

This attribute applies to the actions defined by the current dialog in the current window. It can be specified as action default attribute in a global .4ad file, in the ACTION DEFAULTS section of form files, or as dialog action attribute.

For more details, see CONTEXTMENU action attribute on page 1754.

Related concepts

ui.Dialog.setActionHidden on page 2542

Showing or hiding a default action view.

Configuring actions on page 1744

Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with action attributes.

Action display in the context menu on page 1771

The CONTEXTMENU action default attribute allows you to control action visibility in the context menu.

COMMENT attribute

The COMMENT attribute defines a hint for the user about the form element.

Syntax

| COMMENT = "string" |
1. *string* is the text to display. With the % prefix, it is a localized string.

**Usage**

The most common use of the `COMMENT` attribute is to give information or instructions to the user.

The `COMMENT` attribute can be used for different types of form elements:

- with form field definitions, to show a message when the field gets the focus.
- with action views, to give a hint to the user about the action.

With form fields, this attribute is particularly appropriate when the field accepts only a limited set of values. The screen location where the message displays depends on external configuration. It can be displayed in the comment line, or in the status bar when using a graphical user interface. If the `OPEN WINDOW` statement specifies `COMMENT LINE OFF`, any output to the comment area is hidden even if the window displays a form that includes fields that include the `COMMENT` attribute.

The `COMMENT` attribute can, in specific situations, define a value placeholder message when the field is empty. If the `COMMENT` cannot be rendered in a dedicated area (as is the case with mobile devices), the `COMMENT` is used as the default `PLACEHOLDER`.

With actions, this attribute can be defined in the `ACTION DEFAULTS` section of a form or directly in an action view (BUTTON). See `COMMENT action attribute` on page 1753 for more details.

**Example**

```plaintext
-- In a form field definition
EDIT f1 = customer.name, COMMENT = "The customer name";

-- As action default
ACTION DEFAULTS
  ACTION print (COMMENT="Print current order information")
END

-- In a form button
BUTTON b1: print, COMMENT = "Print customer details";
```

**Related concepts**

Localized strings on page 538

*Localized strings* provide a means of writing applications in which the text of strings can be customized on site.

OPEN WINDOW on page 1131

Creates and displays a new window.

Configuring actions on page 1744

Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with *action attributes*.

**COMPLETER attribute**

The `COMPLETER` attribute enables autocompletion for the edit field.

**Syntax**

```plaintext
COMPLETER
```

**Usage:**

Form fields with `COMPLETER` attribute provide suggestions while the end-user types text into the field, it can be used in text edit fields such as `EDIT` and `BUTTONEDIT` item types.

Normally, the `ON CHANGE` trigger is fired for text edit fields when leaving the field and if the content was modified. Form fields defined with the `COMPLETER` attribute will trigger the `ON CHANGE` control block when the end user modifies the content of the field.
See Enabling autocompletion on page 1733 for more details.

**Example**

```plaintext
EDIT f1 = FORMONLY.custname, COMPLETER;
```

**Related concepts**

- `ui.Dialog.setCompleterItems` on page 2548
  Define autocompletion items for a field defined with COMPLETER attribute.

- `EDIT item type` on page 1274
  Defines a simple line-edit field.

- `BUTTONEDIT item type` on page 1267
  Defines a line-edit with a push-button that can trigger an action.

**COMPONENTTYPE attribute**

The COMPONENTTYPE attribute defines a name identifying the external widget for WEBCOMPONENT fields.

**Syntax**

```plaintext
COMPONENTTYPE = "name"
```

1. `name` defines the HTML file defining the web component.

**Usage**

The COMPONENTTYPE attribute is used to define the type of a WEBCOMPONENT form item for gICAPI web components.

When this attribute is specified, it defines the name of the HTML file that will be loaded by the front-end. If this attribute is not defined, the web component will be specified by an URL set dynamically by program in the field value. Consider using URL-based web components instead of gICAPI web components.

See Web components on page 1922 for more details about web component programming.

**Example**

```plaintext
WEBCOMPONENT f001 = FORMONLY.mycal, COMPONENTTYPE="Calendar";
```

**Related concepts**

- `WEBCOMPONENT item type` on page 1293
  Defines a specialized form item that holds an external component.

**DEFAULT attribute**

The DEFAULT attribute assigns a default value to a field during data entry.

**Syntax**

```plaintext
DEFAULT = value
```

1. `value` can be any literal expression supported by the form compiler, as long as it matches the form field type.
2. `value` can be `TODAY` to specify the current system date as default.
3. `value` can be `CURRENT` to specify the current system datetime as default.

**Usage**

The literal constant specified as default value must match the form field type. For example, when defining a numeric field, use a numeric decimal constant, for character string fields, use a double-quoted character literal.
The effect of the `DEFAULT` attribute depends on the `WITHOUT DEFAULTS` configuration option of the dialog using the form.

With the `INPUT` statement, form default values are ignored when using the `WITHOUT DEFAULTS` option. With this option, the runtime system displays the values in the program variables to the screen. Otherwise, the form default values will be displayed when the dialog starts.

With the `INPUT ARRAY` statement, the form default values are always used for new rows inserted by the user. With `INPUT ARRAY`, the `WITHOUT DEFAULTS` option indicates if the existing program array elements have to be used.

Defaults values can also be specified in the database schema file, for form fields defined with database column reference.

If the field is `FORMONLY`, you must also specify a data type when you assign the `DEFAULT` attribute to a field.

If both the `DEFAULT` attribute and the `REQUIRED` attribute are assigned to the same field, the `REQUIRED` attribute is ignored.

If you do not use the `WITHOUT NULL INPUT` option in the `DATABASE` section of a form, all fields default to null values unless you have specified a `DEFAULT` attribute.

Note that `DATETIME` and `INTERVAL` literals are not supported in the `DEFAULT` attribute.

Example

```
EDIT f001 = order.orderdate, DEFAULT = TODAY;
EDIT f012 = FORMONLY.discount TYPE DECIMAL(5,2), DEFAULT=0.10;
```

**Related concepts**

Form field initialization on page 1722
Form field initialization can be controlled by the `WITHOUT DEFAULTS` dialog option.

REQUIRED attribute on page 1387
The `REQUIRED` attribute forces the user to modify the content of a field during an input dialog.

EDIT item type on page 1274
Defines a simple line-edit field.

**DEFAULTVIEW attribute**
The `DEFAULTVIEW` attribute defines if a default view (a button) must be displayed for a given action.

**Syntax**

```
DEFAULTVIEW = ↓ AUTO ↓ YES ↓ NO ↓
```

**Usage**

This attribute is an action attribute that can be specified in form `ACTION DEFAULTS`.

`DEFAULTVIEW` is an action attribute defining whether the default action view (a button) must be displayed for an action.

Possible values for `DEFAULTVIEW` are:

- `NO` indicates that no default action view must be displayed for this action.
- `YES` indicates that a default action view must always be displayed for this action, if the action is visible.
- `AUTO` means that a default action view is displayed if no explicit action view is used for that action and the action is visible.

The default is `AUTO`.

For more details, see `DEFAULTVIEW action attribute` on page 1755.
The **DEFAULT** attribute assigns a default value to a field during data entry.

The **DISPLAY LIKE** attribute applies column attributes defined in the database schema files (.att) to a field.

### Syntax

```
DISPLAY LIKE [table.]column
```

1. `table` is the optional table name to qualify the column.
2. `column` is the name of the column to be used to retrieve display attributes.

### Usage

Specifying this attribute is equivalent to listing all the attributes that are assigned to `table.column` in the database schema file with the `.att` extension.

Display attributes are automatically taken from the schema file if the field is linked to `table.column` in the field name specification.

The **DISPLAY LIKE** attribute is evaluated at compile time, not at runtime. If the database schema file changes, recompile all forms using this attribute. Even if all of the fields in the form are FORMONLY, this attribute requires the form compiler to access the database schema file that contains the description of `table`.

### Example

```
EDIT f001 = FORMONLY.fullname, DISPLAY LIKE customer.custname;
```

### Related concepts

**Column Video Attributes File (.att)** on page 475
The `.att` database schema file contains the default video attributes of database table columns.

**FORMONLY fields** on page 1245
FORMONLY form fields define their data type explicitly, with or without referencing a database columns.

### DISCLOSUREINDICATOR attribute

The **DISCLOSUREINDICATOR** attribute adds a drill-down decoration to the form item.

### Syntax

```
DISCLOSUREINDICATOR
```

### Usage

The **DISCLOSUREINDICATOR** attribute is used on BUTTON form items to add a graphical hint, to indicate that a click on the button will drill down in the application windows, typically to show a detailed view of the information displayed in the current window.

This is a simple decoration attribute, with no other functional purpose.

For example, on iOS devices, the buttons defined with this attribute will show a typical > icon on the right.

### Example

```
BUTTON b_details : details,
   TEXT="Show details",
   DISCLOSUREINDICATOR;
```
Related concepts
Configuring actions on page 1744
Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with action attributes.

DOUBLECLICK attribute
The DOUBLECLICK attribute defines the action for row choice on TABLE/TREE/SCROLLGRID rows.

Syntax

\[
\text{DOUBLECLICK} = \text{action-name}
\]

1. action-name defines the name of the action to be invoked.

Usage
The DOUBLECLICK attribute can be used in a TABLE, TREE or SCROLLGRID container, to define the action to be sent when the user chooses a row. By default, on desktop and web front-ends, when the controller is a DISPLAY ARRAY, a row choice (mouse double-click) fires the "accept" action.

Important: The row-choice action can also be defined as DISPLAY ARRAY dialog attribute with the DOUBLECLICK option. This dialog attribute has been added to be used in conjunction with the DETAILACTION and ACCESSORYTYPE attributes for mobile apps. When the DOUBLECLICK attribute is specified at the dialog level, it takes precedence over the DOUBLECLICK attribute defined in the form specification file.

The action defined by DOUBLECLICK is by default triggered by a double-click on a front-end using a mouse device, and a finger tap on mobile front-ends. On desktop and web front-ends, the physical event that triggers the row choice action can be controlled by the rowActionTrigger style attribute.

Note: List views supporting the DOUBLECLICK attribute are TABLE, TREE and SCROLLGRID. The DOUBLECLICK attribute does not apply to simple field lists that are defined in the form layout without a parent list container.

For more details about row choice action configuration, see Defining the action for a row choice on page 1867.

Related concepts
TABLE item definition on page 1346
Defines attributes for a table layout tag, in a grid-based layout.

TREE item definition on page 1349
Defines attributes for a tree layout tag, in a grid-based layout.

SCROLLGRID item definition on page 1344
Defines a scrollgrid layout tag in a grid-based layout.

Record list (DISPLAY ARRAY) on page 1501
The DISPLAY ARRAY instruction provides record list navigation in an application form, with optional record modification actions.

DOWNSHIFT attribute
The DOWNSHIFT attribute forces character input to lowercase letters.

Syntax

\[
\text{DOWNSHIFT}
\]

Usage
When defining the DOWNSHIFT attribute for a character field, the characters typed by the user are automatically converted to lowercase.
Note: In text mode, the results of conversions between uppercase and lowercase letters are based on the locale settings.

Because using uppercase and lowercase letters results in different values in the database, storing character strings in one or the other format can simplify sorting and querying.

Important: Values send from the runtime system are displayed as is by the front-ends: No conversion will occur if you display "AbCdE" to a field using UPSHIFT or DOWNSHIFT attributes.

Example

```
EDIT f001 = FORMONLY.name, DOWNSHIFT;
```

Related concepts

UPSHIFT attribute on page 1402
The UPSHIFT attribute forces character input to uppercase letters.

EXPANDEDCOLUMN attribute
The EXPANDEDCOLUMN attribute specifies the form field that indicates whether a tree node is expanded.

Syntax

```
EXPANDEDCOLUMN = column-name
```

1. `column-name` is the name of the form field holding the flag indicating whether a tree node is expanded (opened).

Usage

This attribute is used in the definition of a TREE container.

You must specify form field column names, not item tag identifiers.

This attribute is optional.

For more details about treeview programming, see Tree views on page 1889.

Related concepts

Form fields on page 1242
`Form fields` are form elements designed for data input and/or data display.

Item tags on page 1259
`Item tags` define the position and size in a grid-based container.

FONTPITCH attribute
The FONTPITCH attribute defines the character font type as fixed or variable when the default font is used.

Syntax

```
FONTPITCH = \{FIXED\|VARIABLE\}
```

Usage

By default, most front-ends use variable width character fonts, but some fields might need to use a fixed font.

Tip: Use a STYLE defining a fixed font instead of this attribute.

Related concepts

STYLE attribute on page 1395
The STYLE attribute specifies a presentation style for a form element.

Defining a style on page 1167
Styles can be defined to be global (for all elements), for an element in general, or for specific types of an element.

**FORMAT attribute**
The FORMAT attribute defines the data formatting of numeric and date time fields, for input and display.

**Syntax**

```
FORMAT = "format"
```

1. *format* is a string of characters that specifies a data format.
2. For details about possible format strings, see Formatting data on page 318.

**Usage**
The FORMAT attribute can be set to define input and display format for numeric and date fields.

- For numeric fields such as *MONEY* and *DECIMAL*, the FORMAT attribute defines the formatting string with placeholders for the thousands, the decimal separator and the front or back currency symbols. The symbols for these elements can then be specified (localized) with the DBFORMAT or DBMONEY environment variables.
- For *DATE* fields, when the FORMAT attribute is not used, the default format is defined by the DBDATE environment variable.

**Tip:** Consider using the FORMAT attribute only is specific cases, and keep the default formatting for most of your date and numeric form fields.

The format specification is used when converting the input buffer to the program variable, and when displaying program variable data to form fields. For example, when defining a FORMAT="yyyy-mm-dd" for a form field bound to a program variable defined as a DATE, the user can input a date as 2013-12-24, and the date value will be displayed in the same manner.

**Note:** Do not mix PICTURE and FORMAT attributes in field definitions: The PICTURE attribute is used to define an input mask for character string fields, such as vehicle registration numbers, phone numbers.

If the format string is smaller than the field width, you get a compile-time warning, but the form is usable.

If necessary to satisfy the format specification, number values are rounded before they are displayed. Consider using the format specification that matches the precision and scale of the DECIMAL or MONEY variable bound to the field.

The format string can be any valid string expression using formatting characters as described in Formatting numeric values on page 318 and Formatting DATE values on page 321.

**Example**

```
EDIT f003 = order.totamount, FORMAT = "---,--&.&&@";
EDIT f005 = order.thedate, FORMAT = "mm/dd/yyyy";
```

**Related concepts**

- DATE on page 295
The DATE data type stores calendar dates with a Year/Month/Day representation.
- DECIMAL(p,s) on page 298
The DECIMAL data type is provided to handle large numeric values with exact decimal storage.
- MONEY(p,s) on page 303
The **MONEY** data type is provided to store currency amounts with exact decimal storage.

**GRIDCHILDRENINPARENT attribute**
The **GRIDCHILDRENINPARENT** attribute is used for a container to align its children to the parent container.

**Syntax**

```
GRIDCHILDRENINPARENT
```

**Usage**
By default, in a grid-based layout, child elements of a container are aligned locally inside the container layout cells. With the **GRIDCHILDRENINPARENT** attribute, you can force children to be aligned in a vertical or horizontal direction, based on the layout cells in the parent container of the container to which you assign this attribute.

**Important:** This feature is not supported on mobile platforms.

**Note:** The **GRIDCHILDRENINPARENT** attribute applies only to **GROUP** and **SCROLLGRID** containers used inside a parent **GRID** container.

When the group or scrollgrid containers are placed vertically over each other, the alignment applies on parent grid columns, and when the containers are placed side by side horizontally, the alignment applies on parent grid rows.

**Example**
With this form definition, the elements in the four group boxes will align vertically and horizontally to the parent grid cells:

```
LAYOUT
GRID
{
<G ga       ><G gb       >
  Some text
  [a       ]  b[b     ]
<     ><
<G gc       ><G gd       >
  [c       ]  d[d     ]
<     ><
}
END
END
ATTRIBUTES
GROUP ga: GRIDCHILDRENINPARENT;
GROUP gb: GRIDCHILDRENINPARENT;
GROUP gc: GRIDCHILDRENINPARENT;
GROUP gd: GRIDCHILDRENINPARENT;
EDIT a = FORMONLY.f_a;
EDIT b = FORMONLY.f_b;
EDIT c = FORMONLY.f_c;
EDIT d = FORMONLY.f_d;
END
```

**Related concepts**
- **Form rendering** on page 1415
  The section explains the layout rules to render forms on graphical front-ends.
- **Layout tags** on page 1254
Layout tags define layout areas for containers inside the frame of a grid-based container.

**HEIGHT attribute**
The **HEIGHT** attribute forces an explicit height for a form element.

**Syntax**

```
HEIGHT = integer |CHARACTERS|LINES|POINTS|PIXELS|
```

1. *integer* defines the height of the element.

**Usage**
By default, the height of an element is defined by the size of the form item tag in a grid-based layout, or by the type of the form item in a stack-based layout. Use the **HEIGHT** attribute to define a specific height for a form item.

**Note:** As a general rule, consider not specifying a unit, to default to relative characters/lines/columns, instead of specifying exact pixels or points. This is especially important for mobile devices, where the screen resolution can significantly vary depending on the smartphone or tablet model.

In a grid-based layout and stack-based layout, if you don't specify a size unit, it defaults to **CHARACTERS**, which defines a height based on the characters size in the current font.

**Grid-based layout**
For sizable items like **IMAGE**, the default height is defined by the number of lines of the form item tag in the layout, as a vertical character height. Overwrite this default by specifying the **HEIGHT** attribute.

For **TABLE/TREE** containers, the default height is defined by the number of lines used in the table layout. Overwrite the default by specifying the **HEIGHT = x LINES** attribute.

```
IMAGE img1: image1, WIDTH = 20, HEIGHT = 12;
```

**Stack-based layout**
For **TABLE** containers, the height of a list is defined by the actual number of rows, this cannot be changed.

For **IMAGE** items, by default the image is rendered full size, which means that the actual size of the image is used. Overwrite the default by specifying the **HEIGHT** attribute:

```
IMAGE image1, HEIGHT = 12, ...;
```

By default, **WEBCOMPONENT** items adapt their size to the content. To force a given size, use the **HEIGHT** attribute:

```
WEBCOMPONENT FORMONLY.chart, HEIGHT = 10, ...;
```

A **TEXTEDIT** item always adapts its size to the text value. By using the **HEIGHT** attribute, you can define a minimum height, when the value of the field is empty:

```
TEXTEDIT FORMONLY.comment, HEIGHT = 5, ...;
```

**Related concepts**
- **Controlling the image layout** on page 1148
  Explains how image form items can be sized in different front-end layout systems.
- **Item tags** on page 1259
  *Item tags* define the position and size in a grid-based container.
- **Form rendering basics** on page 1415
Get the essentials about form rendering.

**WIDTH attribute** on page 1411
The WIDTH attribute forces an explicit width of a form element.

**HIDDEN attribute**
The HIDDEN attribute indicates that the element is not to be displayed.

**Syntax**

```
HIDDEN \[ \] = USER \[ \]
```

1. HIDDEN sets the underlying item attribute to 1.
2. HIDDEN=USER sets the underlying item attribute to 2.

**Usage**

By default, all form elements are visible. Specify the HIDDEN attribute to hide a form element, such as a form field or a groupbox.

**Note:** When you set the form file HIDDEN attribute for a form field, the model node gets the hidden attribute, not the view node.

The runtime system detects hidden form fields: If you write an INPUT block using a hidden field, the field is ignored (as if it was declared as NOENTRY).

If the HIDDEN keyword is specified alone, the underlying item attribute is set to 1. The value 1 indicates that the element is definitively hidden to the end user, which cannot show the element, for example with the context menu of TABLE headers. In this hidden mode, the UNHIDABLE attribute is ignored by the front-end.

With HIDDEN=USER, the underlying item attribute is set to 2. The value 2 indicates that the element is hidden by default, but the end user can show/hide the element as needed, when the widget and its container provide a way to do so. For example, in a TABLE or TREE container, the end user can use a context menu option to show a column defined with HIDDEN=USER attribute. When HIDDEN is used without the USER value, the column is definitively invisible.

**Note:** Form fields hidden with HIDDEN=USER (value 2) might be shown anyway, if the field is needed by a dialog for input. In such case, the program dialog takes precedence over the HIDDEN attribute.

The visibility of form elements can be changed by program with the `ui.Form.setElementHidden()` or `ui.Form.setFieldHidden()` methods.

**Example**

```
EDIT f001 = FORMONLY.field1, HIDDEN;
EDIT col1 = FORMONLY.column1, HIDDEN=USER;
```

**Related concepts**

Form field deactivation on page 1728

**IDCOLUMN attribute**
The IDCOLUMN attribute specifies the form field that contains the identifier of a tree node.

**Syntax**

```
IDCOLUMN = column-name
```

1. column-name is a form field name.
Usage

This attribute is used in the definition of a TREE container, to define the name of the form field containing the identifier of a node in a tree view.

You must specify form field column names, not item tag identifiers.

This attribute is mandatory.

For more details about treeview programming, see Tree views on page 1889.

Related concepts

Form fields on page 1242
Form fields are form elements designed for data input and/or data display.

Item tags on page 1259
Item tags define the position and size in a grid-based container.

PARENTIDCOLUMN attribute on page 1383
The PARENTIDCOLUMN attribute specifies the form field that contains the identifier of the parent node of a tree node.

IMAGE attribute
The IMAGE attribute defines the image resource to be displayed for the form item.

Syntax

```plaintext
IMAGE = "resource"
```

1. `resource` defines the file name, path or URL to the image source.

Usage:

The IMAGE attribute is used to define the image resource to be displayed for form items such as a BUTTON, BUTTONEDIT, a TOOLBAR button, or a static IMAGE item.

For more details about image resource specification, see Providing the image resource on page 1149.

This attribute is also an action attribute that can be defined in the ACTION DEFAULTS section of a form or directly in an action view (BUTTON), see IMAGE action attribute on page 1756 for more details.

Example

```plaintext
-- As action default
ACTION DEFAULTS
  ACTION print (IMAGE="printer")
END

-- In a form buttonedit or button
BUTTONEDIT f001 = FORMONLY.field01, IMAGE = "zoom";
BUTTON b01: open_file, IMAGE = "buttons/fileopen";
BUTTON b02: accept, IMAGE = "http://myserver/images/accept.png";

-- In a static image form item
IMAGE: img1, IMAGE = "mylogo.png"
```

Related concepts

Using images on page 1147
Describes how to use pictures in the forms of your application.

IMAGE item type on page 1279
Defines an area that can display an image resource.

Configuring actions on page 1744
Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with *action attributes*.

**IMAGECOLUMN attribute**
The IMAGECOLUMN attribute defines the form field containing the image for the current field.

**Syntax**

```
IMAGECOLUMN = column-name
```

1. *column-name* is a form field name.

**Usage**
The IMAGECOLUMN attribute allows an image to be displayed on the left of the value of the column value. The image can be different for each row.

A typical usage is the TREE container: IMAGECOLUMN will allow a row-specific image to be displayed left of the tree node text. You defined only one image column for a tree node decoration.

When used in the definition of a TABLE column, the image and the column will be displayed in the same table cell. There can be several TABLE columns using an IMAGECOLUMN.

For TREE containers, the images defined by the IMAGECOLLAPSED, IMAGEEXPANDED and IMAGELEAF attributes take precedence over the images defined by the IMAGECOLUMN cell.

This attribute references form field that contains the name of an image. This form field must be defined as a PHANTOM form field, that will be part of the screen record definition in the INSTRUCTIONS section.

For more details about image resource specification in the PHANTOM column, see Providing the image resource on page 1149.

**Example**

```plaintext
... ATTRIBUTES
PHANTOM FORMONLY.icon;
EDIT FORMONLY.file_name, IMAGECOLUMN=icon;
...
END
INSTRUCTIONS
SCREEN RECORD sr(FORMONLY.icon, FORMONLY.file_name, ...);
...
```

**Related concepts**

- **Phantom fields** on page 1246
  A PHANTOM field defines a screen-record field which is not rendered in the layout (it acts as a hidden field).

- **Using images** on page 1147
  Describes how to use pictures in the forms of your application.

- **Tree views** on page 1889
  Describes how to implement tree views.

**IMAGECOLLAPSED attribute**
The IMAGECOLLAPSED attribute sets the global icon to be used when a tree node is collapsed.

**Syntax**

```
IMAGECOLLAPSED = "image-name"
```

1. *image-name* is an image resource.
Usage
This attribute is used in the definition of a TREE container, to define the icon to be used for nodes that are collapsed.
It overwrites the program array image defined by IMAGECOLUMN, if both are used.
This attribute is optional.
For more details about image resource specification, see Providing the image resource on page 1149.
For more details about treeview programming, see Tree views on page 1889.

Related concepts
Using images on page 1147
Describes how to use pictures in the forms of your application.

IMAGECOLUMN attribute on page 1371
The IMAGECOLUMN attribute defines the form field containing the image for the current field.

IMAGEEXPANDED attribute
The IMAGEEXPANDED attribute sets the global icon to be used when a tree node is expanded.

Syntax

```
IMAGEEXPANDED = "image-name"
```

1. *image-name* is an image resource.

Usage
This attribute is used in the definition of a TREE container, to define the icon to be used for nodes that are expanded.
It overwrites the program array image defined by IMAGECOLUMN, if both are used.
This attribute is optional.
For more details about image resource specification, see Providing the image resource on page 1149.
For more details about treeview programming, see Tree views on page 1889.

Related concepts
Using images on page 1147
Describes how to use pictures in the forms of your application.

IMAGECOLLAPSED attribute on page 1371
The IMAGECOLLAPSED attribute sets the global icon to be used when a tree node is collapsed.

IMAGELEAF attribute
The IMAGELEAF attribute defines the global icon for leaf nodes of a TREE container.

Syntax

```
IMAGELEAF = "image-name"
```

1. *image-name* is an image resource.

Usage
This attribute is used in the definition of a TREE container, to specify the name of the icon that must be used for leaf nodes.
It overwrites the program array image defined by IMAGECOLUMN, if both are used.
This attribute is optional.
For more details about image resource specification, see Providing the image resource on page 1149.
For more details about treeview programming, see Tree views on page 1889.

**Related concepts**

Using images on page 1147
Describes how to use pictures in the forms of your application.

### INCLUDE attribute

The INCLUDE attribute defines a list of possible values for a field.

#### Syntax

```plaintext
INCLUDE = ( ↓ NULL ↓ literal ↓ TO ↓ literal↓ ↓ ↓, ...) 
```

1. *literal* can be any literal expression supported by the form compiler.

#### Usage

The INCLUDE attribute specifies acceptable values for a field and causes the runtime system to check the data before accepting an input value.

If the field is FORMONLY, you must also specify a data type when you assign the INCLUDE attribute to a field.

Include the NULL keyword in the value list to specify that it is acceptable for the user to leave the field without entering any value.

Use the TO keyword to specify an inclusive range of acceptable values. When specifying a range of values, the lower value must appear first. The field value is accepted if it is greater or equal to the first literal, and lower or equal to the second literal.

**Example**

```plaintext
INCLUDE = (1 TO 999) 
is equivalent to:
(field_value >= 1 AND field_value <= 999 )
```

Special consideration must be taken for character string fields:

**Example**

```plaintext
INCLUDE = ("AAA" TO "ZZZ") 
is equivalent to:
(field_value >= "AAA" AND field_value <= "ZZZ")
ABC is accepted
A!! is not accepted
Zaa is not accepted
```

When combining several ranges and single values, the value entered by the user is verified for each element of the INCLUDE attribute:

```plaintext
INCLUDE = (1 TO 999, -1, NULL)  
is equivalent to:
(field_value >= 1 AND field_value <= 999 ) OR
(field_value == -1 ) OR
(field_value IS NULL )
```

**Example**

```plaintext
EDIT f001 = compute.rate, INCLUDE = ( 1 TO 100, 200, NULL); 
EDIT f002 = customer.state, INCLUDE = ( "AL" TO "GA", "IA" TO "WY" ); 
EDIT f003 = FORMONLY.valid TYPE CHAR, INCLUDE = ("Y","N");
```
**Related concepts**

- **Form-level validation rules** on page 1727
  Form-level validation rules can be defined for each field controlled by a dialog.
- **Form-only fields** on page 1245
  FORMONLY form fields define their data type explicitly, with or without referencing a database column.
- **ITEMS attribute** on page 1376
  The ITEMS attribute defines a list of possible values that can be used by the form item.

**INITIALIZER attribute**

The INITIALIZER attribute allows you to specify an initialization function that will be automatically called by the runtime system to set up the form item.

**Syntax**

```
INITIALIZER = [module.]function
```

1. `module` is the name of the module implementing the function. If no module is specified, the function must be defined in a module that is already loaded by the runtime system. If a module is specified, it will be automatically loaded on demand. Module name is case sensitive.
2. `function` is an identifier defining the program function to be called to initialize the combobox item list. Function name is case insensitive.

**Usage**

The INITIALIZER attribute defines the function to be called to fill the item list of the COMBOBOX field.

**Important:** The initialization function name is case insensitive.

When using a function qualified with a module prefix, the runtime system will automatically load the module if it is not yet loaded. In such case, the module specification in the INITIALIZER attribute is sufficient.

**Important:** The module prefix of the initialization function name is case sensitive (unlike the function name, which is case insensitive).

When specifying a un-qualified function name (without the module as prefix), the module defining the initialization function must have been loaded before the form is loaded.

**Tip:** To make sure that the module is loaded, define other functions in the module, to be invoked with a regular CALL instruction before displaying the form.

The initialization function must be defined with an `ui.ComboBox` parameter:

```
FUNCTION func-name(var-name ui.ComboBox) RETURNS ()
```

For more details about combobox initializer programming, see **Filling a COMBOBOX item list** on page 1735.

**Related concepts**

- **ui.ComboBox.setDefaultInitializer** on page 2562
  Define the default initializer for combobox form items.

**INITIALPAGESIZE attribute**

The INITIALPAGESIZE attribute defines the initial page size of a list element.

**Syntax**

```
INITIALPAGESIZE = lines
```

1. `lines` is an integer that defines the initial page size for the list element.
Usage

Some list containers such as resizable SCROLLGRID containers display by default a single row, if no other elements in the form layout forces the container size implicitly.

In such case, the INITIALPAGESIZE attribute can be used to specify a default initial number of rows to be displayed.

Related concepts
WANTFIXEDPAGESIZE attribute on page 1407
The WANTFIXEDPAGESIZE attribute controls the vertical resizing of a list element.

SCROLLGRID item type on page 1285
Defines a scrollable grid view widget.

INVISIBLE attribute
The INVISIBLE attribute prevents field data being readable on the screen.

Syntax

```
INVISIBLE
```

Usage

The INVISIBLE attribute can be used for EDIT and BUTTONEDIT fields, to obfuscate the value of the field. A typical usage of INVISIBLE is for password entry fields.

Depending on the front-end type, the characters are converted to blanks, stars, underscores or dots.

Characters that the user enters in a field with the INVISIBLE attribute are obfuscated during data entry. When displaying data by program to a field (with DISPLAY TO / BY NAME or when the interactive dialog synchronizes the form field content with the program variable), the INVISIBLE attribute will also make the field content unreadable.

Related concepts
HIDDEN attribute on page 1369
The HIDDEN attribute indicates that the element is not to be displayed.

KEYBOARDHINT attribute on page 1378
The KEYBOARDHINT attribute gives an indication of the kind of data the form field contains, allowing the front-end to adapt the keyboard accordingly.

ISNODECOLUMN attribute
The ISNODECOLUMN attribute specifies the form field that indicates whether a tree node has children.

Syntax

```
ISNODECOLUMN = column-name
```

1. `column-name` is a form field name.

Usage

This attribute is used in the definition of a TREE container, to specify the name of the form field indicating whether a tree node has children.

Even if the program node does not contain child nodes for this tree node, this attribute may be used, to implement dynamic filling of tree views.

You must specify form field column names, not item tag identifiers.
This attribute is optional.

For more details about treeview programming, see Tree views on page 1889.

**Related concepts**

Form fields on page 1242

*Form fields* are form elements designed for data input and/or data display.

Item tags on page 1259

*Item tags* define the position and size in a grid-based container.

**ITEMS attribute**

The **ITEMS** attribute defines a list of possible values that can be used by the form item.

**Syntax**

```plaintext
ITEMS = \{ single-value-list \} \{ double-value-list \}
```

where **single-value-list** is:

```plaintext
( value [, ...] )
```

where **double-value-list** is:

```plaintext
( ( value, label-value ) [, ...] )
```

1. **single-value-list** is a comma-separated list of single values.
2. **double-value-list** is a comma-separated list of (a, b) values pairs within parentheses.
3. **value** is a numeric or string literal, or one of the following keywords: NULL, TRUE, FALSE.
4. **label-value** is a numeric literal, a string literal, or a localized string.

**Usage**

The list must be delimited by parentheses, and each element of the list can be a simple literal value or a pair of literal values delimited by parentheses.

This attribute is not used by the runtime system to validate the field, you must use the **INCLUDE** attribute to force the possible values.

This example defines a list of simple values:

```plaintext
ITEMS = ("Paris", "London", "New York")
```

This example defines a list of pairs:

```plaintext
```

This attribute can be used, for example, to define the list of a COMBOBOX form item:

```plaintext
COMBOBOX cb01 = FORMONLY.combobox01,
```

In this example, the first value of a pair (1,2,3) defines the data values of the form field and the second value of a pair ("Paris", "London", "New York") defines the value to be displayed in the selection list.

When used in a RADIOGROUP form item, this attribute defines the list of radio buttons:

```plaintext
RADIOGROUP rg01 = FORMONLY.radiogroup01,
```
In this case, the first value of a pair (1,2,3) defines the data values of the form field and the second value of a pair ("Paris", "London", "New York") defines the value to be displayed as the radio button label.

You can specify item labels with localized strings, but this is only possible when you specify a key and a label:

```
ITEMS = ((1,%"item1"), (2,%"item2"), (3,%"item3"))
```

It is allowed to define a NULL value for an item (An empty string is equivalent to NULL):

```
ITEMS = ((NULL,"Enter bug status"), (1,"Open"), (2, "Resolved"))
```

In this case, the behavior of the field depends on the item type used.

**Related concepts**

*Localized strings* on page 538

*Localized strings* provide a means of writing applications in which the text of strings can be customized on site.

*COMBOBOX item type* on page 1270

Defines a line-edit with a drop-down list of values.

*RADIOGROUP item type* on page 1284

Defines a mutual exclusive set of options field.

**JUSTIFY attribute**

The JUSTIFY attribute defines the alignment of a text field content, and table column headers.

**Syntax**

```
JUSTIFY = \{ LEFT \| CENTER \| RIGHT \}
```

**Usage**

With the JUSTIFY attribute, you specify the justification of the content of a text field as LEFT, CENTER or RIGHT when the field is in display state.

**Note:** JUSTIFY has no effect on the content of form item types like IMAGE, CHECKBOX, PROGRESSBAR. It aligns only the content of text fields such as EDIT and BUTTONEDIT.

The JUSTIFY attribute can be used with all form item types. In addition to the text field data alignment, JUSTIFY defines the alignment of table column headers (this means table column header follows the alignment of field data). However, column header alignment in tables may not be enabled by default; check the headerAlignment presentation style attribute for the Table class.

**Important:** Depending on the front-end type, JUSTIFY may not be supported, if the native platform widget does not allow alignment. For example, a DATEEDIT form field may not align the date value. The JUSTIFY attribute is typically used for text data only.

If the field has the focus in a dialog allowing user input, the data alignment rules are front-end specific, and follow either JUSTIFY or the data type of the field variable. When the current dialog is a CONSTRUCT, criteria input is always left-aligned.

With mobile front-ends, tables are rendered as list views with a maximum of two visible columns. By default, the main and the comment columns are displayed vertically in each row (main is above the comment). Use JUSTIFY=RIGHT for the second column, in order to display columns side by side. Note that numeric fields are by default right justified and thus do not need this attribute to be set.

You can also specify the text alignment of static form labels with the JUSTIFY attribute.

**Example**

```
LABEL t01: TEXT="Hello!", JUSTIFY=RIGHT;
```
EDIT f01 = order.value, JUSTIFY=CENTER;

Related concepts

**FORMAT attribute** on page 1366
The FORMAT attribute defines the data formatting of numeric and date time fields, for input and display.

**Presentation styles** on page 1165
Use presentation styles to specify decoration attributes for window and form elements.

Related reference

**Table style attributes** on page 1204
Table presentation style attributes apply to a TABLE container.

**KEY attribute**
The KEY attribute is used to define the labels of keys when the field is made current.

**Syntax**

```plaintext
KEY keyname = "label"
```

1. `keyname` is the name of a key (like F10, "Control-z").
2. `label` is the text to be displayed in the button corresponding to the key.

**Usage**

Use the KEY attribute to define a label for the accelerator key corresponding to an action when the focus is in the field.

The `keyname` must be specified in quotes if you want to use Control / Shift / Alt key modifiers.

See the KEYS section to define key labels for the whole form.

**Note:** This feature is supported for backward compatibility. Consider using action attributes to define accelerator keys and decorate actions.

**Example**

```plaintext
EDIT f001 = customer.city, KEY F10 = "City list";
EDIT f002 = customer.state, KEY "Control-z" = "Open Zoom";
```

Related concepts

**Setting action key labels** on page 1775
Labels can be defined to decorate buttons controlled by ON KEY / COMMAND KEY action handlers.

**KEYS section** on page 1351
The KEYS section can be used to define default key labels for the current form.

**KEYBOARDHINT attribute**
The KEYBOARDHINT attribute gives an indication of the kind of data the form field contains, allowing the front-end to adapt the keyboard accordingly.

**Syntax**

```plaintext
KEYBOARDHINT = DEFAULT | EMAIL | NUMBER | PHONE | URL
```

**Usage**

The KEYBOARDHINT attribute can be used to give a hint to the front-end, regarding the kind of data the form field will contain. Based on this hint, the front-end will open the virtual keyboard adapted to the data type; especially useful when designing application forms for mobile platforms.
Valid values for \texttt{KEYBOARDHINT} are:

- \texttt{DEFAULT}: No hint, the only hint is the data type of the program variable bound to the form field.
- \texttt{EMAIL}: The field is used to enter an email address.
- \texttt{NUMBER}: The field is used to enter a numeric value.
- \texttt{PHONE}: The field is used to enter a phone number.
- \texttt{URL}: The field is used to enter an URL.

For example, when defining a numeric field with the attribute \texttt{KEYBOARDHINT=NUMBER}, the iOS device will display a numeric keyboard when entering data into that field.

\textbf{Figure 51: Mobile application using a numeric keyboard}
Example

```plaintext
EDIT f23 = customer.cust_phone, KEYBOARDHINT=PHONE;
```

Related concepts

**Keyboard type** on page 3579
Depending on the data being entered, a mobile device is expected to display the keyboard that is appropriate for the data.

**MINHEIGHT attribute**
The MINHEIGHT attribute defines the minimum height of a form.

**Syntax**

```
MINHEIGHT = integer
```

1. `integer` defines the minimum height of the element, as a number of grid cells.

**Usage**
The MINHEIGHT attribute is used to define a minimum height for the form/window. It must be specified in the attributes of the LAYOUT section.

The unit defaults to a number of grid cells. This is the equivalent of the `CHARACTERS` in the `HEIGHT` attribute specification.

For more details, see **Widget position and size in grid** on page 1423

**Example**

```plaintext
LAYOUT ( MINWIDTH=60, MINHEIGHT=50 )
GRID
...
```

Related concepts

**LAYOUT section** on page 1301
The LAYOUT section defines the graphical alignment of the form by using a tree of layout containers.

**MINWIDTH attribute** on page 1380
The MINWIDTH attribute defines the minimum width of a form.

**MINWIDTH attribute**
The MINWIDTH attribute defines the minimum width of a form.

**Syntax**

```
MINWIDTH = integer
```

1. `integer` defines the minimum width of the element, as a number of grid cells.

**Usage**
The MINWIDTH attribute is used to define a minimum width for the form/window. It must be specified in the attributes of the LAYOUT section.

The unit defaults to a number of grid cells. This is the equivalent of the `CHARACTERS` in the `WIDTH` attribute specification.

For more details, see **Widget position and size in grid** on page 1423
Example

LAYOUT ( MINWIDTH=60, MINHEIGHT=50 )
GRID
...

Related concepts

LAYOUT section on page 1301
The LAYOUT section defines the graphical alignment of the form by using a tree of layout containers.

MINHEIGHT attribute on page 1380
The MINHEIGHT attribute defines the minimum height of a form.

NOENTRY attribute
The NOENTRY attribute prevents data entry in the field during an input dialog.

Syntax

NOENTRY

Usage

Use the NOENTRY attribute to bypass field input during an INPUT or INPUT ARRAY statement.

A NOENTRY field is like a field permanently disabled with a DIALOG.setFieldActive("fieldname",FALSE) call, it cannot get the focus.

When compiling a form with a field referencing a SERIAL or BIGSERIAL column in the database schema, the NOENTRY attribute is automatically set. However, the attribute will not be set if the field is defined with a TYPE LIKE syntax.

When using a WITHOUT DEFAULTS dialog option, the content of the corresponding program variable is displayed in the field.

The NOENTRY attribute does not prevent data entry into a field during a CONSTRUCT statement.

Example

EDIT f001 = order.totamount, NOENTRY;

Related concepts

Form field initialization on page 1722
Form field initialization can be controlled by the WITHOUT DEFAULTS dialog option.

NOT NULL attribute
The NOT NULL attribute specifies that the field does not accept NULL values.

Syntax

NOT NULL

Usage

The NOT NULL attribute requires that the field contains a non-null value. It can be specified explicitly in the form field definition, or in the corresponding column definition in the database schema file. If no column is associated to the field, the NOT NULL attribute can also be used in the type definition of FORMONLY fields.

The NOT NULL attribute is effective only when the field name appears in the list of screen fields of an INPUT or INPUT ARRAY statement.
If a DEFAULT attribute is used for the field and the input dialog does not use the WITHOUT DEFAULTS option, the runtime system assumes that the default value satisfies the NOT NULL attribute.

Unlike the REQUIRED attribute which has no effect when the INPUT dialog uses the WITHOUT DEFAULTS option, the NOT NULL attribute is always checked when validating a dialog.

Example

```plaintext
EDIT f001 = customer.city, NOT NULL;
```

Related concepts

Form-level validation rules on page 1727
Form-level validation rules can be defined for each field controlled by a dialog.

Form field initialization on page 1722
Form field initialization can be controlled by the WITHOUT DEFAULTS dialog option.

DEFAULT attribute on page 1361
The DEFAULT attribute assigns a default value to a field during data entry.

REQUIRED attribute on page 1387
The REQUIRED attribute forces the user to modify the content of a field during an input dialog.

NOTEDITABLE attribute
The NOTEDITABLE attribute disables the text editor.

Syntax

```
NOTEDITABLE
```

Usage:

The NOTEDITABLE attribute can be used in BUTTONEDIT and TEXTEDIT fields to disable the text editor.

When the field is enabled/active, it can still get the focus.

When used in a BUTTONEDIT, the button of the field remains active, if there is a corresponding active action handler in the current dialog. Use this attribute if you do not want to allow text editing in BUTTONEDIT fields, when the value can only be set by an action.

When used in a TEXTEDIT field, the NOTEDITABLE attribute disallows user input. However, it is still possible to move in the text and select parts to do a copy/paste.

Related concepts

BUTTONEDIT item type on page 1267
Defines a line-edit with a push-button that can trigger an action.

TEXTEDIT item type on page 1289
Defines a multi-line edit field.

OPTIONS attribute
The OPTIONS attribute specifies widget options for the field.

Syntax

Important: This feature is deprecated, its use is discouraged although not prohibited.

```
OPTIONS = "option [...]
```

1. option can be −nolist (to indicate that the column should appear as an independent field).
Usage
The OPTIONS attribute specifies parameters for a form item defined with the WIDGET attribute.

Note: The OPTIONS attribute is deprecated, use new form item types instead.

Related concepts
WIDGET attribute on page 1408
The WIDGET attribute specifies the type of graphical widget to be used for a field.

ORIENTATION attribute
The ORIENTATION attribute defines whether an element displays vertically or horizontally.

Syntax

```
ORIENTATION = \_ VERTICAL \_ HORIZONTAL \_
```

Usage
The ORIENTATION attribute is typically used in the definition of a RADIOGROUP form item, to specify how radio items have to be displayed.

Example

```
RADIOGROUP f001 = customer.status, ORIENTATION=HORIZONTAL;
```

Related concepts
RADIOGROUP item type on page 1284
Defines a mutual exclusive set of options field.

PARENTIDCOLUMN attribute
The PARENTIDCOLUMN attribute specifies the form field that contains the identifier of the parent node of a tree node.

Syntax

```
PARENTIDCOLUMN = column-name
```

1. column-name is a form field name.

Usage
This attribute is used in the definition of a TREE container, to define the name of the form field containing the identifier of the tree node that is the parent of the current node in a tree view.

You must specify form field column names, not item tag identifiers.

This attribute is mandatory.

For more details about treeview programming, see Tree views on page 1889.

Related concepts
Form fields on page 1242
Form fields are form elements designed for data input and/or data display.

Item tags on page 1259
Item tags define the position and size in a grid-based container.

IDCOLUMN attribute on page 1369
The `IDCOLUMN` attribute specifies the form field that contains the identifier of a tree node.

**PICTURE attribute**
The `PICTURE` attribute specifies a character pattern for data entry in a text field, and prevents entry of values that conflict with the specified pattern.

**Syntax**

```
PICTURE = "format-string"
```

1. `format-string` defines the data input pattern of the field.

**Usage**

`format-string` can be any combination of characters, where the characters "A", "#" and "X" have a special meaning.

- The character "X" specifies any character at the given position.
- The character "#" specifies a digit character (1,2,3,...) at the given position.
- The character "A" specifies a letter character (a,b,C,D,...) at the given position.

Any character different from "A", "X" and "#" is treated as a literal. Such characters automatically appear in the field and do not have to be entered by the user.

The `PICTURE` attribute does not require data entry into the entire field. It only requires that whatever characters are entered conform to `format-string`.

**Tip:** Combine the `PICTURE` attribute with the `UPSHIFT` attribute, to allow only uppercase letters (A,B,C,...)

When `PICTURE` specifies input formats for `DATETIME` or `INTERVAL` fields, the form compiler does not check the syntax of `format-string`. Any error in `format-string` such as an incorrect field separator, produces a runtime error.

The typical usage for the `PICTURE` attribute is for (fixed-length) `CHAR` fields. It is not recommended to use `PICTURE` for other data types, especially numeric or date/time fields: The current value of the field must always match the `PICTURE` attribute.

Understand that the `PICTURE` attribute defines a mask for data entry. In order to format fields when data is displayed to the field, use the `FORMAT` attribute instead. `FORMAT` is typically used for formatted character string fields requiring input control.

The `PICTURE` attribute is ignored in `CONSTRUCT` and `DISPLAY / DISPLAY ARRAY` instructions. It only effects `INPUT` and `INPUT ARRAY` dialogs.

**Example**

```
EDIT f001 = carinfo.ident, PICTURE = "AA####-AA(X)";
```

**Related concepts**

- `DATETIME qual1 TO qual2` on page 296
  The `DATETIME` data type stores date and time data with time units from the year to fractions of a second.

- `INTERVAL qual1 TO qual2` on page 301
  The `INTERVAL` data type stores spans of time as Year/Month or Day/Hour/Minute/Second/Fraction units.

- `CHAR(size)` on page 293
  The `CHAR` data type is a fixed-length character string data type.

- `FORMAT attribute` on page 1366
The **FORMAT** attribute defines the data formatting of numeric and date time fields, for input and display.

**PLACEHOLDER attribute**
The **PLACEHOLDER** attribute defines a hint for the user when the field contains no value.

**Syntax**

```
PLACEHOLDER = [\%1"string"]
```

1. *string* defines the hint to be displayed when the field is empty, with the % prefix it is a localized string.

**Usage**
The **PLACEHOLDER** attribute can be used to show hint text in an input field, when a field contains no value.

The placeholder text is displayed when the field value is **NULL**, during an **INPUT**, **CONSTRUCT** or **INPUT ARRAY** dialog. In the case of **INPUT ARRAY**, placeholders are only displayed for the current row where the end user enters data.

This attribute is typically used for mobile and web applications, to display grayed text inside empty input fields.

**Note:** The **COMMENT** attribute can, in specific situations, define a default placeholder message.

**Example**

```
EDIT f001 = customer.cust_name, PLACEHOLDER = "<Enter customer name>";
```

**PROGRAM attribute**
The **PROGRAM** attribute can specify an external application program to edit **TEXT** or **BYTE** fields.

**Syntax**

```
PROGRAM = "editor"
```

1. *editor* is the name of the program that must be used to edit the special field data.

**Usage**

You can assign the **PROGRAM** attribute to a **TEXT** or **BYTE** field to call an external program to work with the **BYTE** or **TEXT** values.

This attribute works in TUI mode only.

Users can invoke the external program by pressing the exclamation point ( ! ) key while the screen cursor is in the field.

The external program then takes over control of the screen. When the user exits from the external program, the form is redisplayed with any display attributes besides **PROGRAM** in effect.

When no **PROGRAM** attribute is used, the **DBEDIT** environment variable defines the default editor.

**Related concepts**

- **TEXT** on page 307
  - The **TEXT** data type stores large text data.
- **BYTE** on page 290
  - The **BYTE** data type stores any type of binary data, such as images or sounds.
- **DBEDIT** on page 268
  - Defines the editor program for **TEXT** fields in TUI mode.
- **Text mode rendering (TUI mode)** on page 1109
PROPERTIES attribute
The PROPERTIES attribute is used to define a list of widget-specific characteristics.

Syntax

```
PROPERTIES = ( single-property | array-property | map-property | [],... )
```

where single-property is:

```
identifier = property-value
```

and array-property is:

```
identifier = ( property-value [,...] )
```

and map-property is:

```
identifier = ( single-property | map-property | [ ,... ] )
```

and property-value is:

```
[ boolean-value | numeric-value | %"string-value"
```

1. boolean-value is true or false.
2. numeric-value is an integer or decimal literal.
3. string-value is a string literal. % prefix can be used to define a localized string.

Usage

The PROPERTIES attribute is typically used to define the widget-specific attributes of a WEBCOMPONENT form item.

Property names and values are not checked, which allows you to freely set any characteristic of an external widget. Verify that the web component implementation supports the specified properties.

For more details about properties usage, see Web components on page 1922.

Example

```
WEBCOMPONENT f01 = FORMONLY.mycalendar,
    COMPONENTTYPE = calendar,
    PROPERTIES = ( type = "gregorian",
                    caption = %"calendar.caption",
                    week_start = 2,
                    days_off = ( 1, 7 ),
                    dates_off = ( "????-11-25", "????-06-20" ),
                    day_titles = ( t1 = "Sunday",
                                  t2 = "Monday",
                                  t3 = "Tuesday",
                                  t4 = "Wednesday",
                                  t5 = "Thursday",
                                  t6 = "Friday",
                                  t7 = "Saturday" )
                );
```

Related concepts
gICAPI.onProperty() on page 1940
The `gICAPI.onProperty()` function is executed when web component properties change.

**WEBCOMPONENT item type** on page 1293
Defines a specialized form item that holds an external component.

**QUERYEDITABLE attribute**
The `QUERYEDITABLE` attribute makes a `COMBOBOX` field editable during a `CONSTRUCT` statement.

### Syntax

```
QUERYEDITABLE
```

### Usage

The `QUERYEDITABLE` attribute is effective only during a `CONSTRUCT` statement. This attribute is useful when the display values match the real values in the `ITEMS` attribute, for example when `ITEMS= ("Paris", "London", "Madrid")`.

**Important:** It is not recommended to use the `QUERYEDITABLE` attribute, when the real field values are mapped to key/label values in the `ITEMS` attribute. For example with `ITEMS= (1,"Paris"),(2,"London"),(3,"Madrid")`.

During a `CONSTRUCT`, a `COMBOBOX` is not editable by default. The end-user is forced to set one of the values of the list as defined by the `ITEMS` attribute, or set the 'empty' item.

The `QUERYEDITABLE` attribute can be used to force the `COMBOBOX` to be editable during a `CONSTRUCT` instruction, in order to allow free search criterion input such as "A*".

If `QUERYEDITABLE` is used and the `ITEMS` are defined with key/label combinations, the text entered by the user will be automatically searched in the list of items. If a label corresponds, the key will be used in the SQL criterion, otherwise the entered text will be used as is. For example, if the items are defined as `ITEMS= (1,"Paris"),(2,"Madrid"),(3,"London")`, and the user enters "Paris" in the field, the item (1,"Paris") will match and will produce the SQL condition "colname = 1". If the user enters ">2", the text does not match any item, so it will be used as is and the SQL condition "colname > 2" is produced. Users may enter values like "Par*", but in this case the runtime system will raise an error, because this criterion is not valid for the numeric data type of the field. To avoid end-user confusion, it is not recommended to combine the `QUERYEDITABLE` attribute with a `COMBOBOX` defined with key/label combinations.

**Related concepts**

- **Query by example (CONSTRUCT) on page 1565**
The `CONSTRUCT` instruction implements database query criteria input in an application form.

- **COMBOBOX item type** on page 1270
Defines a line-edit with a drop-down list of values.

- **ITEMS attribute** on page 1376
The `ITEMS` attribute defines a list of possible values that can be used by the form item.

**REQUIRED attribute**
The `REQUIRED` attribute forces the user to modify the content of a field during an input dialog.

### Syntax

```
REQUIRED
```

### Usage

The `REQUIRED` attribute forces the user to modify the content of a field controlled by an input dialog (INPUT or INPUT ARRAY), when the input dialog does not use the `WITHOUT DEFAULTS` option. Within INPUT ARRAY, the `REQUIRED` attribute always applies to newly created rows.
If an **INPUT** dialog uses the **WITHOUT DEFAULTS** clause, the current value of the variable linked to the **REQUIRED** field is considered as a default value; the runtime system assumes that the field satisfies the **REQUIRED** attribute, even if the variable value is null.

In an **INPUT ARRAY** dialog, the **REQUIRED** attribute always applies to newly created rows, even if **WITHOUT DEFAULTS** is used. In other words, when creating a new row, **INPUT ARRAY** behaves like **INPUT** without the **WITHOUT DEFAULTS** clause.

If **REQUIRED** is effective regarding the **WITHOUT DEFAULTS** conditions, and a **DEFAULT** attribute is used for the field, the runtime system assigns the default value to the field and assumes that the **REQUIRED** attribute is satisfied.

The **REQUIRED** attribute does not prevent fields being null; If the field contains a value, and the user subsequently erases this value during the same input, the runtime system considers the **REQUIRED** attribute satisfied. To insist on a non-null entry, use the **NOT NULL** attribute.

**Example**

```
EDIT f001 = orders.ord_shipcmt, REQUIRED;
```

**Related concepts**

- **Form-level validation rules** on page 1727
  Form-level validation rules can be defined for each field controlled by a dialog.
- **DEFAULT attribute** on page 1361
  The **DEFAULT** attribute assigns a default value to a field during data entry.
- **ITEMS attribute** on page 1376
  The **ITEMS** attribute defines a list of possible values that can be used by the form item.
- **NOT NULL attribute** on page 1381
  The **NOT NULL** attribute specifies that the field does not accept NULL values.

**REVERSE attribute**

The **REVERSE** attribute displays any value in the field in reverse video (dark characters in a bright field).

**Syntax**

```
REVERSE
```

**Usage**

Use the **REVERSE** attribute to highlight specific fields in your forms.

On graphical front-ends, the **REVERSE** attribute is rendered by using the field **COLOR** attribute as background color. If the **COLOR** attribute is not defined, the reverse color defaults to gray.

**Note:** For the predefined (TTY) color names (like `blue`, `red`, `yellow`), the actual rendering color used by the front-ends is adapted to the reverse or normal mode: In normal mode, colors like "yellow" must be darker, to make the foreground text easily readable on an white background. In reverse mode (where the color specification applies to the background), the actual background color must be lighter, so that the black foreground text can be read more easily.

On character based terminals, the **REVERSE** video escape sequences must be defined in the **TERMININFO** or **TERMCAP** databases.

**Example**

```
EDIT f001 = customer.name, COLOR = BLUE, REVERSE;
```

**Related concepts**

- **COLOR attribute** on page 1357
The COLOR attribute defines the foreground color of the text displayed by a form element.

**TERMINFO** on page 265
Defines the terminfo terminal capabilities database.

**TERMCAP** on page 265
Defines the termcap terminal capabilities database on UNIX™ platforms.

**Text mode rendering (TUI mode)** on page 1109

**SAMPLE attribute**
The SAMPLE attribute defines the text to be used to compute the width of a form field widget.

**Syntax**

```
SAMPLE = "text"
```

1. *text* is the sample string that will be used to compute the width of the form field widget.

**Usage**

By default, form fields are rendered by the client with a size determined by the current font and the number of cells used by the form tag in the layout grid. The field width is computed so that the largest value can fit in the widget.

**Important:** This feature is not supported on mobile platforms.

Sometimes, the default computed width is too wide for the typical values displayed in the field. For example, numeric fields usually need less space than alphanumeric fields. If the values are always smaller, you can use the SAMPLE attribute to provide a hint for the front-end to compute the best width for that form field.

When specifying the SAMPLE attribute, you do not have to fill the sample string up to the width of the corresponding field tag: The front-ends will be able to compute a physical width by applying a ratio to fit the best visual result. For example, for a sample of *"xy"* used for a field defined with 10 characters, is equivalent to specifying a sample of *"xxyxxyxxy"*.

By default, when no SAMPLE attribute is used, the first 6 cells are always computed with the pixel width of the *"M"* character in the current font. Next cells are computed with the pixel width of the *"0"* (zero) character. In other words, the default sample model is *"MMMMMM000000....."*, reduced to the size of the field tag in the layout:

```
-123456789-123456789- Default SAMPLE
[f01 ] MMMM
[f02 ] MMMMM
[f03 ] MMMMMMM0000000000
```

A default sample can be specified for all fields used in the form, by using the DEFAULT SAMPLE option in the INSTRUCTIONS section:

```
INSTRUCTIONS
DEFAULT SAMPLE = "0"
END
```

See also **Input length of form fields** on page 1716.

**Example**

```
EDIT cid = customer.custid, SAMPLE="0000";
EDIT ccode = customer.ucode, SAMPLE="MMMMMM";
DATEEDIT be01 = customer.created, SAMPLE="00-00-0000";
```

**Related concepts**

**INSTRUCTIONS section** on page 1350
The INSTRUCTIONS section is used to define screen arrays, non-default screen records, and global form properties.

**SCROLL attribute**
The SCROLL attribute can be used to enable horizontal scrolling in a character field.

**Syntax**

```plaintext
SCROLL
```

**Usage**

By default, the maximum data input length is defined by the width of the item-tag of the field. For example, if you define a CHAR field in the form with a length of 3 characters, users can only enter a maximum of 3 characters, even if the program variable used for input is a CHAR(20).

If you want to let the user input more characters than the width of the item-tag of the field, use the SCROLL attribute.

The SCROLL attribute applies only to fields with character data input.

Use the SCROLL attribute only when the layout of the form does not allow for defining an item tag that is large enough to hold all possible character string data that fits in the corresponding program variable. Understand that the end user can miss a part of the displayed data when the field is too small. Therefore, times when you would need to use the SCROLL attribute are rare.

For more details about field input length, see Input length of form fields on page 1716.

**Related concepts**

- **Item tags** on page 1259
  - *Item tags* define the position and size in a grid-based container.

**SCROLLBARS attribute**
The SCROLLBARS attribute can be used to specify scrollbars for a form item.

**Syntax**

```plaintext
SCROLLBARS = | NONE | VERTICAL | HORIZONTAL | BOTH |
```

**Usage**

This attribute defines scrollbars for a form item like TEXTEDIT and WEBCOMPONENT.

By default, when not specifying the SCROLLBARS attribute, TEXTEDIT fields get a vertical scrollbar.

By default, a WEBCOMPONENT form item gets a vertical scrollbar.

**Example**

```plaintext
TEXTEDIT f001 = customer.fname, SCROLLBARS=BOTH;
```

**Related concepts**

- **STRETCH attribute** on page 1394
  - The STRETCH attribute specifies how a widget must resize when the parent container is resized.
- **SIZEPOLICY attribute** on page 1391
The **SIZEPOLICY** attribute is a sizing directive based on the content of a form item.

**SIZEPOLICY attribute**
The **SIZEPOLICY** attribute is a sizing directive based on the content of a form item.

**Syntax**

```plaintext
SIZEPOLICY = \ INITIAL \ FIXED \ DYNAMIC \n```

**Usage**
The **SIZEPOLICY** attribute defines how front-ends compute the widget size (width and height), or the widget width only of form items, based on the content, or definition of the form field or form item.

**Note:** For most form items types like **LABEL**, **CHECKBOX**, or **COMBOBOX**, the **SIZEPOLICY** attribute applies only to the width of the widget. For form items such as **IMAGE**, the attribute applies to the width and height. However, for example, the height of a (vertical) **RADIOGROUP** is defined by the number of **ITEMS**, and only the width can be controlled with the **SIZEPOLICY** attribute.

The **SIZEPOLICY** attribute applies to AUI tree leaf elements of a form (it does not apply to containers such as **TABLE** or **GRID**): **SIZEPOLICY** can be used with form items having content with variable size such as **IMAGE**, **LABEL** (for localization) and with form items where the definition impacts the widget size, like the width of labels in **ITEMS** of a **COMBOBOX**.

Elements allowing user input such as **EDIT**, or elements where the size does not depend on the value of content such as **PROGRESSBAR**, **SLIDER** do not use the **SIZEPOLICY** attribute.

**Note:** In **TABLE** or **TREE** containers, **SIZEPOLICY** applies only to **COMBOBOX**, **CHECKBOX** and **RADIOGROUP** (this concerns the size policy specified by the user, or the default (**INITIAL**)). In such list views, for widgets like **LABEL**, the size policy is implicitly fixed, and the column width is defined by the form layout.

When the **SIZEPOLICY** is not specified, it defaults to **INITIAL**. The behavior then depends on the type of form item. For more details, see Table 353: Behavior of **SIZEPOLICY=INITIAL**, based on form item type on page 1392.

**SIZEPOLICY = FIXED**

When **SIZEPOLICY** is **FIXED**, the form element's size (or its width only) is defined in the layout of the form specification file.

The size of the element is computed from the width and height in the form grid and the font used on the front-end side.

The element keeps the size, even if the content is modified. However, for form items supporting the **STRETCH** attribute, if the **STRETCH** attribute is set to **X**, **Y** or **BOTH**, the form element can still stretch when the parent window size changes.

**Note:** With GMA, when the last element on the right of a grid row is defined with **SIZEPOLICY=FIXED** and its right hand border does not pass over the screen border, GMA will extend its width to reach the border. The last element with a fixed size policy may only grow. It does not shrink to a smaller width than the width defined in the form.

**SIZEPOLICY = DYNAMIC**

When **SIZEPOLICY** is **DYNAMIC**, the form element's size (or its width only) of the element grows and shrinks to adapt to the size of the content (like the width of a **LABEL** text), or to adapt to the widget definition (like the picture size for an **IMAGE**), during the lifetime of the application.

This can be used for **COMBOBOX** or **RADIOGROUP** fields, when the width of the widget must fit exactly to its content, which can vary during the program execution.

**Note:** Exceptions regarding **SIZEPOLICY=DYNAMIC**:
1. With SIZEPOLICY=DYNAMIC, some element such as BUTTON, LABEL, IMAGE and RADIOGROUP can shrink and grow all the time, while COMBOBOX elements can only grow.
2. With WEBCOMPONENT fields, SIZEPOLICY=DYNAMIC is ignored.
3. With TEXTEDIT fields, SIZEPOLICY=DYNAMIC is only supported by GMA and GMI mobile front-ends.

**SIZEPOLICY = INITIAL**

When SIZEPOLICY is INITIAL, the first time the element appears on the screen, its size (or its width only) is computed from the initial content (like the text of a LABEL) or the widget definition (like the labels in ITEMS of COMBOBOX). Once the widget displays, its size is frozen.

However, for form items supporting the STRETCH attribute, if the STRETCH attribute is set to X, Y or BOTH, the form element can still stretch when the parent window size changes.

**Note:** SIZEPOLICY=INITIAL is ignored for WEBCOMPONENT fields.

SIZEPOLICY=INITIAL is typically used when the size of the element must be fixed, but is not known at design time. For example, when populating a COMBOBOX item list from a database table, the size of the COMBOBOX may need to be increased, based on the size of items labels in the drop-down list. This size policy mode is also useful when the text of labels is unknown at design time because of internationalization.

With SIZEPOLICY=INITIAL, the behavior differs depending on the form element type.

**Table 353: Behavior of SIZEPOLICY=INITIAL, based on form item type**

<table>
<thead>
<tr>
<th>Form item</th>
<th>Behavior with SIZEPOLICY=INITIAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUTTON</td>
<td>The size defined in the form is a minimum size. If the initial button text is bigger, the size grows (width and height).</td>
</tr>
<tr>
<td>CHECKBOX</td>
<td>The width defined in the form is a minimum width. If the checkbox label (TEXT) is larger as the minimum width, this defines the initial size of the widget. Then, the checkbox size remains fixed for the life time of the form.</td>
</tr>
<tr>
<td>COMBOBOX</td>
<td>The width defined in the form is a minimum width. If a label of combobox ITEMS is larger than the minimum width, the size grows in order for the combobox to fully display the largest item. Then, the combobox size remains fixed for the life time of the form.</td>
</tr>
<tr>
<td>LABEL</td>
<td>The width defined in the form is a minimum width. If the length of the initial field value (or the length of the TEXT attribute for static labels) is larger as the minimum width, this defines the initial size of the widget. Then, the label keeps the same size for the life time of the form. The size defined in the form is ignored.</td>
</tr>
<tr>
<td>IMAGE</td>
<td>Image form items adapt their size to the initial image displayed and keep that size. If no initial image is displayed (i.e. the image field value is NULL), the form item does not take space in the layout (and also does not adapt the size if an image is displayed later on). Images can use the STRETCH attribute, so that the widget size is dependent on the parent container, overriding the SIZEPOLICY attribute. If the WIDTH and HEIGHT attributes must be used, the SIZEPOLICY attribute must be set to FIXED.</td>
</tr>
<tr>
<td>RADIOGROUP</td>
<td>For vertical radiogroups: The height of the widget is defined by the number of ITEMS. The width defined in the form is a minimum width. If a label of radiogroup ITEMS is larger than the minimum width, the width grows in order for the radiogroup to fully display the largest item. Then, the radiogroup size remains fixed for the life time of the form.</td>
</tr>
<tr>
<td>TEXTEDIT</td>
<td>N/A: SIZEPOLICY=INITIAL is not supported for TEXTEDIT fields.</td>
</tr>
<tr>
<td>WEBCOMPONENT</td>
<td>N/A: SIZEPOLICY=INITIAL is ignored for WEBCOMPONENT fields.</td>
</tr>
</tbody>
</table>
Example

```java
IMAGE img1 = FORMONLY.main_image,
    SIZEPOLICY=DYNAMIC;

COMBOBOX f001 = customer.city,
    SIZEPOLICY=INITIAL; -- default

WEBCOMPONENT wc1 = FORMONLY.chart,
    COMPONENTTYPE="chart",
    SIZEPOLICY=FIXED,
    STRETCH=BOTH;
```

Related concepts
- **Controlling the image layout** on page 1148
  Explains how image form items can be sized in different front-end layout systems.
- **Web components** on page 1922
  This section describes how to use web components in your application.
- **STRETCH attribute** on page 1394
  The STRETCH attribute specifies how a widget must resize when the parent container is resized.

**SPACING attribute**
The SPACING attribute is a spacing directive to display form elements.

**Syntax**

```
SPACING = \[ \text{NORMAL} \\| \text{COMPACT} \]
```

**Usage**

This attribute defines the global distance between two neighboring form elements. In \text{NORMAL} mode, the front-end displays form elements consistent with the desktop spacing, which is, for example, 6 and 10 pixels on Microsoft\textsuperscript{TM} Windows\textsuperscript{®} platforms.

When using the \text{COMPACT} mode, large forms that by default do not fit to the screen can be displayed with less space between elements.

By default, forms are displayed with \text{COMPACT} spacing.

**Example**

```
LAYOUT ( SPACING=COMPACT )
```

Related concepts
- **LAYOUT section** on page 1301
  The LAYOUT section defines the graphical alignment of the form by using a tree of layout containers.

**SPLITTER attribute**
The SPLITTER attribute forces the container to use a splitter widget between each child element.

**Syntax**

```
SPLITTER
```
Usage
This attribute indicates that the container (typically, a VBOX or HBOX) must have a splitter between each child element held by the container. If a container is defined with a splitter and if the children are stretchable (like TABLE or TEXTEDIT), users can resize the child elements inside the container.

Example
VBOX ( SPLITTER )

Related concepts
HBOX item type on page 1278
Defines a layout area to render child elements in horizontal direction.

VBOX item type on page 1293
Defines a layout area to render child elements in vertical direction.

STEP attribute
The STEP attribute specifies how a value is increased or decreased in one step (by a mouse click or key up/down).

Syntax
STEP = integer

1. integer defines a positive integer value to be added (for an increase) or subtracted (for a decrease).

Usage
This attribute is typically used with form items allowing the user to change the current integer value by a mouse click like SLIDER, SPINEDIT.

Example
SLIDER s01 = FORMONLY.slider, STEP=10;

Related concepts
SLIDER item type on page 1286
Defines a slider form item.

SPINEDIT item type on page 1287
Defines a spin box widget to enter integer values.

STRETCH attribute
The STRETCH attribute specifies how a widget must resize when the parent container is resized.

Syntax
STRETCH = NONE ↓ X ↓ Y ↓ BOTH ↓

Usage
This attribute is typically used with form items that can be re-sized like IMAGE, TEXTEDIT, or WEBCOMPONENT fields.

By default, such form items have a fixed width and height, but in some cases you may want to force the widget to resize vertically, horizontally, or in both directions.

• To allow the widget to resize vertically only, use STRETCH=Y.
• To allow the widget to resize horizontally only, use STRETCH=X.
To allow the widget to resize vertically and horizontally, use \texttt{STRETCH=BOTH}.
To deny the widget to resize in both directions, use \texttt{STRETCH=NONE}.

\textbf{Note:} By default, \texttt{WEBCOMPONENT} form items stretch in both directions (\texttt{STRETCH=BOTH} is the default).

\textbf{Example}

\begin{verbatim}
IMAGE i01 = FORMONLY.image01, STRETCH=BOTH;
\end{verbatim}

\textbf{Related concepts}

- \texttt{Controlling the image layout} on page 1148
  Explains how image form items can be sized in different front-end layout systems.
- \texttt{Packed and unpacked grids} on page 1420
  When resizing a window, the content will either grow with the window or be packed in the top left position.

\textbf{STYLE attribute}

The \texttt{STYLE} attribute specifies a presentation style for a form element.

\textbf{Syntax}

\begin{verbatim}
STYLE = "string"
\end{verbatim}

1. \textit{string} is a user-defined style.

\textbf{Usage}

This attribute specifies a presentation style to be applied to a form element.

The presentation style can define decoration attributes such as a background color, a font type, and so on.

\textbf{Note:} The string used to define this attribute must be a \textit{style-name} only, it must not contain the \textit{element-type} that is typically used to define the style in a .4st file (as \texttt{CheckBox.important} for example)

For more details about using the \textit{STYLE} attribute, see \textit{Using presentation styles} on page 1167.

\textbf{Example}

\begin{verbatim}
EDIT c01 = item.comment, STYLE = "important";
\end{verbatim}

\textbf{Related concepts}

- \texttt{COLOR attribute} on page 1357
  The \texttt{COLOR} attribute defines the foreground color of the text displayed by a form element.
- \texttt{COLOR WHERE Attribute} on page 1358
  The \texttt{COLOR WHERE} attribute defines a condition to set the foreground color dynamically.

\textbf{TABINDEX attribute}

The \texttt{TABINDEX} attribute defines the tab order for a form item.

\textbf{Syntax}

\begin{verbatim}
TABINDEX = \texttt{integer}
\end{verbatim}

1. \texttt{integer} defines the order of the item in the tab sequence.

\textbf{Usage}

This attribute can be used to define the order in which the form items are selected as the user "tabs" from field to field.
To take TABINDEX attributes into account in dialogs, the program must defined the form tabbing order with the OPTIONS FIELD ORDER FORM instruction. Alternatively, a dialog can use the FIELD ORDER FORM option as well.

**Important:** TABINDEX values must be unique in a form.

The TABINDEX attribute can also be used to define which field must get the focus when a FOLDER page is selected.

By default, form items get a tab index depending on the order in which they appear in the LAYOUT section.

**Tip:** TABINDEX can be set to zero in order to exclude the form item from the tabbing list. The item can still get the focus with the mouse.

For more details, see [Defining field tabbing order method](#) on page 561

**Example**

```
EDIT f001 = customer.fname, TABINDEX = 1;
EDIT f002 = customer.lname, TABINDEX = 2;
EDIT f003 = customer.comment,
            TABINDEX = 0; -- Excluded from tabbing list
```

**Related concepts**

- **LAYOUT section** on page 1301
  The LAYOUT section defines the graphical alignment of the form by using a tree of layout containers.
- **FOLDER container** on page 1307
  Defines the parent container for folder pages, in a grid-based layout.
- **Defining the tabbing order** on page 1729
  Control the order of tabbing through the fields with the TABINDEX attribute.

**TAG attribute**

The TAG attribute can be used to identify the form item with a specific string.

**Syntax**

```
TAG = "tag-string"
```

1. *tag-string* is user-defined string.

**Usage**

This attribute is used to identify form items with a specific string. It can be queried in the program to perform specific processing.

You are free to use this attribute as you need. For example, you can define a numeric identifier for each field in the form in order to show context help, or group fields for specific input verification.

If you need to handle multiple data, you can format the text, for example, by using a pipe separator, or even the JSON notation.

**Example**

```
EDIT f001 = customer.fname, TAG = "name";
EDIT f002 = customer.lname, TAG = "name|optional";
```

**Related concepts**

- `ui.ComboBoxsetDefaultInitializer` on page 2562
  Define the default initializer for combobox form items.
- `ui.FormsetDefaultInitializer` on page 2499
Define the default initializer for all forms.

**TEXT attribute**
The TEXT attribute defines the label associated with a form item.

**Syntax**

```
TEXT = [ ]"string"
```

1. *string* defines the label to be associated with the form item, with the % prefix it is a localized string.

**Usage**
The TEXT attribute is used to define the label of a form field, for example for a CHECKBOX form field or a BUTTON action view.

Consider using localized strings with the %"string-id" syntax, if you plan to internationalize your application.

This attribute is also an action attribute that can be defined in the ACTION DEFAULTS section of a form or directly in an action view (BUTTON), see TEXT action attribute on page 1758 for more details.

**Example**

```
-- As form action default
ACTION DEFAULTS
  ACTION print (TEXT="Print")
END

-- As a CHECKBOX label
CHECKBOX cb01 = FORMONLY.checkbox01,
  TEXT="OK" ...
;

-- As a BUTTON label
BUTTON b1: print, TEXT="Print";
```

**Related concepts**

- Localized strings on page 538
- Localized strings provide a means of writing applications in which the text of strings can be customized on site.
- Configuring actions on page 1744
  Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with action attributes.
- Form items on page 1238
  The concept of form item includes all elements used in the definition of a form.

**TITLE attribute**
The TITLE attribute defines the title of a form item.

**Syntax**

```
TITLE = [ ]"string"
```

1. *string* defines the title to be associated with the form item, with the % prefix it is a localized string.

**Usage**
The TITLE attribute is typically used to define the title of a form field that will be defined as a TABLE or TREE column, or form items used in a stacked layout, to define the label associated to the item.

**Note:** It is recommended to restrict your use of the TITLE attribute to form fields that make up the columns of a table/tree container, or form items used in a stacked layout.
Consider using localized strings with the `%"string-id"` syntax, if you plan to internationalize your application.

**Example**

```
EDIT col4 = FORMONLY.ord_shipdate, TITLE="Ship date";
```

**Related concepts**

- **Localized strings** on page 538  
  *Localized strings* provide a means of writing applications in which the text of strings can be customized on site.

- **TABLE container** on page 1326  
  Defines a re-sizable table designed to display a list of records.

- **Form items** on page 1238  
  The concept of *form item* includes all elements used in the definition of a form.

**UNHIDABLE attribute**

The **UNHIDABLE** attribute indicates that the element cannot be hidden or shown by the user with the context menu.

**Syntax**

```
UNHIDABLE
```

**Usage**

By default, a **TABLE** or **TREE** container allows the user to hide the columns by a right-click on the column header. Use the **UNHIDABLE** attribute to prevent the user from hiding a specific column.

**Note:** The end user is also not allowed to show columns that are hidden by default with **HIDDEN=USER**.

Makes sense only for a field that is used for the definition of a column in a **TABLE** or **TREE** container.

**Example**

```
EDIT col1 = item.comment, UNHIDABLE;
```

**Related concepts**

- **TABLE container** on page 1326  
  Defines a re-sizable table designed to display a list of records.

- **TREE container** on page 1327  
  The **TREE** container defines the presentation of a list of ordered records in a tree-view widget.

- **HIDDEN attribute** on page 1369  
  The **HIDDEN** attribute indicates that the element is not to be displayed.

- **UNHIDABLECOLUMNS attribute** on page 1398  
  The **UNHIDABLECOLUMNS** attribute indicates that the columns of the table cannot be hidden or shown by the user with the context menu.

**UNHIDABLECOLUMNS attribute**

The **UNHIDABLECOLUMNS** attribute indicates that the columns of the table cannot be hidden or shown by the user with the context menu.

**Syntax**

```
UNHIDABLECOLUMNS
```
Usage

When using the UNHIDABLECOLUMNS attribute in a TABLE or TREE definition, the end user will not be allowed to hide columns.

Note: The end user is also not allowed to show columns that are hidden by default with HIDDEN=USER.

Example

```plaintext
table t1 ( unhidablecolumns )
```

Related concepts

**HIDDEN attribute** on page 1369
The HIDDEN attribute indicates that the element is not to be displayed.

**TABLE container** on page 1326
Defines a re-sizable table designed to display a list of records.

**TREE container** on page 1327
The TREE container defines the presentation of a list of ordered records in a tree-view widget.

**UNHIDABLE attribute** on page 1398
The UNHIDABLE attribute indicates that the element cannot be hidden or shown by the user with the context menu.

**UNMOVABLE attribute**
The UNMOVABLE attribute prevents the user from moving a defined column of a table.

Syntax

```plaintext
unmovable
```

Usage

By default, a TABLE or TREE container allows the user to move the columns by dragging and dropping the column header.

Use the UNMOVABLE attribute to prevent the user from changing the order of a specific column.

Makes sense only for a field that is used for the definition of a column in a TABLE or TREE container.

Typically, UNMOVABLE is used on at least two columns, to prevent the user from changing the order of the input on these columns.

Example

```plaintext
edit c01 = item.comment, unmovable;
```

Related concepts

**TABLE container** on page 1326
Defines a re-sizable table designed to display a list of records.

**TREE container** on page 1327
The TREE container defines the presentation of a list of ordered records in a tree-view widget.

**UNMOVABLECOLUMNS attribute** on page 1400
The UNMOVABLECOLUMNS attribute prevents the user from moving columns of a table.

**UNMOVABLECOLUMNS attribute**
The UNMOVABLECOLUMNS attribute prevents the user from moving columns of a table.

**Syntax**

```
UNMOVABLECOLUMNS
```

**Usage**
When using this attribute in a TABLE or TREE definition, the end user will not be allowed to move columns around.

**Example**

```
TABLE t1 ( UNMOVABLECOLUMNS )
```

**Related concepts**
- **TABLE container** on page 1326
  Defines a re-sizable table designed to display a list of records.
- **TREE container** on page 1327
  The TREE container defines the presentation of a list of ordered records in a tree-view widget.
- **UNMOVABLE attribute** on page 1399
  The UNMOVABLE attribute prevents the user from moving a defined column of a table.

**UNSIZABLE attribute**
The UNSIZABLE attribute indicates that the element cannot be resized by the user.

**Syntax**

```
UNSIZABLE
```

**Usage**
By default, a TABLE or TREE container allows the user to resize the columns by a drag-click on the column header. Use the UNSIZABLE attribute to prevent a resize on a specific column.

Makes sense only for a field that is used for the definition of a column in a TABLE or TREE container.

**Example**

```
EDIT c01 = item.comment, UNSIZABLE;
```

**Related concepts**
- **TABLE container** on page 1326
  Defines a re-sizable table designed to display a list of records.
- **TREE container** on page 1327
  The TREE container defines the presentation of a list of ordered records in a tree-view widget.
- **UNSIZABLECOLUMNS attribute** on page 1401
The **UNSIZABLECOLUMNS** attribute indicates that the columns of the table cannot be resized by the user.

### UNSIZABLECOLUMNS attribute

The **UNSIZABLECOLUMNS** attribute indicates that the columns of the table cannot be resized by the user.

**Syntax**

```
UNSIZABLECOLUMNS
```

**Usage**

When using this attribute in a **TABLE** or **TREE** definition, the end user will not be allowed to resize the columns.

**Example**

```
TABLE t1 ( UNSIZABLECOLUMNS )
```

**Related concepts**

- **TABLE container** on page 1326
  - Defines a re-sizable table designed to display a list of records.
- **TREE container** on page 1327
  - The **TREE** container defines the presentation of a list of ordered records in a tree-view widget.
- **UNSIZABLE attribute** on page 1400
  - The **UNSIZABLE** attribute indicates that the element cannot be resized by the user.

### UNSORTABLE attribute

The **UNSORTABLE** attribute indicates that the element cannot be selected by the user for sorting.

**Syntax**

```
UNSORTABLE
```

**Usage**

By default, a **TABLE** or **TREE** container allows the user to sort the columns by a left-click on the column header.

Use the **UNSORTABLE** attribute to prevent a sort on a specific column.

Makes sense only for a field that is used for the definition of a column in a **TABLE** or **TREE** container.

**Example**

```
EDIT c01 = item.comment, UNSORTABLE;
```

**Related concepts**

- **TABLE container** on page 1326
  - Defines a re-sizable table designed to display a list of records.
- **TREE container** on page 1327
  - The **TREE** container defines the presentation of a list of ordered records in a tree-view widget.
- **UNSIZABLECOLUMNS attribute** on page 1402
The **UNSORTABLECOLUMNS** attribute indicates that the columns of the table cannot be selected by the user for sorting.

**UNSORTABLECOLUMNS** attribute
The **UNSORTABLECOLUMNS** attribute indicates that the columns of the table cannot be selected by the user for sorting.

**Syntax**

```
UNSORTABLECOLUMNS
```

**Usage**

When using this attribute in a **TABLE** or **TREE** definition, the end user will not be allowed to sort rows.

**Example**

```
TABLE t1 ( UNSORTABLECOLUMNS )
```

**Related concepts**

**TABLE container** on page 1326
Defines a re-sizable table designed to display a list of records.

**TREE container** on page 1327
The **TREE** container defines the presentation of a list of ordered records in a tree-view widget.

**UNSORTABLE attribute** on page 1401
The **UNSORTABLE** attribute indicates that the element cannot be selected by the user for sorting.

**UPSHIFT attribute**
The **UPSHIFT** attribute forces character input to uppercase letters.

**Syntax**

```
UPSHIFT
```

**Usage**

When defining the **UPSHIFT** attribute for a character field, the characters typed by the user are automatically converted to uppercase.

**Note:** In text mode, the results of conversions between uppercase and lowercase letters are based on the **locale settings**.

Because using uppercase and lowercase letters results in different values in the database, storing character strings in one or the other format can simplify sorting and querying.

**Important:** Values send from the runtime system are displayed as is by the front-ends: No conversion will occur if you display "AbCdE" to a field using **UPSHIFT** or **DOWNSHIFT** attributes.

**Example**

```
EDIT f001 = FORMONLY.name, UPSHIFT;
```

**Related concepts**

**DOWNSHIFT attribute** on page 1364
The **DOWNSHIFT** attribute forces character input to lowercase letters.

**VALIDATE attribute**

The **VALIDATE** action attribute defines the data validation level for a given action.

**Syntax**

```
VALIDATE = NO
```

**Usage**

This attribute is an action attribute that can be specified in form **ACTION DEFAULTS**.

When the **VALIDATE** action attribute is set to **NO**, it indicates that no data validation must occur for this action. However, current input buffer contains the text modified by the user before triggering the action.

For more details, see **VALIDATE action attribute** on page 1759.

**VALIDATE LIKE attribute**

The **VALIDATE LIKE** attribute applies column attributes defined in the .val database schema files to a field.

**Syntax**

```
VALIDATE LIKE `[table.]`column
```

**Note:**

1. *table* is the optional table name to qualify the column.
2. *column* is the name of the column used to search for validation rules.

**Usage**

Specifying the **VALIDATE LIKE** attribute is equivalent to writing in the field definition all the attributes that are assigned to *table.column* in the .val database schema file.

**Note:** The .val attributes are taken automatically from the schema file if the field is linked to *table.column* in the field name specification. The **VALIDATE LIKE** attribute is usually specified for FORMONLY fields.

The **VALIDATE LIKE** attribute is evaluated at compile time, not at runtime. If the database schema file changes, recompile all your forms.

Even if all of the fields in the form are FORMONLY, the **VALIDATE LIKE** attribute requires the form compiler to access the database schema file that contains the description of *table*.

**Example**

```
EDIT f001 = FORMONLY.fullname, VALIDATE LIKE customer.custname;
```

**Related concepts**

- **Formonly fields** on page 1245
- FORMONLY form fields define their data type explicitly, with or without referencing a database columns.
- **VALIDATE** on page 403
The VALIDATE instructions checks a variable value based on database schema validation rules.

**VALUEMIN attribute**
The VALUEMIN attribute defines a lower limit of values displayed in widgets (such as progress bars).

**Syntax**

```
VALUEMIN = integer
```

1. `integer` is an integer literal.

**Usage**

This attribute is typically used to define the lower limit in PROGRESSBAR, SPINEDIT and SLIDER fields.

This attribute is not used by the runtime system to validate the field. You must use the INCLUDE attribute to control value boundaries.

For more details, see for example the SLIDER item type.

**Example**

```
SLIDER s01 = FORMONLY.slider01,
    VALUEMIN=0,
    VALUEMAX=500;
```

**Related concepts**

- **PROGRESSBAR item type** on page 1283
  Defines a progress indicator field.
- **SPINEDIT item type** on page 1287
  Defines a spin box widget to enter integer values.
- **SLIDER item type** on page 1286
  Defines a slider form item.
- **VALUEMAX attribute** on page 1404
  The VALUEMAX attribute defines a upper limit of values displayed in widgets (such as progress bars).

**VALUEMAX attribute**
The VALUEMAX attribute defines a upper limit of values displayed in widgets (such as progress bars).

**Syntax**

```
VALUEMAX = integer
```

1. `integer` is an integer literal.

**Usage**

This attribute is typically used to define the upper limit in PROGRESSBAR, SPINEDIT and SLIDER fields.

This attribute is not used by the runtime system to validate the field. You must use the INCLUDE attribute to control value boundaries.

For more details, see for example the SLIDER item type.

**Example**

```
SLIDER s01 = FORMONLY.slider01,
    VALUEMIN=0,
```
Related concepts

PROGRESSBAR item type on page 1283
Defines a progress indicator field.

SPINEDIT item type on page 1287
Defines a spin box widget to enter integer values.

SLIDER item type on page 1286
Defines a slider form item.

VALUEMIN attribute on page 1404
The VALUEMIN attribute defines a lower limit of values displayed in widgets (such as progress bars).

VALUECHECKED attribute
The VALUECHECKED attribute defines the value associated with a checkbox item when it is checked.

Syntax

```
VALUECHECKED = value
```

1. value is a numeric or string literal, or one of the following keywords: NULL, TRUE, FALSE.

Usage

This attribute is used together with the VALUEUNCHECKED attribute to define the values corresponding to the states of a CHECKBOX.

This attribute is not used by the runtime system to validate the field, you must use the INCLUDE attribute to control value boundaries.

For more details, see for example the CHECKBOX item type.

Example

```
CHECKBOX cb01 = FORMONLY.checkbox01,
    TEXT="OK",
    VALUECHECKED=TRUE,
    VALUEUNCHECKED=FALSE;
```

Related concepts

CHECKBOX item type on page 1269
Defines a boolean or three-state checkbox field.

VALUEUNCHECKED attribute on page 1405
The VALUEUNCHECKED attribute defines the value associated with a checkbox item when it is not checked.

VALUEUNCHECKED attribute
The VALUEUNCHECKED attribute defines the value associated with a checkbox item when it is not checked.

Syntax

```
VALUEUNCHECKED = value
```

1. value is a numeric or string literal, or one of the following keywords: NULL, TRUE, FALSE.

Usage

This attribute is used together with the VALUECHECKED attribute to define the values corresponding to the states of a CHECKBOX.
This attribute is not used by the runtime system to validate the field, you must use the INCLUDE attribute to control value boundaries.

For more details, see for example the CHECKBOX item type.

**Example**

```plaintext
CHECKBOX cb01 = FORMONLY.checkbox01,
    TEXT="OK",
    VALUECHECKED="Y",
    VALUEUNCHECKED="N";
```

**Related concepts**

- **CHECKBOX item type** on page 1269
  - Defines a boolean or three-state checkbox field.
- **VALUECHECKED attribute** on page 1405
  - The VALUECHECKED attribute defines the value associated with a checkbox item when it is checked.
- **VERIFY attribute**
  - The VERIFY attribute requires users to enter data in the field twice to reduce the probability of erroneous data entry.

**Syntax**

```plaintext
VERIFY
```

**Usage**

This attribute supplies an additional step in data entry to ensure the integrity of your data. After the user enters a value into a VERIFY field and presses the Return or Tab key, the runtime system erases the field and requests reentry of the value. The user must enter exactly the same data each time, character for character: 15000 is not exactly the same as 15000.00.

The VERIFY attribute takes effect in INPUT or INPUT ARRAY instructions only, it has no effect on CONSTRUCT statements.

**Related concepts**

- **REQUIRED attribute** on page 1387
  - The REQUIRED attribute forces the user to modify the content of a field during an input dialog.
- **NOT NULL attribute** on page 1381
  - The NOT NULL attribute specifies that the field does not accept NULL values.
- **VERSION attribute**
  - The VERSION attribute is used to specify a user version string for an element.

**Syntax**

```plaintext
VERSION = { "string" | TIMESTAMP }
```

1. `string` is a user-defined version string.

**Usage**

This attribute specifies a version string to distinguish different versions of a form element. Specify an explicit version string or use the TIMESTAMP keyword to make the form compiler write a timestamp string into the 42f file.

Typical usage is to specify a version of the form to indicate if the form content has changed.

**Important:** It is recommended that you use the TIMESTAMP clause only during development.
Example

```
LAYOUT ( TEXT="Orders", VERSION = "1.23" )
```

Related concepts

**LAYOUT section** on page 1301
The **LAYOUT** section defines the graphical alignment of the form by using a tree of layout containers.

**WANTFIXEDPAGESIZE attribute**
The **WANTFIXEDPAGESIZE** attribute controls the vertical resizing of a list element.

**Syntax**

```
WANTFIXEDPAGESIZE \[ = NO \]
```

**Usage**
The **WANTFIXEDPAGESIZE** attribute can be used for **TABLE**, **TREE** and **SCROLLGRID** containers to control the vertical resizing of the list element.

- By default, a **TABLE** container is resizable (vertically and horizontally). To freeze the height of the table to the number of lines defined by the form file, use the attribute **WANTFIXEDPAGESIZE**.
- By default, a **SCROLLGRID** container is not resizable in height. The number of visible scrollgrid rows is defined by the form file. To allow the scrollgrid to stretch vertically, use the attribute **WANTFIXEDPAGESIZE=NO**.

**Note**: When using a stretchable **SCROLLGRID** with **WANTFIXEDPAGESIZE=NO**, one can define the rendering with the **customWidget** style attribute.

**Related concepts**

**INITIALPAGESIZE attribute** on page 1374
The **INITIALPAGESIZE** attribute defines the initial page size of a list element.

**TABLE container** on page 1326
Defines a re-sized table designed to display a list of records.

**SCROLLGRID container** on page 1325
Defines a scrollable grid view widget, in a grid-based layout.

**WANTNORETURNS attribute**
The **WANTNORETURNS** attribute forces a text field to reject newline characters when the user presses the Return key.

**Syntax**

```
WANTNORETURNS
```

**Usage**

By default, text fields like **TEXTEDIT** insert a newline (ASCII 10) character in the text when the user presses the Return key. As the Return key is typically used to fire the **accept** action to validate the dialog, you can force the field to reject Return keys with this attribute.

The user can still enter new line characters with Shift-Return or Ctrl-Return, if these keys are not bound to actions.

For more details, see the **TEXTEDIT** item type.

**Related concepts**

**WANT TABS attribute** on page 1408
The **WANTTABS** attribute forces a text field to insert Tab characters in the text when the user presses the Tab key.

### WANTTABS attribute
The **WANTTABS** attribute forces a text field to insert Tab characters in the text when the user presses the Tab key.

#### Syntax

```plaintext
WANTTABS
```

#### Usage

By default, text fields like **TEXTEDIT** do not insert a Tab character in the text when the user presses the Tab key, since the Tab key is used to move to the next field. You can force the field to consume Tab keys with this attribute.

The user can still jump out of the field with Shift-Tab, if this key is not bound to an action.

For more details, see the **TEXTEDIT** item type.

### Related concepts

**WANTNORETURNS** attribute on page 1407

The **WANTNORETURNS** attribute forces a text field to reject newline characters when the user presses the Return key.

### WIDGET attribute

The **WIDGET** attribute specifies the type of graphical widget to be used for a field.

#### Syntax

**Important:** This feature is deprecated, its use is discouraged although not prohibited.

```plaintext
WIDGET = "identifier"
```

1. *identifier* defines the type of widget, it can be one of the keywords listed in Table 354: Supported widgets on page 1409.

#### Usage

The **WIDGET** attribute defines the type of widget to be used for a form field.

This attribute is used with **CONFIG**, **CLASS**, and **INCLUDE** attributes, to define parameters for the field widget.

**Note:** The **WIDGET** attribute is deprecated, use new form item types instead.

- Instead of **WIDGET**="BMP", use a **IMAGE** form item.
- Instead of **WIDGET**="CANVAS", use a **CANVAS** form item.
- Instead of **WIDGET**="CHECK", use a **CHECKBOX** form item.
- Instead of **WIDGET**="COMBO", use a **COMBOBOX** form item.
- Instead of **WIDGET**="BUTTON", use a **BUTTON** form item.
- Instead of **WIDGET**="FIELD_BMP", use a **BUTTONEDIT** form item.
- Instead of **WIDGET**="RADIO", use a **RADIOGROUP** form item.

The *identifier* widget type is case sensitive, only uppercase letters are recognized.

When you use the **WIDGET** attribute, the form cannot be properly displayed on character based terminals, it can only be displayed on graphical front-ends.
### Table 354: Supported widgets

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Effect</th>
<th>Other attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Canvas</strong></td>
<td>The field is used as a drawing area. Field must be declared as FORMONLY field.</td>
<td>None.</td>
</tr>
</tbody>
</table>
| **BUTTON** | The field is presented as a button widget with a label. | **CONFIG**: The unique parameter defines the key to be sent when the user clicks on the button. Button text is defined in configuration files or from the program with a DISPLAY TO instruction. For example:  
**CONFIG = "Control-z"** |
| **BMP** | The field is presented as a button with an image. | **CONFIG**: First parameter defines the image file to be displayed, second parameter defines the key to be sent when the user clicks on the button. For example:  
**CONFIG = "smiley.bmp F11"**  
**Important**: Image files are not centralized on the machine where the program is executed; image files must be present on the Workstation. See front-end specific documentation for more details. |
| **CHECK** | The field is presented as a checkbox widget. It can be used with the **CLASS** attribute to change the behavior of the widget. | **CONFIG**: First and second parameters define the values corresponding respectively to the state "Checked" / "Unchecked" of the check box, while the third parameter defines the label of the checkbox. For example:  
**CONFIG = "Y N Confirmation"**  
If the text part must include spaces, add {} curly brackets around the text:  
**CONFIG= "Y N {Order validated}"**  
If the **CLASS** attribute is used with the "KEY" value, the first and second parameters defines the keys to be sent respectively when the checkbox is "Checked" / "Uncheck", and the third parameter defines the label of the checkbox as with normal checkbox usage. For example (line breaks added for document readability):  
**CLASS = "KEY",**  
**CONFIG = "F11 F12 Confirmation"** |
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Effect</th>
<th>Other attributes</th>
</tr>
</thead>
</table>
| COMBO  | The field is presented as a combobox widget. It can be used with the CLASS attribute to change the behavior of the widget. | INCLUDE: This attribute defines the list of acceptable values that will be displayed in the combobox list. For example (line breaks added for document readability):  
\[
\text{INCLUDE} = ("Paris", "London", "Madrid")
\]  
**Important:** The INCLUDE attribute cannot hold value range definitions, because all items must be explicitly listed to be added to the combobox list.  
The following example is not supported:  
\[
\text{INCLUDE} = (1 \ TO\ 10)
\] |
| FIELD_BMP | The field is presented as a normal editbox, plus a button on the right. | CONFIG: The first parameter defines the image file to be displayed in the button; the second parameter defines the key to be sent when the user clicks on the button. For example:  
\[
\text{CONFIG} = "combo.bmp Control-z"
\] |
| LABEL | The field is presented as a simple label, a read-only text. | None. |
| RADIO | The field is presented as a radiogroup widget. | CONFIG: Parameter pairs define respectively the value and the label corresponding to one radio button. For example (line breaks added for document readability):  
\[
\text{CONFIG} = "AA First BB Second CC Third"
\]  
If the radio texts must include spaces, add {} curly brackets around the texts (line breaks added for document readability):  
\[
\text{CONFIG} = "AA \{First\ option}\ BB \{Second\ option\} CC \{Third\ option\}"
\]  
If the CLASS attribute is used with the value "KEY", the first element of each pairs represents the key to be sent when the user selects a radio button. For example (line breaks added for document readability):  
\[
\text{CLASS} = "KEY", \text{CONFIG} = "F11 First F12 Second F13 Third"
\] |

**Controlling old style widgets activation**

The following list of widgets can be enabled or disabled from programs with a DISPLAY TO instruction:

- **Text buttons** (WIDGET="BUTTON")
- Image buttons (WIDGET="BMP")
- Checkboxes of class "KEY" (WIDGET="CHECK", CLASS="KEY")
- Radio buttons of class "KEY" (WIDGET="RADIO", CLASS="KEY")

If you display an exclamation mark in such fields, the button is enabled, but if you display a star (*), it is disabled:

```plaintext
DISPLAY "*" TO button1 # disables the button
DISPLAY "!" TO button1 # enables the button
```

**Changing the text of WIDGET="BMP" fields**

The text of button fields (WIDGET="BUTTON") can be changed from programs with the DISPLAY TO instruction:

```plaintext
DISPLAY "Click me" TO button1
# Sets text and enables the button
```

**Changing the image of WIDGET="BMP" fields:**

The image of button fields (WIDGET="BMP") can be changed from programs with the DISPLAY TO instruction:

```plaintext
DISPLAY "smiley.bmp" TO button1
# Sets image and enables the button
```

Image files are not centralized on the machine where the program is executed; image files must be present on the Workstation. See front end specific documentation for more details.

**Changing the text of WIDGET="LABEL" fields:**

The text of label fields (WIDGET="LABEL") can be changed from programs with the DISPLAY TO instruction:

```plaintext
DISPLAY "Firstname" TO l_firstname
# Sets text of the label field
```

**Using WIDGET="Canvas" fields:**

The fields declared with the WIDGET="Canvas" attribute can be used by the program as drawing areas. Canvas fields must be defined in the LAYOUT section. A set of drawing functions are provided to fill canvas fields with graphical elements.

**Related concepts**

- **Form items** on page 1238
  The concept of *form item* includes all elements used in the definition of a form.

- **Migrating form field WIDGET="type"** on page 240
  Genero BDL supports the now-deprecated BDS fields using the WIDGET attribute, but recommends the use of Genero BDL form item types instead.

**WIDTH attribute**

The WIDTH attribute forces an explicit width of a form element.

**Syntax**

```
WIDTH = integer [CHARACTERS | COLUMNS | POINTS | PIXELS]
```

1. `integer` defines the width of the element.
**Usage**

By default, the width of an element is defined by the size of the form item tag in a grid-based layout, or by the type of the form item in a stack-based layout. Use the `WIDTH` attribute to define a specific width for a form item.

**Note:** As a general rule, consider not specifying a unit, to default to relative characters/lines/columns, instead of specifying exact pixels or points. This is especially important for mobile devices, where the screen resolution can significantly vary depending on the smartphone or tablet model.

**Grid-based layout**

For sizable items like `IMAGE`, the default width is defined by the number of horizontal characters used in the form item tag. Overwrite this default by specifying the `WIDTH` attribute.

```
IMAGE img1: imagel, WIDTH = 20, HEIGHT = 12;
```

For `TABLE/TREE` containers, the default width is defined by the columns used in the table layout. Overwrite the default by specifying the `WIDTH = x COLUMNS` attribute. This will give a small initial width for tables with a large number of columns.

```
TABLE t1: table1, WIDTH = 5 COLUMNS;
```

In a grid-based layout, if you don't specify a size unit, it defaults to `CHARACTERS`, which defines a width based on the characters size in the current font.

**Stack-based layout**

In a stack-based layout, the `WIDTH` attribute cannot be used. The width of form elements are automatically computed.

**Related concepts**

- [Controlling the image layout](#) on page 1148
  
  Explains how image form items can be sized in different front-end layout systems.

- [Item tags](#) on page 1259
  
  *Item tags* define the position and size in a grid-based container.

- [Form rendering basics](#) on page 1415
  
  Get the essentials about form rendering.

- [HEIGHT attribute](#) on page 1368
  
  The `HEIGHT` attribute forces an explicit height for a form element.

**WINDOWSTYLE attribute**

The `WINDOWSTYLE` attribute defines the style to be used by the parent window of a form.

**Syntax**

```
WINDOWSTYLE = "string"
```

1. `string` is a user-defined style name.

**Usage**

The `WINDOWSTYLE` attribute can be used to specify the style of the parent window that will hold the form. This attribute is specific to the `LAYOUT` element. Do not confuse this attribute with the `STYLE` attribute, which is used to specify the decoration style of the form elements.

When a form is loaded by the `OPEN WINDOW` or `DISPLAY FORM` instruction, the runtime system automatically assigns the `WINDOWSTYLE` to the `STYLE` attribute of the parent window element.
Example
LAYOUT { STYLE="BigFont", WINDOWSTYLE="dialog" }

Related concepts
STYLE attribute on page 1395
The STYLE attribute specifies a presentation style for a form element.

WORDWRAP Attribute
The WORDWRAP attribute enables a multiple-line editor in TUI mode.

Syntax

| WORDWRAP [ COMPRESS | NONCOMPRESS ] |

Usage

This attribute is provided for backward compatibility with character-based forms, to support word wrapping in multiline text input.

In GUI mode, it is recommended that you use a TEXTEDIT form item instead. When used, the WORDWRAP attribute is ignored, because text input and display is managed by the text editor widget. The text data is not automatically modified by the editor by adding blank spaces to put words on the next line.

In TUI mode, the WORDWRAP attribute has following effects:

- During input and display, the runtime system treats all segments that have that field tag as segments of a single field.
- The multi-line editor can wrap long character strings to the next line of a multiple-segment field for data entry, data editing, and data display.
- The COMPRESS option prevents blanks produced by the editor from being included in the program variable. COMPRESS is applied by default and can cause truncation to occur if the sum of intentional characters exceeds the field or column size. Because of editing blanks in the WORDWRAP field, the stored value might not correspond exactly to its multiple-line display.
- Specifying NONCOMPRESS after the WORDWRAP keyword causes any editor blanks to be saved when the string value is saved in a database column, in a variable, or in a file.

Using WORDWRAP fields with character-based terminals results in quite different behavior than with graphical front-ends. With character-based terminals, the text input and display is modified by the multi-line editor. This editor can automatically modify the text data by adding blank spaces to put words to the next line, in order to make the text fit into the form field. In GUI mode, the text input and display is managed by a multi-line edit control.

The maximum number of bytes a user can enter is the width of the form-field multiplied by the height of the form-field. Blank characters may be intentional blanks or fill blanks. Intentional blanks are initially stored in the target variable where entered by the user. Fill blanks are inserted at the end of a line by the editor when a newline or a word-alignment forces a line-break. It is not possible to set the cursor at a fill blank. Intentional blanks are always displayed (even on the beginning of a line; the word-wrapping method used in reports with PRINT WORDWRAP works differently).

When entering characters with Japanese locales, special characters are prohibited from being the first or the last character on a line. If the last character is prohibited from ending a line, this character is wrapped down to the next line. If the first character is prohibited from beginning a line, the preceding character will also wrap down to the next line. This method is called kinsoku. The test for prohibited characters will be done only once for the first and the last character on each line.

Word-wrapping is disabled on the last row of a WORDWRAP field. The last word on the last row may by truncated. The WORDWRAP COMPRESS attribute instructs the editor to remove fill blanks before assigning the field-buffer to the target variable. The WORDWRAP NONCOMPRESS attribute instructs the editor to store fill blanks to the target variable. The WORDWRAP and WORDWRAP NONCOMPRESS attributes are equivalent.
The \texttt{WORDWRAP} attribute is not used by the \texttt{CONSTRUCT} instruction.

\textbf{Example}

\textbf{Related concepts}

\textbf{Query by example (\texttt{CONSTRUCT})} on page 1565

The \texttt{CONSTRUCT} instruction implements database query criteria input in an application form.

\textbf{Text mode rendering (TUI mode)} on page 1109

\textbf{Examples}

Form definition (.per) examples.

\textbf{Example 1: Grid-based layout form}

\begin{verbatim}
LAYOUT ( TEXT = "Customer orders" )
  VBOX
    GROUP group1 ( TEXT = "Customer" )
    GRID
    {<GROUP Name            >
     [f001                   ]
     <
     <GROUP Identifiers     ><GROUP Contact     >
     FCode: [f002]  Phone: [f004]
     LNum: [f003]  EMail: [f005]
     <                        >
     }<                        >
    }
  END
END
TABLE
{ OrdNo  Date       Ship date   Weight
  [c01  |c02       |c03        |c04        ]
  [c01  |c02       |c03        |c04        ]
  [c01  |c02       |c03        |c04        ]
  [c01  |c02       |c03        |c04        ]
}END
FOLDER
PAGE pg1 ( TEXT = "Address" )
GRID
{ Address: [f011]
  State: [f012]
  Zip Code: [f013]
}END
END
PAGE pg2 ( TEXT = "Comments" )
GRID
{ [f021]
  [ ]
  [ ]
  [ ]
}END
END
END
\end{verbatim}
### Related concepts

**Grid-based layout** on page 1418  
A form file can define a grid-based layout within a tree of layout items.

**Example 2: Stack-based layout form** on page 1415

#### Example 2: Stack-based layout form

```
SCHEMA stores
  ACTION DEFAULTS
    ACTION import(TEXT=%"action.import")
  END

TABLES
  customer
END

LAYOUT(TEXT=%"title.customer")
STACK
  GROUP
    LABEL, TEXT=%"label.new_customer";
  END --GROUP
  GROUP
    EDIT customer.customer_num, NOENTRY, TITLE=%"label.number";
    EDIT customer.fname, TITLE=%"label.first_name";
    EDIT customer.lname, TITLE=%"label.last_name";
    EDIT customer.company, TITLE=%"label.company";
  END --GROUP
  GROUP(TEXT = "group.address")
    EDIT customer.address1, TITLE=%"label.address1";
    EDIT customer.address2, TITLE=%"label.address2";
    EDIT customer.city, TITLE=%"label.city";
    BUTTONEDIT customer.state, TITLE=%"label.state", UPSHIFT,
        NOTEDITABLE, ACTION = zoom;
    EDIT customer.zipcode, TITLE=%"label.zipcode";
  END --GROUP
GROUP phone_edit
  EDIT customer.phone, TITLE=%"label.telephone",
      KEYBOARDHINT=PHONE;
END --GROUP
GROUP phone_dial
  BUTTON dial, TEXT=%"button.dial";
END --GROUP
END --STACK
END --LAYOUT
```

### Related concepts

**Stack-based layout** on page 1431  
A form file can define a stack-based layout within a tree of stack items.

**Example 1: Grid-based layout form** on page 1414

### Form rendering

The section explains the layout rules to render forms on graphical front-ends.

#### Form rendering basics

Get the essentials about form rendering.

In the graphical mode (GUI mode), forms are not displayed in a fixed text-mode screen. Application windows can display complex layouts and are resizable by the end user, if the platform allows window resizing (mobile devices versus desktop platforms).
When developing with command line tools, forms are designed with .per form specification files, which are text files. In order to display text-based forms in graphical mode, the text-based form definitions must be converted to graphical forms, which implies specific layout rules. These rules are explained in this section.

We distinguish two type of form rendering techniques:

- **Grid-based** rendering, based on a grid of cells, to place and size form elements.
- **Stack-based** rendering, where all form elements are place over each other vertically.

**Related concepts**

- **Form specification files** on page 1237
  Form specification files are the source files defining the layout and content of application forms.

**Character set usage**

The character set used to edit and compile .per form specification files is defined by the current locale.

Form elements (typically, labels) can be written with non-ASCII characters of the current codeset.

In a grid-based layout, the form element positions and sizes are determined by counting the width of characters, rather than the number of bytes identifying the characters in the current codeset. This rule can be ignored when using a single-byte character set such as ISO-8859-1 or CP-1252, where each character has width of 1 and codepoint of 1 byte. This rule is important when using a multibyte character set such as BIG5 or UTF-8.

For example, in the UTF-8 multibyte codeset, a Chinese character is encoded with three bytes, while the visual width of the character is twice the size of a Latin character. In the next example, the labels with three Chinese characters have the same width as the labels using six Latin characters. As a result, all the labels will get the same size (6 cells), and all fields will be aligned properly in a proportional font display:

```
LAYOUT
GRID
{
###[f001  ]abcdef[f002  ]
abcdef[f003  ]###[f004  ]
}
END
END
ATTRIBUTES
f001 = FORMONLY.field1;
f002 = FORMONLY.field2;
f003 = FORMONLY.field3;
f004 = FORMONLY.field4;
END
```

In a stack-based container, the position of form elements is logical, the current locale does not impact on the form item positions as in a grid-based container:

```
LAYOUT
STACK
GROUP
   EDIT FORMONLY.cust_num, TITLE="###";
   EDIT FORMONLY.cust_name;
   EDIT FORMONLY.cust_address;
END
END
END
```

For maximum portability, it is recommended to write all form specification files in ASCII (7 bit), and use localized strings to internationalize your forms.

**Related concepts**

- **Localization** on page 512
Localization support allows you to implement programs that follow specific language and cultural rules.

**Localized strings** on page 538

Localized strings provide a means of writing applications in which the text of strings can be customized on site.

**GRID container** on page 1308

Defines a layout area based on a grid of cells.

**STACK container** on page 1309

The STACK container holds stack items defining a logical alignment of form items.

### Adapting to viewport changes

Application forms and functions can be adapted to the front-end viewport size or mobile device orientation.

#### Detecting viewport size / orientation changes

When the mobile device orientation changes, or when the current window is resized on desktop/web front-ends, the `windowresized` specific predefined action will be sent, if an ON ACTION handler is defined by the current dialog for this action.

**Note:** The `windowresized` action is used only to hide/show items on the current form using the standard user interface API (`ui.Form.setElementHidden()`) and it is not recommended for reloading forms on the fly.

This predefined action can be used to detect viewport geometry changes and adapt the application form to the new size:

```
ON ACTION windowresized
   -- Code to adapt to the new viewport size
```

**Note:** In dialogs allowing field input (INPUT / INPUT ARRAY or CONSTRUCT), take care of the current field input: The `windowresized` action can force the field validation. Therefore, it is not recommended to use this special action in these dialogs. The action can be safely used in DISPLAY ARRAY and MENU dialogs.

To control action view rendering defaults and current field validation behavior when the `windowresized` action is used, consider setting action default attributes for this action in your .4ad file as follows:

```
<ActionDefaultList>
   ...<ActionDefault name="windowresized" validate="no" defaultView="no"
contextMenu="no"/>
   ...
</ActionDefaultList>
```

Another option is to define these action defaults attributes in the ON ACTION handler:

```
ON ACTION windowresized (VALIDATE=NO, DEFAULTVIEW=NO)
   ...
```

### Querying the geometry of the viewport

Use the `feInfo/windowSize` front call to query the actual size of the front-end viewport (GDC current window, GBC webview, or mobile screen size):

```
DEFINE size STRING
CALL ui.Interface.frontCall("standard","feInfo",["windowSize"],[size])
IF size == "1200x1824" THEN
   ...
END IF
```

### Related concepts

- **List of predefined actions** on page 1741
Related reference
Window style attributes on page 1220
Window presentation style attributes apply to a window element.

Front-end stored settings
Front-ends can store some layout properties of windows and form elements, for subsequent program executions.

Purpose of stored settings
With a GUI front-end, when a window/form is closed, the front-end stores widget sizing and positionning properties locally on the platform where the front-end executes. When the window/form is reopened (in a new or in the same program instance), these variable layout properties are restored, to display the forms and its content with the same aspect as when they were closed.

For example, TABLE columns can be resized, reordered, and selected to sort rows. Such variable layout properties are saved in the stored settings and restored when

Managing stored settings on front-end side
Stored settings can be controlled by the end-user, in the front-end configuration panel.

For example, it is possible to disable stored settings completely, or to reset them in order to get the original form layouts.

See front-end specific documentation for more details.

Controlling stored settings with program files
Some window presentation styles attributes such as forceDefaultSettings or position can be used to control stored settings.

Related concepts
Form specification files on page 1237
Form specification files are the source files defining the layout and content of application forms.

Style attributes reference on page 1182
A presentation style attribute may be a common attribute that can be applied to any graphical element. Most presentation style attributes apply only to a specific graphical element.

Grid-based layout
A form file can define a grid-based layout within a tree of layout items.

In a .per form specification file, the LAYOUT section defines a tree of layout containers, which hold form items such as labels and form fields.

The GRID container can be used to define a grid of cells that hold form items: in the layout tree, the GRID container acts as a leaf node, which holds the visible widgets (fields, buttons, and so on).

Note: SCROLLGRID and GROUP containers defined by layout tags inside a grid without the GRIDCHILDRENINPARENT attribute, are similar to GRID containers in regards to the layout rules describe in this section.

The .per form specification file defines a form layout based on a character grid, each character defines a cell of the grid:

```
GRID
{
  First Name [fname  ]
  Last Name  [lname ]
}
END
```
The .per file layout specification can be shown in a character grid.

```
GR ID
{
  1
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
  0 First Name [fname ]
  1 Last Name  [lname ]
}
END
```

**Figure 52: Character grid of a form layout**

With a fixed-font based front-end (such as a dumb terminal), the forms appear within a screen where each cell is identified by x and y coordinates, as in the SCREEN section of the form specification file. There is no particular layout issue, as all characters can be displayed at the same (relative) position as in the source form file.

With the graphical front-end, text-based forms must be displayed in a graphical window using fonts with a proportional size. In a proportional font, the field label "Key" has a different graphical length than the label "Num", despite having the same number of characters.

In the compiled version of the form specification file, all form items get coordinates in a virtual grid (defined by posX and posY attributes), and the number of cells the item occupies in the grid (in the gridWidth and gridHeight attributes):
The "First Name" and "Last Name" texts are identified as whole labels, even if the words "First" and "Name" (or "Last" and "Name") are not joined in the form definition, because the form compiler considers a single blank as a word separator within labels.

**Related concepts**

GRID container on page 1308
Defines a layout area based on a grid of cells.

STACK container on page 1309
The STACK container holds stack items defining a logical alignment of form items.

**Packed and unpacked grids**

When resizing a window, the content will either grow with the window or be packed in the top left position.

If elements in the window can grow, they will follow the window container and resize accordingly. Some elements can grow vertically, some can grow horizontally, and some can grow in both directions. The way resizable form items can grow is controlled by the STRETCH attribute. The window content is packed horizontally, vertically or in both directions, if none of the elements can grow in that direction.

The following form item types can grow horizontally:

- TABLE / TREE items
- IMAGE items (with STRETCH=BOTH or STRETCH=X)
- TEXTEDIT items (with STRETCH=BOTH or STRETCH=X)
The following form item types can grow vertically:

- **TABLE / TREE** items (without `WANTFIXEDPAGESIZE` attribute)
- **IMAGE** items (with `STRETCH=BOTH` or `STRETCH=Y`)
- **TEXTEDIT** items (with `STRETCH=BOTH` or `STRETCH=Y`)

In general, a **GRID** container can grow if any object inside the **GRID** can grow. There is an exception to this rule. If there is a single **GROUP** container (defined without the `GRIDCHILDRENINPARENT` attribute) inside a **GRID** and nothing else, the grid can grow even if the objects inside the grid cannot grow.

This exception allows better rendering of a grouped grid.

**Figure 54: Packed grid**
Automatic HBoxes and VBoxes
Horizontal and vertical boxes are added automatically when stretchable elements are used.

When using layout tags in a GRID container, the fglform compiler will automatically add hbox or vbox containers with splitters in the following conditions:

- An hbox is created when two or more stretchable elements are stacked side by side and touch each other (no space between).
- A vbox is created when two or more stretchable elements are stacked vertically and touch each other (no space between).

No hbox or vbox will be created if the elements are in a SCROLLGRID container.

This example defines two tables stacked vertically, generating a vbox with splitter. The ending tags for the tables are omitted.

```
<T table1  
[ colA  | colB    ]
[ colA  | colB    ]
[ colA  | colB    ]
[ colA  | colB    ]
</T table1>

<T table2  
[ colC  | colD    ]
[ colC  | colD    ]
[ colC  | colD    ]
</T table2>
```

This example defines a layout with two stretchable TEXTEDIT fields placed side by side, which would generate an automatic hbox with splitter. To make both widgets touch, you need to use a pipe delimiter in between the two widgets.

```
[ textedit1 | textedit2   ]
[            |            ]
[            |            ]
[            |            ]
```
Related concepts

Layout tags on page 1254

*Layout tags* define layout areas for containers inside the frame of a grid-based container.

STRETCH attribute on page 1394

The **STRETCH** attribute specifies how a widget must resize when the parent container is resized.

GRIDCHILDRENINPARENT attribute on page 1367

The **GRIDCHILDRENINPARENT** attribute is used for a container to align its children to the parent container.

Widget position and size in grid

Form items render as widgets in the window, at a given position and with a given size.

To render form items, grid-based rendering follows the layout rules described below:

1. The position of the widgets in the virtual grid is defined by the `posX` and `posY` AUI tree attributes.
2. The number of virtual grid cells occupied by a widget is defined by the `gridWidth` and `gridHeight` AUI tree attributes.
3. The real size (i.e. pixels) of a widget is defined by the `width` and `height` AUI tree attributes.
4. Empty lines and empty columns in the form layout definition take a size of 0 pixels.
5. The size of a cell in the virtual grid depends on the real size of the widgets inside the grid.
6. A widget's minimum size is computed via its real size and the SAMPLE attribute.
7. The preferred size of the widget is computed following the SIZEPOLICY attribute.
8. The final widget size is computed depending on the minimum and preferred size, to fill the cells in the grid.
9. A small spacing is applied in non-empty cells.

The next screenshot shows 2 labels and 2 fields placed in a grid.

![Figure 56: Two labels and two fields placed in a grid: Grid view](image)

By default, empty grid rows and empty grid columns get no size when rendered on the front-end. For example, in the above grid sample, the grid columns #10 and #11 are empty.

**Form item dependencies in grids**

Form items interact with each other in terms of width, depending on the front-end widget size.

This example illustrates how form items are dependent on each other inside the grid.
This .per implies that form items a and b start at the same position and have the same size, whatever a and b are. This rule leads to very different results, especially when a large widget is assigned into a small number of cells.

Example:

```
LAYOUT
GRID
{
  [a  b  ][f  ]
  [c  d  ][e  ]
}
END

ATTRIBUTES
CHECKBOX a = formonly.a, TEXT="A Checkbox"
EDIT b = formonly.b;
EDIT c = formonly.c;
CHECKBOX d = formonly.d, TEXT="Another Checkbox"
EDIT e = formonly.e;
EDIT f = formonly.f;
END
```

The grid is computed with regard to the character cells in the form definition:

```
| 1 |
| 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 |
|GRID|
|{|
|0 [a b ] [f ]|
|1 [c d ] [e ]|
|}|
|END|
```

**Figure 58: Grid layout**
Then the minimum size of each widget and the layout is computed.

Cells (0,1) and (1,3) contain a checkbox; these checkboxes will enlarge columns 1 and 3.

Because the EDIT field "c" is defined to have the same width as checkbox "a", it will be much larger as expected.

To avoid this visual result, you must assign a realistic number of grid cells for each form item.
Even if the grid area is wider in the source form file, the real graphical result will be smaller.

**Figure 62: Resulting form**

**Related concepts**

- **Item tags** on page 1259
  - *Item tags* define the position and size in a grid-based container.

- **Hbox tags** on page 1261
  - *Hbox tags* group several item tags within the same horizontal layout box, inside a grid-based container (**GRID**).

**Complex grid layout example**

Describes how form items align in grid-based front-ends with an example.

These diagrams show the virtual grid of a complex form, with several field item tags.

```
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**GRID**

```
{ 
  0 [ a | b ] [ c ] 
  1 [ d ] [ e | f | g ] 
  2 [ h | i | j | k ] [ l ] 
}
```

**Figure 63: Grid containing several fields**

For each form field, the position and the number of cells is computed by the form compiler.

At runtime, the front-end creates the widgets and sets them on the virtual grid.
Once widgets are on the grid, their minimum size is computed based on widget size, SAMPLE and SIZEPOLICY attributes.

Then the sizes of the grid cells adapt to the size of the widgets.

In this screenshot, the fields \( k \) and \( c \) are much bigger than expected:

- Field \( g \) and \( l \) make columns 33, 34 and 35 bigger than the others,
- Field \( f \) extends columns 25 to 31.
- As field \( c \) has to fill columns 25 to 35, its size grows; the same for field \( k \).

Some fields are proportionally bigger than others because some parameters are variable, while others are fixed.
The width of the widget is the sum of border width, plus the content width (depending on SIZEPOLICY and the SAMPLE attributes).

Since the default SAMPLE is MMMMMM000..., the graphical width of a field is not linearly proportional to the width defined in the form file. For example, a field of 1 will be as wide as 2 borders + 1 'M'. A field of 10 will be as wide as 2 borders + 6 'M' + 4 '0'. This means that a field of 1 is far from being 10 times smaller than a field of 10.

**Using hbox tags to align form items**

The hbox tag concept has been introduced to bypass the limitations of the character-based grid in forms.

An hbox tag defines a widget container that arranges child widgets horizontally, like the horizontal box container. All widgets inside this container are no longer dependent on the parent grid.

**Defining hbox tags in grids**

An hbox tag is defined by using a : colon in an item tag delimited by square brackets.

This example creates a hbox container containing widgets a, b and c. These widgets won't be aligned in the grid.

```
GRID
{
[a:b:c    ]
[d|e|f    ]
}
END
```

**Figure 67: Using an hbox tag**

```
0 1 2 3 4 5 6 7 8 9
GRID
{
0 [ a : b : c   ]
1 [ d | e | f   ]
}
END
```

**Figure 68: HBox tag rendering**

Hbox tags are useful when the form contains large widgets in a small number of cells in one row, and don't want to have dependencies.
Figure 69: Using an HBox tag

We can modify the Form item dependencies in grids on page 1423 example, using hbox tags:

```
GRID
{
  [a:b ] [f     ]
  [c:d ] [e     ]
}
END
```

Figure 70: HBox rendering

Related concepts

Automatic HBoxes and VBoxes on page 1422
Horizontal and vertical boxes are added automatically when stretchable elements are used.

Spacer items in hbox tags

HBox tags also introduces the spacer items concept: when a grid hbox is created, the content may be smaller than the container.

```
GRID
{
  [a :b :c ]
  [d :e :f ]
}
END
```

Figure 71: Spacer items
Because of the checkbox, cell 1 is very large, and then the hbox is larger than the three fields. A spacer item object is automatically created by the form compiler; the role of the spacer item is to take all the free space in the container. Then all the widgets are packed to the left side of the hbox.

By default, a spacer item is created at the right of the container, but the spacer can also be defined in another place:

```
GRID
{
[a      :b      :c       ] <- default: spacer on the right
[ :d    :e      :f       ] <- spacer on the left
[g      :     :h         ] <- spacer between g and h
[i:j]:k : :l       ] <- multiple spacers (between i and j, j and k, k
 and l
}
END
```

**Figure 72: Form using spacers**

**Widget size within hbox tags**

By default, the real width of `BUTTONEDIT`, `DATEEDIT`, `DATETIMEEDIT` and `COMBOBOX` widgets are computed as follows:

```
if item-tag-width > 2
    real-width = item-tag-width - 2
else
    real-width = item-tag-width
```

Where `item-tag-width` represents the number of characters used in the form layout by the item tag, to define the width of the element.

If the default widget size computing does not satisfy the needs, it is possible to specify the exact with of a `BUTTONEDIT`, `DATEEDIT` or `COMBOBOX` with an hbox tag, combined to the `SAMPLE` attribute.

The hbox tag can be used with a : (colon) and - (dash) marker to define the exact number of characters the field can display, while the `SAMPLE` attribute will define the size.

For example:
Here the \( ba \) item tag occupies 7 grid columns and gets a real width of 5 (7-2). The `SAMPLE` attribute makes the edit field part as large as 5 characters '0' in the current font, so with this field you can input or display only 5 digits.

The \( bb \) item tag, which is in an hbox tag that occupies 7 grid columns, gets a width of 2. Since the `SAMPLE` attribute is "M", one can input 2 characters as wide as an "M".

The \( bc \) item tag, which is in an hbox tag that occupies 7 grid columns, gets a width of 3 (5-2). Since the `SAMPLE` attribute is "Pi", the edit field part will be as large as the word "Pi". (If `SAMPLE` contains more than 1 character it must have the same number of characters as in the field definition).

When using an hbox tag, one can explicitly specify the width of the field with the dash size indicator: The \( bd \), which is in an hbox tag that occupies 7 grid columns, gets a width of 4 (because of the dash size indicator). Since the `SAMPLE` attribute is "0", the edit field part will be as large as 4 digits.

Related concepts

Hbox tags on page 1261

`Hbox tags` group several item tags within the same horizontal layout box, inside a grid-based container (GRID).

Item tags on page 1259

`Item tags` define the position and size in a grid-based container.

Stack-based layout

A form file can define a stack-based layout within a tree of stack items.

In a .per form specification file, the `LAYOUT` section defines a tree of layout containers, which hold layout items such as labels and form fields.

Use the `STACK` layout container, to define a logical grouping of form elements, to be rendered vertically by the front-end.

Important: This feature is experimental. The syntax/name and semantics/behavior may change in a future version.

The `STACK` container must appear immediately after the `LAYOUT` keyword.

A `STACK` layout defines a tree of stacked containers, which holds a set of stack items such as form fields. In the sample code, the `STACK` container holds a `GROUP` container, which defines some stacked `EDIT` fields:

```
LAYOUT
STACK
  GROUP g1(TEXT="Customer info")
    EDIT customer.cust_num, NOENTRY, TITLE="Id:";
    EDIT customer.cust_name, TITLE="Name:";
    EDIT customer.cust_address, TITLE="Address:";
  END
END
END
```
There is no such thing as x,y coordinates in a stack container: The form element position definition is abstract and relative to other elements. Arranging form elements logically allows more flexibility in the final rendering of the form on the front-end.

Stack-based forms are typically used in mobile application design, to get a similar, but adaptable layout rendering on different mobile device brands.

The visual result of the above form definition would look as follows on an iOS mobile device:

![Figure 73: iOS stacked form 2](image)

**Related concepts**

- [Grid-based layout](#) on page 1418
  A form file can define a grid-based layout within a tree of layout items.

- [GRID container](#) on page 1308
  Defines a layout area based on a grid of cells.

**Label internationalization**

Define form files with stacked containers for different languages.

To internationalize your application, define `TITLE` attributes using `%string-id` localized strings, in stack contains and stack item definitions:

```plaintext
-- myform.per
LAYOUT
STACK
  GROUP (TEXT=%"group.custinfo")
    EDIT customer.cust_num, NOENTRY, TITLE=%"cust.label.id";
    EDIT customer.cust_name, TITLE=%"cust.label.name";
    EDIT customer.cust_address, TITLE=%"cust.label.address";
  END
END
ED

-- myapp.str
```
"group.custinfo" = "Customer information"
"cust.label.id" = "Id:"
"cust.label.name" = "Name:"
"cust.label.address" = "Address:"

If more space is needed for text fields, remove field labels and add a COMMENT attribute to show a grayed text inside empty fields:

```plaintext
-- myform.per
LAYOUT
STACK
  GROUP (TEXT="group.custinfo")
    EDIT customer.cust_num, NOENTRY, TITLE="cust.label.id";;
    EDIT customer.cust_name, COMMENT="cust.comment.name";
    EDIT customer.cust_address, COMMENT="cust.comment.address";
  END
END
END

-- myapp.str
"group.custinfo" = "Customer information"
"cust.label.id" = "Id:"
"cust.comment.name" = "Customer's name"
"cust.comment.address" = "Customer's address"
```

The visual result for the about stack-based form will look like this on an iOS device:

![iOS stacked form 3](image)

Figure 74: iOS stacked form 3

Related concepts
Localized strings on page 538
*Localized strings* provide a means of writing applications in which the text of strings can be customized on site.

**Stacked group rendering**
Groups render in a native way on front-ends supporting the stacked layout.

Use `GROUP` containers in your form definition, to control the stacked layout: Fields and other form elements such as buttons can be grouped together by domain.

The header of a group box is defined by the `TEXT` attribute of the `GROUP` container.

For example, in a form designed for customer data input, customer identification (number, name) appear in a dedicated group, while address information (street, zip code, state, country fields) may appear under another group:

```plaintext
-- myform.per
LAYOUT STACK
  GROUP g1 (TEXT=%"cust.group1")
    EDIT FORMONLY.id, TITLE=%"cust.label.id";
    EDIT FORMONLY.name, TITLE=%"cust.label.name";
  END
GROUP g2 (TEXT=%"cust.group2")
  LABEL : l_street, TEXT=%"cust.label.street";
  TEXTEDIT FORMONLY.street, HEIGHT=3;
  EDIT FORMONLY.zipcode, TITLE=%"cust.label.zipcode";
  EDIT FORMONLY.state, TITLE=%"cust.label.state";
  EDIT FORMONLY.country, TITLE=%"cust.label.country";
  END
END
END

-- myapp.str
"cust.group1" = "Customer id"
"cust.label.id" = "Id:"
"cust.label.name" = "Name:"
"cust.group2" = "Address"
"cust.label.street" = "Street:
"cust.label.zipcode" = "Zip Code:"
"cust.label.state" = "State:"
"cust.label.country" = "Country:"
```

This code example will render as follows on an iOS mobile device:
Figure 75: iOS stacked form 4

Related concepts

GROUP stack item on page 1315
Defines a stack area to group other layout elements together, in a stack-based layout.

GROUP container on page 1306
Defines a layout area to group other layout elements together, in a grid-based layout.

Layout tags on page 1254
Layout tags define layout areas for containers inside the frame of a grid-based container.

Toolbars

Toolbars define a bar of buttons that appears at the top of application forms.

Understanding toolbars

This is an introduction to toolbars definition.

A toolbar defines action views presented as a set of buttons that can trigger events in an interactive instruction.

This section describes how to define toolbars with XML in files or in programs as global/default toolbars; it is also possible to define toolbars in forms with the TOOLBAR section, as form-specific toolbars.

Toolbar files can be loaded by a program with the methods ui.Interface.loadToolBar() (for global/default toolbars) or ui.Form.loadToolBar() (to load form-specific toolbars).

The global/default toolbar is displayed in all windows, or in the global window container when using a window container. The form-specific toolbar is displayed in the form where it is defined.

The position and overall rendering of toolbar can be defined with presentation style attributes of window elements, such as toolBarPosition. Typical "modal windows" (windowType="modal") do not display toolbars. See FGLDIR/lib/default.4st, where toolBarPosition is set to "none" for that kind of windows.

The toolbar items (or buttons) are enabled/disabled based on the ON ACTION handlers defined by the current interactive instruction. A toolbar item is bound to an action handler by name. A click on the toolbar button will execute the user code in the action handler.

Toolbar elements can get a style attribute in order to use a specific rendering/decoration following presentation style definitions. Specific toolbar style attributes are supported to customize your toolbars.

The DOM tag names are case sensitive; Toolbar is different from ToolBar.

When binding to an action, make sure that you are using the right value in the name attribute. As ON ACTION and COMMAND generate lowercase identifiers, it is recommended to use lowercase names.

It is recommended that you define the decoration of toolbar items for common actions with action defaults.

Related concepts

Configuring actions on page 1744

Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with action attributes.

Defining toolbars in the form file

Toolbars can be defined in the form specification file within the TOOLBAR section.

Form toolbars are only displayed in the window where the form is loaded. Only one toolbar can be defined in a form file. It is recommended that toolbar button attributes that are common to topmenu options are centralized in action defaults.

Example

```xml
TOOLBAR tb
    ITEM accept ( TEXT="Ok", IMAGE="ok" )
    ITEM cancel ( TEXT="Cancel", IMAGE="cancel" )
    SEPARATOR
    ...
END

LAYOUT
GRID
{
    ...
```
Related concepts

**TOOLBAR section** on page 1298
The **TOOLBAR** section defines a toolbar with buttons that are bound to actions.

**Configuring actions** on page 1744
Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with *action attributes*.

**Syntax of a toolbar file (.4tb)**
A .4tb toolbar file is an XML file that holds a tree of elements defining a toolbar structure.

**Syntax: XML Toolbar**

```xml
<ToolBar [
  toolbar-attribute="value" [...] >
  [...] <ToolBarSeparator separator-attribute="value" [...] />
  [...] <ToolBarItem item-attribute="value" [...] />
...</ToolBar
```

1. *toolbar-attribute* defines a property of the toolbar.
2. *item-attribute* defines a property of the toolbar item.

**Toolbar XML attributes**

**Table 355: ToolBar node attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>style</td>
<td>STRING</td>
<td>Use to decorate the element with a presentation style.</td>
</tr>
<tr>
<td>tag</td>
<td>STRING</td>
<td>User-defined attribute to identify the node.</td>
</tr>
<tr>
<td>name</td>
<td>STRING</td>
<td>Identifies the toolbar.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Defines if the text of toolbar buttons must appear by default (0 = visible, 1 = hidden).</td>
</tr>
<tr>
<td>buttonTextHidden</td>
<td>INTEGER</td>
<td>Note: On front-ends where the toolbar button texts can be hidden with an option (context menu), the stored settings take precedence over the BUTTONTEXTHIDDEN attribute.</td>
</tr>
</tbody>
</table>

**Table 356: ToolBarItem node attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>STRING</td>
<td>Identifies the action corresponding to the toolbar button. Can be prefixed with the sub-dialog identifier.</td>
</tr>
<tr>
<td>style</td>
<td>STRING</td>
<td>Use to decorate the element with a presentation style.</td>
</tr>
<tr>
<td>tag</td>
<td>STRING</td>
<td>User-defined attribute to identify the node.</td>
</tr>
<tr>
<td>text</td>
<td>STRING</td>
<td>The text to be displayed in the toolbar button.</td>
</tr>
<tr>
<td>comment</td>
<td>STRING</td>
<td>The message to be shown as tooltip when the user selects a toolbar button.</td>
</tr>
<tr>
<td>hidden</td>
<td>INTEGER</td>
<td>Indicates if the item is hidden (0 = visible, 1 = hidden).</td>
</tr>
<tr>
<td>image</td>
<td>STRING</td>
<td>The icon to be used in the toolbar button.</td>
</tr>
</tbody>
</table>
Table 357: ToolBarSeparator node attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>STRING</td>
<td>Identifies the toolbar separator.</td>
</tr>
<tr>
<td>style</td>
<td>STRING</td>
<td>Use to decorate the element with a presentation style.</td>
</tr>
<tr>
<td>tag</td>
<td>STRING</td>
<td>User-defined attribute to identify the node.</td>
</tr>
<tr>
<td>hidden</td>
<td>INTEGER</td>
<td>Indicates if the separator is hidden (0 = visible, 1 = hidden).</td>
</tr>
</tbody>
</table>

Related concepts

Configuring actions on page 1744
Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with action attributes.

Presentation styles on page 1165
Use presentation styles to specify decoration attributes for window and form elements.

Binding action views to action handlers on page 1762
How are action views of the forms bound to action handlers in the program code?

Loading .4tb toolbar definition files
Toolbar XML definition files can be loaded at runtime.

Loading an XML toolbar file to define a default/global toolbar
To load a .4tb toolbar definition file as default toolbar for all forms, use the `ui.Interface.loadToolbar()` class method:

```plaintext
CALL ui.Interface.loadToolbar("standard")
```

The purpose of the default/global toolbar will be displayed in all forms.

Loading a XML toolbar file for the current form/window
To load a .4tb toolbar definition file for a given form, use the `ui.Form.loadToolbar()` method:

```plaintext
DEFINE myform ui.Form
...
CALL myform.loadToolbar("standard")
```

The toolbar will be displayed in that form only.

This method is typically used in form initializers.

Related concepts

Syntax of a toolbar file (.4tb) on page 1437
A .4tb toolbar file is an XML file that holds a tree of elements defining a toolbar structure.

Creating the toolbar manually with DOM
Toolbars can be created at runtime by creating the corresponding XML representation in the AUI tree.

This example shows how to create a toolbar in all forms by using the default initialization function and the `om.DonNode` class:

```plaintext
CALL ui.Form.setDefaultInitializer("myinit")
OPEN FORM f1 FROM "form1"
DISPLAY FORM f1
...
FUNCTION myinit(form)
DEFINE form ui.Form
```
DEFINE f om.DomNode
LET f = form.getNode()
...
END FUNCTION

After getting the DOM node of the form, create a node with the "ToolBar" tag name:

DEFINE tb om.DomNode
LET tb = f.createChild("ToolBar")

For each toolbar button, create a sub-node with the "ToolBarItem" tag name and set the attributes to define the button:

DEFINE tbi om.DomNode
LET tbi = tb.createChild("ToolBarItem")
CALL tbi.setAttribute("name","update")
CALL tbi.setAttribute("text","Modify")
CALL tbi.setAttribute("comment","Modify the current record")
CALL tbi.setAttribute("image","change")

If needed, you can create a "ToolBarSeparator" node to separate toolbar buttons:

DEFINE tbs om.DomNode
LET tbs = tb.createChild("ToolBarSeparator")

Related concepts
The DomNode class on page 2584
The `om.DomNode` class provides methods to manipulate a DOM node of a data tree.
Example 2: Toolbar created dynamically on page 1443

Toolbars on mobile devices
Toolbars can be used to control action view rendering on mobile devices.

Rendering of default action views
On mobile devices, actions render usually as default action views, that show up implicitly in dedicated panes on the screen, following the platform native user interface rules.

When displaying forms on a mobile front-end, you can use a toolbar to control the rendering of the actions.
For more details, see Action views on mobile devices on page 1777.

Toolbar action views in GBC chrome (Universal Rendering)
When using the GBC front-end on a mobile device (for example with Universal Rendering), it is possible to display the toolbar action views in the GBC chrome bar, by setting the "chrome" value in the `toolBarPosition` style attribute of the Window class:

```xml
<Style name="Window">
...
  <StyleAttribute name="toolBarPosition" value="chrome" />
</Style>
```
Figure 76: Google Nexus 5 display with toolbar action views in the GBC chrome

For more details about GBC chrome related style attributes, see Action views with GBC on page 1769.

Using toolbars for Android™ devices (GMA)

On Android devices, a TOOLBAR can be used to define the action views that appear in the Android action bar.

Toolbar action views are listed first and ordered as they are defined in the TOOLBAR section, followed by the default action views for remaining actions that are not part of the TOOLBAR definition.
Using toolbars for iOS devices (GMI)

On iOS devices, a TOOLBAR renders as the iOS toolbar panel. This toolbar appears at the bottom of the screen, displaying a icon or text for each toolbar item. If there is not enough space to render all toolbar items, a three-dot overflow icon appears on the right, to show up the remaining toolbar items.
The `iosSeparatorStretch` ToolBarSeparator style attribute can be used to stretch the toolbar separators to give more space between action buttons.
The background color for the iOS toolbar must be defined with the `iosToolbarTintColor` style attribute at the Window level.

In order to define the text color for iOS toolbar elements, define the `iosTintColor` style attribute at the Window level. However, other form elements such as Folder, Button, SpinEdit and RadioGroup will be impacted by the attribute defined for Window elements. To overwrite this, define another `iosTintColor` for the Form elements. For more details, see Decorate iOS UI elements on page 3582.

**Related concepts**

Action views on mobile devices on page 1777

Action views are rendered following mobile specific standards.

**Examples**

Toolbar usage examples.

**Example 1: Toolbar in XML format**

```xml
<ToolBar style="mystyle">
    <ToolBarItem name="f5" text="List" image="list" />
    <ToolBarSeparator/>
    <ToolBarItem name="query" text="Query" image="search" />
    <ToolBarItem name="add" text="Append" image="add" />
    <ToolBarItem name="delete" text="Delete" image="delete" />
    <ToolBarItem name="modify" text="Modify" image="change" />
    <ToolBarSeparator/>
    <ToolBarItem name="f1" text="Help" image="list" />
    <ToolBarSeparator/>
    <ToolBarItem name="quit" text="Quit" image="quit" />
</ToolBar>
```

**Example 2: Toolbar created dynamically**

```plaintext
MAIN
DEFINE aui om.DomNode
DEFINE tb om.DomNode
DEFINE tbi om.DomNode
DEFINE tbs om.DomNode

LET aui = ui.Interface.getRootNode()
LET tb = aui.createChild("ToolBar")
LET tbi = createToolBarItem(tb,"f1","Help","Show help","help")
LET tbs = createToolBarSeparator(tb)
LET tbi = createToolBarItem(tb,"upd","Modify","Modify current record","change")
LET tbi = createToolBarItem(tb,"del","Remove","Remove current record","delete")
LET tbi = createToolBarItem(tb,"add","Append","Add a new record","add")
LET tbs = createToolBarSeparator(tb)
LET tbi = createToolBarItem(tb,"xxx","Exit","Quit application","quit")

MENU "Example"
COMMAND KEY(F1)
    DISPLAY "F1 action received"
COMMAND "upd"
    DISPLAY "Update action received"
COMMAND "Del"
    DISPLAY "Delete action received"
COMMAND "Add"
    DISPLAY "Append action received"
COMMAND "xxx"
EXIT PROGRAM
```
Example 3: Toolbar section in form file

```xml
TOOLBAR (STYLE="mystyle")
  ITEM accept (TEXT="Ok", IMAGE="ok")
  ITEM cancel (TEXT="cancel", IMAGE="cancel")
  SEPARATOR
  ITEM editcut -- Gets decoration from action defaults
  ITEM editcopy -- Gets decoration from action defaults
  ITEM editpaste -- Gets decoration from action defaults
  SEPARATOR
  ITEM append (TEXT="Append", IMAGE="add")
  ITEM update (TEXT="Update", IMAGE="modify")
  ITEM delete (TEXT="Delete", IMAGE="del")
  ITEM search (TEXT="Search", IMAGE="find")
END
```

Topmenus

Topmenus define typical pull-down menus that appear at the top of application forms.

Understanding topmenus

This is an introduction to tompmenu definitions.

A tompmenu defines a graphical menu that holds views for actions controlled in programs with ON ACTION handlers. A tompmenu renders to the user following the front-end platform standards.

On a desktop / web front-end, the tompmenu appears as a typical pull-down menu.

On mobile devices, a tompmenu displays as a flat list of options (Android™), and as a set of option screens the user can drill down (iOS).

This section describes how to define tompmenus with XML in files or in programs as global/default tompmenus; it is also possible to define tompmenus in forms with the TOPMENU section, as form-specific tompmenus.

Tompmenu files can be loaded by program with the methods `ui.Interface.loadTopMenu()` (for default tompmenus) or `ui.Form.loadTopMenu()` (for form-initializers).
In the abstract user interface tree, the TopMenu node must be created under the Form node, and must contain TopMenuGroup nodes. The TopMenuGroup nodes group topmenu commands and other topmenu groups. A TopMenuCommand is a leaf node in the topmenu tree that will trigger an action:

```
TopMenu
  +- TopMenuGroup
  +- TopMenuCommand
  +- TopMenuCommand
  +- TopMenuCommand
  +- TopMenuGroup
    +- TopMenuCommand
    +- TopMenuCommand
    +- TopMenuGroup
    +- TopMenuCommand
    +- TopMenuCommand
    +- TopMenuCommand
    +- TopMenuCommand
```

The topmenu options are enabled based on the ON ACTION handlers defined by the current interactive instruction. A topmenu option is bound to an action handler by name. Selecting the topmenu option will execute the user code in the action handler.

Topmenu elements can get a style attribute in order to use a specific rendering/decoration following presentation style definitions.

The DOM tag names are case sensitive; Topmenu is different from TopMenu.

When binding to an action, make sure that you are using the right value in the name attribute. As ON ACTION and COMMAND generate lowercase identifiers, it is recommended to use lowercase names.

It is recommended that you define the decoration of topmenu options for common actions with action defaults.

Images cannot be displayed for the first level of TopMenuGroup elements.

**Related concepts**

[Configuring actions](#) on page 1744

Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with *action attributes*.

**Defining the topmenu in a form file**

Topmenus can be defined in the form specification file within the TOPMENU section.

Form topmenus will only be displayed in the window where the form is loaded. Only one topmenu can be defined in a form file. It is recommended that topmenu item attributes that are common to toolbar buttons are centralized in action defaults.

**Example**

```
TOPMENU tm
  GROUP form (TEXT="Form")
    COMMAND help (TEXT="Help", IMAGE="quest")
    COMMAND quit (TEXT="Quit")
  END
  ...
END

LAYOUT
GRID
{...
```

**Related concepts**

[TOPMENU section](#) on page 1297
The **TOPMENU** section defines a pull-down menu with options that are bound to actions.

[Configuring actions](#) on page 1744

Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with **action attributes**.

**Syntax of a topmenu file (.4tm)**

A `.4tm` topmenu file is an XML file that holds a tree of elements defining a topmenu structure.

**Syntax: XML Topmenu**

```xml
<TopMenu [ topmenu-attribute="value" [....] ]>
  group
  [....]
</TopMenu>
```

where **group** is:

```xml
<TopMenuGroup group-attribute="value" [....]>
  [ .... ]
  <TopMenuSeparator separator-attribute="value" [....] />
  [ .... ]
  <TopMenuCommand command-attribute="value" [....] />
  [ .... ]
</TopMenuGroup>
```

1. **topmenu-attribute** defines a property of the **TopMenu**.
2. **group-attribute** defines a property of a **TopMenuGroup**.
3. **command-attribute** defines a property of a **TopMenuCommand**.
4. **separator-attribute** defines a property of a **TopMenuSeparator**.

**Topmenu XML attributes**

**Table 358: TopMenu node attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>STRING</td>
<td>Identifies the topmenu.</td>
</tr>
<tr>
<td>style</td>
<td>STRING</td>
<td>Can be used to decorate the element with a presentation style.</td>
</tr>
<tr>
<td>tag</td>
<td>STRING</td>
<td>User-defined attribute to identify the node.</td>
</tr>
</tbody>
</table>

**Table 359: TopMenuCommand node attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>STRING</td>
<td>Identifies the action corresponding to the topmenu command.</td>
</tr>
<tr>
<td>style</td>
<td>STRING</td>
<td>Can be prefixed with the sub-dialog identifier.</td>
</tr>
<tr>
<td>tag</td>
<td>STRING</td>
<td>User-defined attribute to identify the node.</td>
</tr>
<tr>
<td>text</td>
<td>STRING</td>
<td>The text to be displayed in the pull-down menu option.</td>
</tr>
<tr>
<td>comment</td>
<td>STRING</td>
<td>The message to be shown for this element.</td>
</tr>
<tr>
<td>hidden</td>
<td>INTEGER</td>
<td>Indicates if the command is hidden (0 = visible, 1 = hidden).</td>
</tr>
<tr>
<td>image</td>
<td>STRING</td>
<td>The icon to be used in the pull-down menu option.</td>
</tr>
</tbody>
</table>
### Table 360: TopMenuGroup node attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acceleratorName</td>
<td>STRING</td>
<td>Defines the accelerator name to be display on the left of the menu option text. Note this attribute is only used for decoration (you must also define an action default accelerator).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>STRING</td>
<td>Identifies the topmenu group.</td>
</tr>
<tr>
<td>style</td>
<td>STRING</td>
<td>Can be used to decorate the element with a presentation style.</td>
</tr>
<tr>
<td>tag</td>
<td>STRING</td>
<td>User-defined attribute to identify the node.</td>
</tr>
<tr>
<td>text</td>
<td>STRING</td>
<td>The text to be displayed in the pull-down menu group.</td>
</tr>
<tr>
<td>comment</td>
<td>STRING</td>
<td>The message to be shown for this element.</td>
</tr>
<tr>
<td>hidden</td>
<td>INTEGER</td>
<td>Indicates if the group is hidden (0 = visible, 1 = hidden).</td>
</tr>
<tr>
<td>image</td>
<td>STRING</td>
<td>The icon to be used in the pull-down menu group.</td>
</tr>
</tbody>
</table>

### Table 361: Separator-attributes for the TopMenuSeparator node

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>STRING</td>
<td>Identifies the topmenu separator.</td>
</tr>
<tr>
<td>style</td>
<td>STRING</td>
<td>Can be used to decorate the element with a presentation style.</td>
</tr>
<tr>
<td>tag</td>
<td>STRING</td>
<td>User-defined attribute to identify the node.</td>
</tr>
<tr>
<td>hidden</td>
<td>INTEGER</td>
<td>Indicates if the separator is hidden (0 = visible, 1 = hidden).</td>
</tr>
</tbody>
</table>

**Related concepts**

- **Configuring actions** on page 1744
  Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with *action attributes*.

- **Presentation styles** on page 1165
  Use presentation styles to specify decoration attributes for window and form elements.

- **Binding action views to action handlers** on page 1762
  How are action views of the forms bound to action handlers in the program code?

- **Loading .4tm topmenu definition files**
  Topmenu XML definition files can be loaded at runtime.

- **Loading an XML topmenu file to define a default/global topmenu**
  To load a .4tm topmenu definition file as default/global topmenu for all forms, use the `ui.Interface.loadTopmenu()` class method:

  ```java
  CALL ui.Interface.loadTopmenu("standard")
  ```

  The purpose of the default/global topmenu will be displayed in all forms.
Loading a XML topmenu file for the current form/window

To load a .4tm topmenu definition file for a given form, use the `ui.Form.loadTopmenu()` method:

```
DEFINE myform ui.Form
...
CALL myform.loadTopmenu("standard")
```

The topmenu will be displayed in that form only.

This method is typically used in form initializers.

Related concepts

Syntax of a topmenu file (.4tm) on page 1446

A .4tm topmenu file is an XML file that holds a tree of elements defining a topmenu structure.

Creating the topmenu dynamically

Topmenus can be created at runtime by creating the corresponding XML representation in the AUI tree.

This example shows how to create a topmenu in all forms by using the default initialization function and the `om.DomNode` class:

```
CALL ui.Form.set_DefaultInitializer("myinit")
OPEN FORM f1 FROM "form1"
DISPLAY FORM f1
...
FUNCTION myinit(form)
  DEFINE form ui.Form
  DEFINE f om.DomNode
  LET f = form.getNode()
  ...
END FUNCTION
```

After getting the DOM node of the form, create a node with the "TopMenu" tag name:

```
DEFINE tm om.DomNode
LET tm = f.createChild("TopMenu")
```

For each Topmenu group, create a subnode with the "TopMenuGroup" tag name and set the attributes to define the group:

```
DEFINE tmg om.DomNode
LET tmg = tm.createChild("TopMenuGroup")
CALL tmg.setAttribute("text","Reports")
```

For each Topmenu option, create a sub-node in a group node with the "TopMenuCommand" tag name and set the attributes to define the option:

```
DEFINE tmi om.DomNode
LET tmi = tmg.createChild("TopMenuCommand")
CALL tmi.setAttribute("name","report")
CALL tmi.setAttribute("text","Order report")
CALL tmi.setAttribute("comment","Orders entered today")
CALL tmi.setAttribute("image","smiley")
```

If needed, you can create a "TopMenuSeparator" node inside a group, to separate menu options:

```
DEFINE tms om.DomNode
LET tms = tmg.createChild("TopMenuSeparator")
```
Related concepts

The DomNode class  on page 2584
The om.DomNode class provides methods to manipulate a DOM node of a data tree.

Topmenus on mobile devices

Topmenus can be used to implement a general options menu in mobile apps.

On mobile devices, actions render usually as default action views, that display implicitly in dedicated panes on the screen. When displaying forms on a mobile front-end, you can use a topmenu to get a list of options the end user can choose from.

Using topmenus for Android™ devices (GMA)

On Android devices, a TOPMENU renders as a menu icon on the top left of the screen, in the Android action bar.

When the user taps on this icon, a list with topmenu items shows up. Selecting an option fires the corresponding action handler is fired.

Note: On Android, the topmenu can only display one level of options (no tree of options is possible).
Table 362: TOPMENU rendering with GMA on Android

Using topmenus for iOS devices (GMI)

On iOS devices, a TOPMENU renders as a menu icon on the top left corner of the device screen, in the iOS navigation controller.

When the user taps on this icon, a new view appears with the first level of topmenu items. The user can the drill down to a next level, select an option if it's a leaf item, or tap on the back button to move one level up in the topmenu tree. Selecting a leaf item will fire the corresponding action handler and close the menu.
Table 363: TOPMENU rendering with GMI on iOS

Related concepts
Action views on mobile devices on page 1777
Action views are rendered following mobile specific standards.

Examples
Topmenu usage examples.

Example 1: Topmenu in XML format

```xml
<TopMenu>
  <TopMenuGroup text="Form" style="mystyle">
    <TopMenuCommand name="help" text="Help" image="quest" />
    <TopMenuCommand name="quit" text="Quit" acceleratorName="Alt-F4"/>
  </TopMenuGroup>
  <TopMenuGroup text="Edit">
    <TopMenuCommand name="accept" text="Validate" image="ok" />
    <TopMenuCommand name="cancel" text="Cancel" image="cancel" />
  </TopMenuGroup>
</TopMenu>
```
Example 2: Topmenu section in form file

```xml
<TOPMENU
  GROUP form (TEXT="Form", STYLE="mystyle")
    COMMAND help (TEXT="Help", IMAGE="quest")
    COMMAND quit (TEXT="Quit", ACCELERATOR=ALT-F4)
  END
  GROUP edit (TEXT="Edit")
    COMMAND accept (TEXT="Validate", IMAGE="ok")
    COMMAND cancel (TEXT="Cancel", IMAGE="cancel")
    SEPARATOR
    COMMAND editcut -- Gets decoration from action defaults
    COMMAND editcopy -- Gets decoration from action defaults
    COMMAND editpaste -- Gets decoration from action defaults
  END
  GROUP records (TEXT="Records")
    COMMAND append (TEXT="Add", IMAGE="add")
    COMMAND delete (TEXT="Remove", IMAGE="del")
    COMMAND update (TEXT="Modify", IMAGE="change")
    SEPARATOR
    COMMAND search (TEXT="Search", IMAGE="find")
  END
END
```

### Dialog instructions

This section describes the dialog instructions to control application forms and the concepts related to dialog implementation.

#### Static display (DISPLAY/ERROR/MESSAGE/CLEAR)

This section explains the instructions displaying static information to application forms, such as DISPLAY, ERROR, MESSAGE, CLEAR.

**Display of data and messages**

The values contained in program variables can be displayed to the current form with the DISPLAY BY NAME or DISPLAY TO instruction.

Forms can be cleared with the CLEAR FORM or CLEAR field-list instructions. Complete record lists (in SCROLLGRID, TABLE or TREE containers) can be cleared with the CLEAR SCREEN ARRAY instruction.

Application messages and warnings can be displayed to the user with the MESSAGE and ERROR instructions.

The DISPLAY BY NAME/TO instructions are not interactive, and are usually not needed if the program is always in the context of a dialog controlling the form fields. The data of the program variables will be displayed in form fields when the dialog starts, if the WITHOUT DEFAULTS option is specified, and during the dialog execution, form fields will be automatically synchronized with the program variables when using the UNBUFFERED option.
Related concepts

Windows and forms on page 1131
The section describes the concept of windows and forms in the language.

DISPLAY (to stdout)
The DISPLAY instruction displays text in line mode to the standard output channel.

Syntax

```
DISPLAY expression
```

1. `expression` is any expression supported by the language.

Usage

The DISPLAY instruction can be used to print information to the standard output channel (stdout) of the terminal the program is attached to.

The `expression` is typically a list of string constants and program variables separated by the `comma concatenation operator`.

Before displaying to the standard output channel, the expression is converted to a character string. The values contained in variables are formatted depending on the data types and environment settings such as `DBDATE` and `DBMONEY`.

Example

```
MAIN
  DISPLAY "Today's date is: ", TODAY
END MAIN
```

Related concepts

The Channel class on page 2412
The `base.Channel` class is a built-in class providing basic input/output functions.

MESSAGE

The MESSAGE instruction displays a message to the user.

Syntax

```
MESSAGE expression [, ...]
[ ATTRIBUTES ( display-attribute [, ...] ) ]
```

where `display-attribute` is:

```
BLACK | BLUE | CYAN | GREEN
MAGENTA | RED | WHITE | YELLOW
BOLD | DIM | INVISIBLE | NORMAL
REVERSE | BLINK | UNDERLINE
STYLE = "style-name"
```

1. `expression` is any expression supported by the language.
2. `style-name` is a presentation style name.

Usage

The MESSAGE instruction displays a message to the user in an interactive program.
In TUI mode, the text is displayed in the message line of the current window. The message line can be defined by the 
**OPTIONS MESSAGE LINE** instruction. Note that the default message line position is the same as the MENU option 
comment line.

In GUI mode, the text is displayed in a specific area, depending on the **STYLE** attribute. When you specify the 
**STYLE** attribute, you can reference a style defined in the presentation styles file. This allows you to display errors or 
messages in GUI mode with more sophisticated visual effects than the regular TTY attributes. Advanced automatic 
rendering can be obtained with message specific style attributes. If you want to apply a style automatically to all 
program messages displayed with the **MESSAGE** instruction, you can use the **:message** pseudo selector in the style 
definition.

**Example**

```
INPUT BY NAME custrec.* ...
  BEFORE INPUT
    MESSAGE "Enter customer data."
  ...
```

**Related concepts**

**Windows and forms** on page 1131
The section describes the concept of windows and forms in the language.

**Defining the position of reserved lines** on page 559
The **OPTIONS element** LINE defines position of dedicated screen lines.

**Text mode rendering (TUI mode)** on page 1109

**ERROR**
The **ERROR** instruction displays an error message to the user.

**Syntax**

```
ERROR expression
  ↓ ATTRIBUTES ( display-attribute ↓,...↓ ) ↓
```

where **display-attribute** is:

```
<table>
<thead>
<tr>
<th></th>
<th>BLACK</th>
<th>BLUE</th>
<th>CYAN</th>
<th>GREEN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MAGENTA</td>
<td>RED</td>
<td>WHITE</td>
<td>YELLOW</td>
</tr>
<tr>
<td></td>
<td>BOLD</td>
<td>DIM</td>
<td>INVISIBLE</td>
<td>NORMAL</td>
</tr>
<tr>
<td></td>
<td>REVERSE</td>
<td>BLINK</td>
<td>UNDERLINE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>STYLE = &quot;style-name&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

1. **expression** is any expression supported by the language.
2. **style-name** is a presentation style name.

**Usage**
The **ERROR** instruction displays an error message to the user in an interactive program.

In TUI mode, the error text is displayed in the error line of the screen. The text of the **ERROR** statement is always 
displayed independently to the current window. The error line can be defined by the **OPTIONS ERROR LINE** 
instruction.

In GUI mode, the text is displayed in a specific area, depending on the **STYLE** attribute. When you specify the 
**STYLE** attribute, you can reference a style defined in the presentation styles file. This allows you to display errors or 
messages in GUI mode with more sophisticated visual effects as the regular TTY attributes. Advanced automatic 
rendering can be obtained with message specific style attributes. If you want to apply a style automatically to all
program warnings displayed with the ERROR instruction, you can use the :error pseudo selector in the style definition.

Example

```... UPDATE tab1 SET col1 = ...
IF sqlca.sqlcode < 0 THEN
   ERROR SFMT("Row update failed (err=%1)", sqlca.sqlcode)
   ATTRIBUTES(STYLE="important")
   ...
END IF
...```

Related concepts

- Windows and forms on page 1131
  The section describes the concept of windows and forms in the language.
- Defining the position of reserved lines on page 559
  The OPTIONS element LINE defines position of dedicated screen lines.
- Text mode rendering (TUI mode) on page 1109

**DISPLAY TO**
The DISPLAY TO instruction displays data to form fields explicitly.

**Syntax**

```DISPLAY expression [,....] TO field-spec [,....] 
   [ ATTRIBUTES ( display-attribute [,....] ) ]```

where `field-spec` is:

- `field-name`
- `table-name.*`
- `table-name.field-name`
- `screen-array[line].*`
- `screen-array[line].field-name`
- `screen-record.*`
- `screen-record.field-name`
- `[,...]`

where `display-attribute` is:

- BLACK | BLUE | CYAN | GREEN
- MAGENTA | RED | WHITE | YELLOW
- BOLD | DIM | NORMAL
- REVERSE | BLINK | UNDERLINE

1. `expression` is any expression supported by the language.
2. `field-name` is the identifier of a field of the current form.
3. `table-name` is the identifier of a database table of the current form.
4. `screen-record` is the identifier of a screen record of the current form.
5. `screen-array` is the screen array that will be used in the form.

**Usage**

A DISPLAY TO statement copies the data from program variables to the form fields specified after the TO keyword.
When the program variables do not have the same names as the form fields, you must use the TO clause to explicitly map the variables to form fields using a screen record or screen array. You can list the fields individually, or you can use the `screen-record.*` or `screen-record[n].*` notation, where `screen-record[n].*` specifies all the fields in line `n` of a screen array.

In this example, the values in the `p_items` program record are displayed in the first row of the `s_items` screen array:

```plaintext
DISPLAY p_items.* TO s_items[1].*
```

The expanded list of screen fields must correspond in order and in number to the expanded list of identifiers after the DISPLAY keyword. Identifiers and their corresponding fields must have the same or compatible data types. For example, the next DISPLAY statement displays the values in the `p_customer` program record in fields of the `s_customer` screen record:

```plaintext
DISPLAY p_customer.* TO s_customer.*
```

For this example, the `p_customer` program record and the `s_customer` screen record require compatible declarations. The following DEFINE statement declares the `p_customer` program record:

```plaintext
DEFINE p_customer RECORD
    customer_num LIKE customer.customer_num,
    fname LIKE customer.fname,
    lname LIKE customer.lname,
    phone LIKE customer.phone
END RECORD
```

This fragment of a form specification declares the `s_customer` screen record:

```plaintext
ATTRIBUTES
    f000 = customer.customer_num;
    f001 = customer.fname;
    f002 = customer.lname;
    f003 = customer.phone;
END
```

The DISPLAY TO instruction is usually not needed if the program is always in the context of a dialog controlling the form fields.

**DISPLAY TO changes the touched flag**

The DISPLAY TO statement changes the modification flag of the target fields. When displaying a field value with DISPLAY TO, the FIELD_TOUCHED() operator returns true and the ON CHANGE and ON ROW CHANGE triggers may be invoked if the current field value was changed with a DISPLAY TO.

In dialogs controlling field input such as INPUT or INPUT ARRAY, use the UNBUFFERED attribute to display data to fields automatically without changing the 'touched' status of fields. The UNBUFFERED clause will perform automatic form field and program variable synchronization. When using the UNBUFFERED mode, the touched flag can be set with DIALOG.setFieldTouched() if you want to get the same effect as a DISPLAY TO.

**Specifying TTY attributes in the DISPLAY TO statement**

The ATTRIBUTES clause temporarily overrides any default display attributes or any attributes specified in the OPTIONS or OPEN WINDOW statements for the fields. When the DISPLAY TO statement completes execution, the default display attributes are restored. In a DISPLAY TO statement, any screen attributes specified in the ATTRIBUTES clause apply to all the fields that you specify after the TO keyword.

The REVERSE, BLINK, INVISIBLE, and UNDERLINE attributes are not sensitive to the color or monochrome status of the terminal, if the terminal is capable of displaying these intensity modes. The ATTRIBUTES clause can
include zero or more of the BLINK, REVERSE, and UNDERLINE attributes, and zero or one of the other attributes. That is, all of the attributes except BLINK, REVERSE, and UNDERLINE are mutually exclusive.

The DISPLAY TO statement ignores the INVISIBLE attribute, regardless of whether you specify it in the ATTRIBUTES clause.

Related concepts
DISPLAY BY NAME on page 1457
The DISPLAY BY NAME instruction displays data to form fields explicitly by name.

DISPLAY BY NAME
The DISPLAY BY NAME instruction displays data to form fields explicitly by name.

Syntax

```
DISPLAY BY NAME { variable | record.* } [ ..., ... ]
   [ ATTRIBUTES ( display-attribute [ ..., ... ] ) ]
```

where display-attribute is:

```
BLACK | BLUE | CYAN | GREEN
MAGENTA | RED | WHITE | YELLOW
BOLD | DIM | NORMAL
REVERSE | BLINK | UNDERLINE
```

1. variable is a program variable that has the same name as a form field.
2. record.* is a record variable that has members with the same names as form fields.

Usage

A DISPLAY BY NAME statement copies the data from program variables to the form fields associated to the variables by name. The program variables used in DISPLAY BY NAME must have the same name as the form fields where they have to be displayed. The language ignores any record structure name prefix when matching the names. The names must be unique and unambiguous; if not, the instruction raises an error.

For example, the following statement displays the values for the specified variables in the form fields with corresponding names (company and address1):

```
DISPLAY BY NAME p_customer.cust_company,
     p_customer.cust_address1
```

The DISPLAY BY NAME instruction is usually not needed if the program is always in the context of a dialog controlling the form fields.

DISPLAY BY NAME uses the default screen record

Unlike the DISPLAY TO instruction where you can explicitly specify a screen record or screen array, DISPLAY BY NAME displays data to the screen fields of the default screen records. The default screen records are those having the names of the tables defined in the TABLES section of the form specification file. When the form fields define a record list in the layout, only the first row can be referenced with the default screen record. In the next example, the form contains a static record list definition in the layout.

```
SCHEMA mystock
SCREEN
  { [ f01  f02  f03 ]
   [ f01  f02  f03 ]
   [ f01  f02  f03 ]
   [ f01  f02  f03 ]
  }
```
In the program, a DISPLAY BY NAME statement will display the data in the first line of the record list in the form:

```plaintext
DISPLAY BY NAME record_cust.*
```

**DISPLAY BY NAME changes the touched flag**

The DISPLAY BY NAME statement changes the modification flag of the target fields. When displaying a value with DISPLAY BY NAME, the `FIELD_TOUCHED()` operator returns true and the `ON CHANGE` and `ON ROW CHANGE` triggers may be invoked if the current field value was changed with a DISPLAY BY NAME.

In dialogs controlling field input such as `INPUT` or `INPUT ARRAY`, use the `UNBUFFERED` attribute to display data to fields automatically without changing the 'touched' status of fields. The UNBUFFERED clause will perform automatic form field and program variable synchronization. When using the UNBUFFERED mode, the touched flag can be set with `DIALOG.setFieldTouched()` if you want to get the same effect as a DISPLAY BY NAME statement.

**Specifying TTY attributes in the DISPLAY BY NAME statement**

The ATTRIBUTES clause temporarily overrides any default display attributes or any attributes specified in the OPTIONS or OPEN WINDOW statements for the fields. When the DISPLAY BY NAME statement completes execution, the default display attributes are restored.

The `REVERSE`, `BLINK`, `INVISIBLE`, and `UNDERLINE` attributes are not sensitive to the color or monochrome status of the terminal, if the terminal is capable of displaying these intensity modes. The ATTRIBUTES clause can include zero or more of the `BLINK`, `REVERSE`, and `UNDERLINE` attributes, and zero or one of the other attributes. That is, all of the attributes except `BLINK`, `REVERSE`, and `UNDERLINE` are mutually exclusive.

The DISPLAY BY NAME statement ignores the `INVISIBLE` attribute, regardless of whether you specify it in the ATTRIBUTES clause.

**Related concepts**

- **DISPLAY TO** on page 1455
  The DISPLAY TO instruction displays data to form fields explicitly.

- **CLEAR FORM**
  The CLEAR FORM instruction clears all fields in the current form.

**Syntax**

```plaintext
CLEAR FORM
```

**Usage**

The CLEAR FORM instruction clears all form fields of the current form. It has no effect on any part of the screen display except the form fields.

**Important:** Unlike CLEAR `field-list`, the CLEAR FORM instruction does not set the field modification flags.
In a similar way to `CLEAR field-list`, the `CLEAR FORM` instruction is typically used when the program is outside a dialog block execution controlling the form fields. For example, after a database query with a `CONSTRUCT` instruction, you might want to clear all search criteria entered by the user with this instruction, to cleanup the form.

**Note:** `CLEAR FORM` clears the field values and resets the TTY attributes to `NORMAL`.

The `CLEAR FORM` instruction is usually not needed, if the program is always in the context of a dialog controlling the form fields.

**Example**

```plaintext
CONSTRUCT BY NAME sql
   ON cust_name, cust_address, ...
   ...
END CONSTRUCT
CLEAR FORM
```

**Related concepts**

- **CLEAR WINDOW** on page 1136
  Clears the contents of a window.

- **CLEAR SCREEN** on page 1139
  Clears the complete application screen.

**CLEAR SCREEN ARRAY**
The `CLEAR SCREEN ARRAY` instruction clears the values of all rows of the form list identified by the specified screen array.

**Syntax**

```plaintext
CLEAR SCREEN ARRAY screen-array.*
```

1. `screen-array` is a screen array specified in the form.

**Usage**

After executing a `DISPLAY ARRAY` or `INPUT ARRAY` instruction, values remain in the form list identified by the screen array.

**Important:** The `CLEAR SCREEN ARRAY` instruction sets the field modification flags.

The `CLEAR SCREEN ARRAY` instruction automatically clears all rows of the list, regardless of the view: a `TABLE`, `TREE`, `SCROLLGRID`, or in a matrix of fields (an old-style/text-mode static screen array).

The `CLEAR SCREEN ARRAY` instruction replaces code which clears each individual row through the use of a loop:

```plaintext
-- Clearing each row individually
FOR i=1 TO <screen-array-length>
   CLEAR screen-array[i].*
END FOR
-- Unique instruction to clear a list
CLEAR SCREEN ARRAY screen-array.*
```

Using the `CLEAR SCREEN ARRAY` instruction eliminates the need for calculating the screen array length, a value which can change when using a `TABLE` container, that can be resized.

**Note:** `CLEAR SCREEN ARRAY` clears the field values and resets the TTY attributes to `NORMAL`.

The `CLEAR SCREEN ARRAY` instruction is usually not needed if the program is always in the context of a dialog controlling the form fields.
Example

```plaintext
... DISPLAY ARRAY cust_arr TO sa.*
    CLEAR SCREEN ARRAY sa.*
...```

Related concepts

**CLEAR FORM** on page 1458
The **CLEAR FORM** instruction clears all fields in the current form.

**CLEAR field-list** on page 1460
The **CLEAR field-list** instruction clears specific fields in the current form.

**CLEAR field-list**
The **CLEAR field-list** instruction clears specific fields in the current form.

Syntax

```plaintext
CLEAR field-list
```

where `field-list` is:

```plaintext
    | field-name
    | table-name.*
    | table-name.field-name
    | screen-array[line].*
    | screen-array[line].field-name
    | screen-record.*
    | screen-record.field-name
|,
|...
```

1. `field-name` is the identifier of a field of the current form.
2. `table-name` is the identifier of a database table of the current form.
3. `screen-record` is the identifier of a screen record of the current form.
4. `screen-array` is the screen array that will be used in the form.

Usage

The **CLEAR field-list** instruction can be used to clear the content of the specified form fields.

**Important:** The **CLEAR field-list** instruction sets the field modification flags.

The fields to be cleared can be specified individually or by referencing a **screen record** or **screen array**, with the `.*` notation to specify all fields.

Similar to **CLEAR FORM**, the **CLEAR field-list** is typically used when the program is not inside a dialog block execution controlling the form fields. For example, after a database query with a **CONSTRUCT** instruction, you might want to clear all search criteria entered by the user with this instruction, to cleanup the form.

**Note:** **CLEAR field-list** clears the field values and resets the TTY attributes to **NORMAL**.

The **CLEAR field-list** instruction is usually not needed if the program is always in the context of a dialog controlling the form fields.

Example

```plaintext
CONSTRUCT BY NAME sql
    ON s_customer.*
```
Related concepts

CLEAR FORM on page 1458
The CLEAR FORM instruction clears all fields in the current form.

CLEAR SCREEN ARRAY on page 1459
The CLEAR SCREEN ARRAY instruction clears the values of all rows of the form list identified by the specified screen array.

SCROLL
The SCROLL instruction moves data rows up or down in a screen array.

Syntax

```
SCROLL field-list { UP | DOWN } [ BY lines ]
```

where field-list is:

```
[ field-name
  table-name.*
  table-name.field-name
  screen-array[line].*
  screen-array[line].field-name
  screen-record.*
  screen-record.field-name
  [, ....]
```

1. field-name is the identifier of a field of the current form.
2. table-name is the identifier of a database table of the current form.
3. screen-record is the identifier of a screen record of the current form.
4. screen-array is the name of the screen array used of the current form.
5. lines is an integer expression that specifies how far (in lines) to scroll the display.

Usage

The SCROLL instruction specifies vertical movements of displayed values in all or some of the fields of a screen array within the current form.

The fields to be scrolled can be specified individually or by referencing a screen record or screen array, with the . * notation to specify all fields.

The SCROLL instruction is supported for applications running in TUI mode, to scroll screen array rows when no interactive instruction is executing. In a GUI application, use a TABLE container with a DISPLAY ARRAY instruction.

Related concepts

Text mode rendering (TUI mode) on page 1109
Form fields on page 1242

Form fields are form elements designed for data input and/or data display.

Windows and forms on page 1131
The section describes the concept of windows and forms in the language.

**Prompt for values (PROMPT)**

The PROMPT instruction provides unique field input in an automatic pop-up window.

**Understanding the PROMPT instruction**

The PROMPT instruction is used to query for a single value from the user.

PROMPT requires the text of the question to be displayed to the user and the variable that receives the value entered by the user. The variable can be of any simple data type except TEXT and BYTE.

The runtime system displays the question in the prompt area, waits for the user to enter a value, reads whatever value was entered until the user validates (for example with the Enter key), and stores this value in a response variable. The prompt dialog remains visible until the user enters a response.

The prompt finishes after ON IDLE, ON ACTION, or ON KEY block execution (to ensure backward compatibility).

**Prompt display in TUI mode**

In TUI mode, the PROMPT question and input field is displayed in the prompt line of the current window, which is defined by the OPTIONS PROMPT LINE instruction or with the ATTRIBUTES clause of OPEN WINDOW.

**Note:** If the prompt line is not as wide as the prompt string and the size required for the variable input, runtime error -1146 occurs.

**Prompt display in GUI mode**

In GUI mode, the PROMPT instruction opens a modal window with an OK and a Cancel button, and waits for input from the user.

**Figure 77: PROMPT window (Linux/Gnome GDC)**

![PROMPT window](Linux/Gnome GDC)

**Related concepts**

- Defining the position of reserved lines on page 559
  The OPTIONS element LINE defines position of dedicated screen lines.
- Dialog programming basics on page 1706
  This section describes basic dialog programming concepts.

**Syntax of PROMPT instruction**

The PROMPT statement assigns a user-supplied value to a variable.

**Syntax**

```
PROMPT question
  [ ATTRIBUTES ( display-attribute [, ...] ) ]
  FOR [ CHARACTER ] variable
  [ HELP number ]
  [ ATTRIBUTES ( control-attribute [, ...] ) ]
  [ dialog-control-block
    [ , ... ]
```
where dialog-control-block is one of:

```
| ON IDLE seconds
| ON TIMER seconds
| ON ACTION action-name
| ON KEY (key-name [,...])

statement
```

where display-attribute is:

```
| BLACK | BLUE | CYAN | GREEN
| MAGENTA | RED | WHITE | YELLOW
| BOLD | DIM | INVISIBLE | NORMAL
| REVERSE | BLINK | UNDERLINE
```

where control-attribute is:

```
| ACCEPT [ = boolean ]
| CANCEL [ = boolean ]
| CENTURY = "century-spec"
| FORMAT = "format-spec"
| PICTURE = "picture-spec"
| SHIFT = { "up" | "down" }
| HELP = help-number
| UNBUFFERED [ = boolean ]
| WITHOUT DEFAULTS [ = boolean ]
```

1. `question` is a string expression displayed as a message for the input of the value.
2. `variable` is the name of the variable that receives the data typed by the user.
3. The `FOR CHAR` clause exits the prompt statement when the first character has been typed.
4. `number` is the help message number to be displayed when the user presses the help key.
5. `key-name` is an hot-key identifier (such as `F11` or `Control-z`).
6. `action-name` identifies an action that can be executed by the user.
7. `seconds` is an integer literal or variable that defines a number of seconds.
8. `statement` is an instruction that is executed when the user presses the key defined by `key-name`.
9. `century-spec` is a string specifying the century input rule, like the `CENTURY` attribute.
10. `format-spec` is a string defining the display format for the prompt field, like the `FORMAT` attribute.
11. `picture-spec` is a string defining the input format for the prompt field, like the `PICTURE` attribute.

**PROMPT programming steps**

To use the `PROMPT` statement, you must:

1. Declare a program variable with the `DEFINE` statement.
2. Set the `INT_FLAG` variable to `FALSE`.
3. Define the `PROMPT` dialog, with dialog control blocks to control the instruction. Use the `FOR CHAR` clause if a single character is to be entered.
4. After executing the `PROMPT`, check the `INT_FLAG` variable to determine whether the input was validated or canceled by the user.

**Related concepts**

Example 2: Simple PROMPT with Interrupt Checking on page 1467
Using simple prompt inputs
Dialog coding concepts, configuration and code structure.

PROMPT instruction configuration

HELP option
The HELP clause specifies the number of a help message to display if the user invokes the help while executing the instruction. The predefined help action is automatically created by the runtime system. You can bind action views to the help action.

The HELP clause overrides the HELP attribute.

ACCEPT option
The ACCEPT attribute can be set to FALSE to avoid the automatic creation of the accept default action.

CANCEL option
The CANCEL attribute can be set to FALSE to avoid the automatic creation of the cancel default action. This is useful for example when you only need a validation action (accept), or when you want to write a specific cancellation procedure, by using EXIT INPUT.

If the CANCEL=FALSE option is set, no close action will be created, and you must write an ON ACTION close control block to create an explicit action.

Related concepts
Syntax of PROMPT instruction on page 1462
The PROMPT statement assigns a user-supplied value to a variable.

Default actions in PROMPT
When a PROMPT instruction executes, the runtime system creates a set of default actions.

Depending on the invoked default action, field validation occurs and different PROMPT control blocks are executed.

This table lists the default actions created for this dialog:

Table 364: Default actions created for the PROMPT dialog

<table>
<thead>
<tr>
<th>Default action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>accept</td>
<td>Validates the PROMPT dialog (validates field criteria) <em>Creation can be avoided with the ACCEPT attribute.</em></td>
</tr>
<tr>
<td>cancel</td>
<td>Cancels the PROMPT dialog (no validation, INT_FLAG is set) <em>Creation can be avoided with the CANCEL attribute.</em></td>
</tr>
<tr>
<td>close</td>
<td>By default, cancels the PROMPT dialog (no validation, INT_FLAG is set) Default action view is hidden. See Implementing the close action on page 1772.</td>
</tr>
<tr>
<td>help</td>
<td>Shows the help topic defined by the HELP clause. <em>Only created when a HELP clause is defined.</em></td>
</tr>
</tbody>
</table>
**Related concepts**

**Dialog programming basics** on page 1706
This section describes basic dialog programming concepts.

**Interaction blocks**

**ON ACTION block**

You can use **ON ACTION** blocks to execute a sequence of instructions when the user raises a specific action. This is the preferred solution compared to **ON KEY** blocks, because **ON ACTION** blocks use abstract names to control user interaction.

**Important:** The **PROMPT** instruction is automatically finished after **ON IDLE, ON ACTION, or ON KEY** block execution.

**Related concepts**

**Dialog actions** on page 1739
Describes how to program action handling when the end user triggers an action on the front-end.

**ON IDLE block**

The **ON IDLE seconds** clause defines a set of instructions that must be executed after a given period of user inactivity. This interaction block can be used, for example, to quit the dialog after the user has not interacted with the program for a specified period of time.

Do not mix **ON TIMER** and **ON IDLE** clauses.

As **ON IDLE** can fire field input validation, it is therefore not recommended in dialogs allowing input.

The parameter of **ON IDLE** must be an integer literal or variable. If the value is zero, the dialog timeout is disabled.

It is not recommended to use the **ON IDLE** trigger with a short timeout period such as 1 or 2 seconds; The purpose of this trigger is to give the control back to the program after a relatively long period of inactivity (10, 30 or 60 seconds). This is typically the case when the end user leaves the workstation, or gets a phone call. The program can then execute some code before the user gets the control back.

```
ON IDLE 30
  IF ask_question("Do you want to reload information from the database?") THEN
    -- Fetch data back from the db server
  END IF
```

**Important:** The timeout value is taken into account when the dialog initializes its internal data structures. If you use a program variable instead of an integer constant, any change of the variable will have no effect if the variable is changed after the dialog has initialized. If you want to change the value of the timeout variable, it must be done before the dialog block.

**Related concepts**

**Get program control if user is inactive** on page 1712
Execute some code after a given number of seconds, when the user does not interact with the program.

**ON TIMER block** on page 1466

**ON KEY block**

An **ON KEY (key-name)** block defines an action with a hidden action view (no default button is visible), that executes a sequence of instructions when the user presses the specified key.

**Note:** The **ON KEY** block is supported for backward compatibility with TUI mode applications. In new developments, consider using **ON ACTION** with accelerators defined in action defaults.

An **ON KEY** block can specify up to four accelerator keys. Each key creates a specific action object that will be identified by the key name in lowercase.
For example, \texttt{ON KEY(F5,F6)} creates two actions with the names \texttt{f5} and \texttt{f6}. Each action object will get an \texttt{acceleratorName} attribute assigned, with the corresponding accelerator name. The specified keys must be one of the \texttt{virtual keys}.

\textbf{Note:} The \texttt{KEY()} clause allows a comma-separated list of keys. Up to four keys can be specified. For new developments, consider using a single key, or prefer \texttt{ON ACTION} handlers with a single accelerator definition in action defaults.

In GUI mode, action defaults are applied for \texttt{ON KEY} actions by using the name of the action (the key name). You can define secondary accelerator keys, as well as default decoration attributes like button text and image, by using the key name as action identifier. The action name is always in lowercase letters.

Check carefully \texttt{ON KEY CONTROL-?} statements to avoid having duplicate accelerators for multiple actions due to the accelerators defined by action defaults. Additionally, \texttt{ON KEY} statements used with \texttt{ESC, TAB, UP, DOWN, LEFT, RIGHT, HELP, NEXT, PREVIOUS, INSERT, CONTROL-M, CONTROL-X, CONTROL-V, CONTROL-C and CONTROL-A} should be avoided for use in GUI programs, because it's very likely to clash with default accelerators defined in the factory action defaults file provided by default.

By default, \texttt{ON KEY} actions are not decorated with a default button in the action frame (the default action view). You can show the default button by configuring a \texttt{text} attribute with the action defaults.

\begin{verbatim}
ON KEY (CONTROL-Z)
    CALL open_zoom()
\end{verbatim}

\textbf{Related concepts}

- \texttt{Configuring actions} on page 1744
  Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with \textit{action attributes}.

- \texttt{Default action views} on page 1743
  A default action view is created to render an action handler when no explicit action view exists for it.

\texttt{ON TIMER block}

The \texttt{ON TIMER seconds} clause defines a set of instructions that must be executed at regular intervals. This interaction block can be used, for example, to check if a message has arrived in a queue, and needs to be processed.

Do not mix \texttt{ON TIMER} and \texttt{ON IDLE} clauses.

As \texttt{ON TIMER} can fire field input validation, it is therefore not recommended in dialogs allowing input.

The parameter of \texttt{ON TIMER} must be an integer literal or variable. If the value is zero, the dialog timeout is disabled.

It is not recommended to use the \texttt{ON TIMER} trigger with a short timeout period, such as 1 or 2 seconds. The purpose of this trigger is to give the control back to the program after a reasonable period of time, such as 10, 20 or 60 seconds.

\begin{verbatim}
ON TIMER 30
    CALL check_for_messages()
\end{verbatim}

\textbf{Important:} The timer value is taken into account when the dialog initializes its internal data structures. If you use a program variable instead of an integer constant, a change of the variable has no effect if the change takes place after the dialog has initialized. If you want to change the value of the timeout variable, it must be done before the dialog block.

\textbf{Related concepts}

- \texttt{Get program control on a regular (timed) basis} on page 1713
  Execute some code after a given number of seconds, with or without user interaction with the program.

- \texttt{ON IDLE block} on page 1465
Examples
PROMPT dialog examples.

Example 1: Simple PROMPT statements

```plaintext
MAIN
  DEFINE birth DATE
  DEFINE chkey CHAR(1)
  PROMPT "Please enter your birthday: " FOR birth
  DISPLAY "Your birthday is: " || birth
  PROMPT "Now press a key... " FOR CHAR chkey
  DISPLAY "You pressed: " || chkey
END MAIN
```

Example 2: Simple PROMPT with Interrupt Checking

```plaintext
MAIN
  DEFINE birth DATE
  LET INT_FLAG = FALSE
  PROMPT "Please enter your birthday: " FOR birth
  IF INT_FLAG THEN
    DISPLAY "Interrupt received."
  ELSE
    DISPLAY "Your birthday is: " || birth
  END IF
END MAIN
```

Example 3: PROMPT with ATTRIBUTES and ON ACTION handlers

```plaintext
MAIN
  DEFINE birth DATE
  LET birth = TODAY
  PROMPT "Please enter your birthday: " FOR birth
  ATTRIBUTES(WITHOUT DEFAULTS)
    ON ACTION action1
      DISPLAY "Action 1"
  END PROMPT
  DISPLAY "Your birthday is " || birth
END MAIN
```

Ring menus (MENU)

The MENU instruction implements a list of options the end user can choose from.

Understanding ring menus

The MENU instruction implements a set of choices, also known as action handlers.

A MENU dialog defines a list of options that can trigger actions to execute associated program code. Ring menus are implemented with the MENU interactive instruction.

A MENU block lists the possible actions that can be triggered in a given place in the program, with the associated program code to be executed.

```plaintext
MENU "Sample"
  COMMAND "Say hello"
    DISPLAY "Hello, world!"
  COMMAND "Exit"
  EXIT MENU
END MENU
```

A ring menu can only define a set of options for a given level of the program. You cannot define all menu options of your program in a single MENU instruction; you must implement nested menus.
The MENU instruction is mainly designed for text mode applications, displaying ring menus at the top of the screen. A typical TUI mode application starts with a global menu, defining general options to access subroutines, which in turn implement specific menus with database record handling options such as 'Append', 'Delete', 'Modify', and 'Search'.

Ring menus can also be used in a GUI application. However, as this instruction does not handle form fields, other parts of the form are disabled during the menu dialog execution. In GUI applications, ring menus are typically used to open a modal window with Yes / No / Cancel options, where options are typically defined with the ON ACTION clauses, to keep the code abstract and define decoration in form or with attributes:

```plaintext
MENU ""
  ON ACTION print ATTRIBUTES(TEXT=\"print\",IMAGE=\"printer\")
      CALL print_document()
  ON ACTION save -- Get decoration from form or action defaults
      CALL save_document()
  ON ACTION close -- Binds to GUI window close button
      EXIT PROGRAM
END MENU
```

Related concepts
Action handling basics on page 1739
This topic describes the basics of action views, action events, and action handlers.

The model-view-controller paradigm on page 1706
The dynamic user interface architecture is based on the Model-View-Controller (MVC) paradigm.

Syntax of the MENU instruction
The MENU instruction defines a set of options the end user can select to trigger actions in a program.

Syntax

```plaintext
MENU [title] 
  \| ATTRIBUTES ( menu-attribute [,...] ) \|
  \| BEFORE MENU \|
      menu-statement \|
      [...] \|
  \| menu-option \|
      [...] \|
END MENU
```

where menu-option is one of:

```plaintext
\| COMMAND option-name \|
      [option-comment] [ HELP help-number ] 
      menu-statement \|
\| COMMAND KEY ( key-name ) option-name \|
      [option-comment] [ HELP help-number ] 
      menu-statement \|
\| COMMAND KEY ( key-name ) \|
      menu-statement \|
\| ON ACTION action-name \|
      [ ATTRIBUTES ( action-attributes-menu ) ] 
      menu-statement \|
\| ON IDLE seconds \|
      menu-statement \|
\| ON TIMER seconds
```
1. title is a string expression defining the title of the menu.
2. menu-attribute is an attribute that defines the behavior and presentation of the menu.
3. key-name is a hot-key identifier (like F11 or Control-z).
4. option-name is a string expression defining the label of the menu option and identifying the action that can be executed by the user.
5. option-comment is a string expression containing a description for the menu option, displayed when option-name is the current.
6. help-number is an integer that allows you to associate a help message number with the menu option.
7. action-name identifies an action that can be executed by the user.
8. seconds is an integer literal or variable that defines a number of seconds.
9. action-name identifies an action that can be executed by the user.
10. action-attributes are dialog-specific action attributes.

**MENU programming steps**

Follow this procedure to use the MENU dialog instruction.

To implement a MENU statement:

1. Create a MENU block with a title and write the end of the menu block with the END MENU keywords.
2. Depending on the type of menu rendering you need, add an ATTRIBUTES clause with the required STYLE attribute.
3. List all the options that you want to offer to the end user when the menu executes. Typical CRUD programs will implement "Append", "Modify", "Delete" operations for a given database application entity (customers, orders, items tables). Typical dialog box menus have "Yes" / "No" / "Cancel" options.
4. Whether TUI or GUI mode, define action views (TOPMENU, TOOLBAR or form BUTTON) for each menu action, and use either COMMAND [KEY] or ON ACTION clauses to define the menu options.
5. When the menu is not a pop-up or dialog menu, do not forget to implement an option to leave the menu with the EXIT MENU control instruction.

6. Implement the code to be executed in every option.

Using ring menus
Dialog coding concepts, configuration and code structure.

Rendering modes of a menu

Purpose of MENU styles
When you add a style to a MENU's attributes list, you define the look-and-feel of that menu and how that menu acts.

MENU rendering specification
The rendering mode of a MENU instruction can be controlled with the STYLE dialog attribute:

```plaintext
MENU "Test" ATTRIBUTES ( STYLE = "mode" )
...
END MENU
```

Note: MENU ... ATTRIBUTES(STYLE="mode") is not a presentation style defined in a 4st file: It defines a display mode, a rendering hint for front-ends.

The decoration of the different rendering modes of a MENU depends on the front-end type and the platform used. Consider testing the menu instruction with all front-ends that you intend to support for end users.

Default MENU rendering
By default, if no STYLE attribute is used in the MENU instruction, each menu option will be displayed as a push BUTTON in a dedicated area of the current window, depending on the front-end. This dedicated area is called the action frame.

Note that when an explicit action view (for example, a button in form layout) is associated with a menu option, the default button will not appear in the action frame area.

The default rendering of a MENU, including the position of the action frame in the window, can be controlled with window presentation style attributes.

```plaintext
MAIN
    MENU "File"
        COMMAND "New"
            DISPLAY "New"
        COMMAND "Open"
            DISPLAY "Open"
        COMMAND "Save"
            DISPLAY "Save"
        COMMAND "Import"
            DISPLAY "Import"
        COMMAND "Quit"
            EXIT MENU
    END MENU
END MAIN
```
Modal dialog MENU rendering

Menus can be rendered in a modal dialog window by specifying the STYLE="dialog" attribute in the MENU instruction.

```
MAIN
  MENU "Example of dialog menu"
    ATTRIBUTES ( STYLE="dialog", COMMENT="Delete the file?" )
    COMMAND "Yes"
      DISPLAY "Yes"
    COMMAND "No"
      DISPLAY "No"
    COMMAND "Cancel"
      DISPLAY "Cancel"
  END MENU
END MAIN
```

When the user clicks on an option, the MENU instruction automatically exits and the modal dialog window closes. There is no need for an EXIT MENU command.

With STYLE="dialog", when the user clicks on an option, the MENU instruction automatically exits and the pop-up menu closes. There is no need for an EXIT MENU command.

Figure 79: MENU displayed as a modal box (Linux/Gnome GDC)
**Pop-up MENU rendering**

Menus can also be displayed as pop-up choice lists, when the `STYLE="popup"` attribute is used in the `MENU` instruction.

```plaintext
MAIN
  DEFINE r INTEGER
  MENU "test"
      COMMAND "popup"
      DISPLAY popup()
      COMMAND "quit"
      EXIT MENU
  END MENU
END MAIN

FUNCTION popup()
  DEFINE r INTEGER
  LET r = -1
  MENU "unused" ATTRIBUTES ( STYLE="popup" )
      COMMAND "Copy all"
      LET r = 1
      COMMAND "Copy current"
      LET r = 2
      COMMAND "Paste all"
      LET r = 3
      COMMAND "Paste current"
      LET r = 4
  END MENU
  RETURN r
END FUNCTION
```

With `STYLE="popup"`, when the user clicks on an option, the `MENU` instruction automatically exits and the pop-up menu closes. There is no need for an `EXIT MENU` command.

![Screenshot of a pop-up menu](image)

**Figure 80: MENU displayed as pop-up pick list (Linux/Gnome GDC)**

**MENU rendering on mobile platforms**

On mobile devices, the rendering of the `MENU` dialog depends on whether or not the current window has a form.
If a `MENU` is active and the current Window has no form, then the `MENU` is shown as a list of actions.

If a `MENU` is active and has a `FORM`, then the menu actions are rendered like all other dialog actions (see Mobile programming chapter for more details).

For example, when executing the following code with and without a form as program argument:

```plaintext
MAIN
  IF num_args()==1 THEN
    OPEN FORM f1 FROM arg_val(1)
    DISPLAY FORM f1
  END IF
  MENU "Farbee logistics"
    COMMAND "Orders" "Enter orders"
    COMMAND "Customers" "Manage customer information"
    COMMAND "Stock" "Check stock status"
    COMMAND "Setting" "Change app settings"
    COMMAND "Quit"
  EXIT MENU
END MENU
END MAIN
```

You get the following rendering in Genero Mobile for Android. On the left side (without form), the `MENU` shows up as a list of options that can be selected. On the right (with form), the `MENU` options render as default action views in the Android™ action bar.
Binding action views to menu options

A MENU statement is a controller for user actions, defining action handlers triggered by action views. Actions views in the form file (such as TOOLBAR buttons, TOPMENU items or push BUTTON) are bound to menu options by name. For example, if a MENU instruction defines ON ACTION sendmail, a form button with the name "sendmail" will be attached to that action handler.

When binding action views to menu option clauses, the action name is case sensitive. The compiler converts COMMAND labels and ON ACTION identifiers to lowercase to create the action name. It is recommended that you use all lowercase letters when defining the action name for action views and menu options.

Menu options can also be defined with the COMMAND clause. Unlike ON ACTION, the COMMAND clause takes a string literal as argument, that defines both the action name and the default text to be displayed in the default action view. For example, COMMAND "Help" will define the action name help and the default button text "Help". Action views must be bound with the action name in lowercase (help).

When the menu is rendered as a pop-up of dialog box, no explicit action views need to be defined, default action views will be created and will get the decoration specified in action defaults.
Related concepts

Configuring actions on page 1744
Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with action attributes.

MENU instruction configuration

The rendering and behavior of a MENU instruction can be configured with the ATTRIBUTES clause:

```plaintext
MENU "Question"
   ATTRIBUTES ( 
      STYLE="dialog",
      COMMENT="Do you want to commit your changes?"
   )
```

When the STYLE dialog attribute is set to 'default' or when you do not specify the menu type, the runtime system generates a default decoration as a set of buttons in a specific area of the current window.

When the STYLE attribute is set to 'dialog', the menu options appear as buttons at the bottom in a temporary modal window, in which you can define the message and the icon with the COMMENT and IMAGE attributes.

When the STYLE is set to 'popup', the menu appears as a pop-up menu (a context menu).

If the menu STYLE is "dialog" or "popup", the dialog is automatically exited after any action clause such as ON ACTION, COMMAND, ON IDLE or ON TIMER.

Related concepts

Rendering modes of a menu on page 1470
Syntax of the MENU instruction on page 1468

The MENU instruction defines a set of options the end user can select to trigger actions in a program.

Default actions in MENU

When an MENU instruction executes, the runtime system creates a set of default actions.

Table 366: Default actions created for the MENU instruction

<table>
<thead>
<tr>
<th>Default action</th>
<th>Control Block execution order</th>
</tr>
</thead>
</table>
| close          | Created to execute COMMAND KEY (INTERRUPT) if used (can be overwritten with ON ACTION close)  
                 Default action view is hidden. See Implementing the close action on page 1772. |
| help           | Shows the help topic defined by the HELP clause.  
                 Default action view is hidden. |

Window close events can be trapped with COMMAND KEY (INTERRUPT) clause.

Related concepts

The model-view-controller paradigm on page 1706
The dynamic user interface architecture is based on the Model-View-Controller (MVC) paradigm.

MENU control blocks

BEFORE MENU block

If the MENU block contains a BEFORE MENU clause, statements within this clause are executed before the menu dialog starts.
This block is typically used to hide or disable some menu options depending on the current context of the program. For example, when the current user is not allowed to create new records, the menu options can be disabled as follows:

```
MENU "Orders"
    BEFORE MENU
      CALL DIALOG.setActionActive("append", can_user_append() )
      ...
      COMMAND "Append" -- creates "append" action (lowercase)
      ...
      ...
END MENU
```

In TUI mode, the menu options can also be disabled, but they will still be displayed on the screen. The end user will see the option, but cannot select it. In this case it's more convenient to hide the option from the end user with the `DIALOG.setActionHidden()` method, instead of disabling the action.

**Related concepts**

The Dialog class on page 2512

The `ui.Dialog` class provides a set of methods to configure, query and control the current interactive instruction.

**MENU interaction blocks**

`COMMAND [KEY()] "option" block`

The `COMMAND [KEY(key-name)] "option-name"` clause defines a menu action handler with a set of instructions to be executed when an action is invoked. The option text (`option-name`), converted to lowercase letters, defines the name of the action.

For example, when defining:

```
COMMAND "Hello"
```

The name of the action will be "hello" (not "Hello" with a capital H).

When used with the `KEY()` clause, the command specifies both accelerator keys and an option text.

**Note:** The `KEY()` clause allows a comma-separated list of keys. Up to four keys can be specified. For new developments, consider using a single key, or prefer `ON ACTION` handlers with a single accelerator definition in action defaults.

Action defaults will be applied by using the action name defined by the option text (converted to lowercase).

In TUI mode, actions created with `COMMAND [KEY]` do not get accelerators from action defaults; only actions defined with `ON ACTION` will get accelerators of action defaults.

Explicit action views defined in the form (`BUTTON` in layout, `TOPMENU` or `TOOLBAR` items) will get all action defaults associated to the menu command, while default action views (i.e. buttons in the action frame) will be decorated with the menu option text and comment specified in the program (this means that the `TEXT` and `COMMENT` attributes of the corresponding action defaults entry are not used for the default action views). However, other attributes such as the `IMAGE` will also be applied to default action views.

For example, when defining:

```
COMMAND "Hello" "This is the Hello option"
```

The name of the action will become "hello", the default action view button text will be "Hello", and the button hint will be "This is the Hello option", even if an action default defines a different text or comment for the "hello" action. If the corresponding action default defines an `IMAGE` icon, it will display in the default action view button.

**Note:** The keys defined with the `KEY()` clause will take precedence over accelerators defined with action defaults corresponding to the action name.
The first letter of the display text of a COMMAND menu clause can be used as default accelerator, if no other accelerator is defined by a KEY() clause or by action defaults:

- When the menu option is not rendered as a default view and is bound to an action view like a toolbar button, the first letter of the COMMAND option will define the default accelerator, except if several menu COMMAND options start with the same letter, or if the letter is used as accelerator by another command.

- When the menu option is rendered as a default view, and the first letter of a menu COMMAND option is not used by other menu option labels, pressing the key corresponding to that letter will execute that action. If several menu COMMAND options start with the same letter and the user presses that key, the front-end shifts the focus from button to button. To fire the action, the user must press ENTER.

When using an ampersand (&) in the command name, some front-ends consider the letter following & as an Alt-key accelerator, and the letter will be underscored. The ampersand character forms part of the action name. For example, COMMAND "&Save" will create an action with the name "&save".

In a MENU instruction, the alternative to COMMAND [KEY] "option-name" can be the ON ACTION action-name clause, to write abstract code without having the decoration (option name, comment and accelerators) in the program code.

**Related concepts**

Configuring actions on page 1744

Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with *action attributes.*

Action handling basics on page 1739

This topic describes the basics of action views, action events, and action handlers.

**COMMAND KEY() block**

The COMMAND KEY(key-name) block (without an option text) defines a menu action handler with a set of instructions to be executed when an action is invoked.

The KEY() clause defines one to four accelerator keys separated by a comma. The specified key name must be one of the virtual keys.

**Note:** The KEY() clause allows a comma-separated list of keys. Up to four keys can be specified. For new developments, consider using a single key, or prefer ON ACTION handlers with a single accelerator definition in action defaults.

While a COMMAND KEY(key-name) "option-name" (with option text) defines the name of the action with the option text (converted to lowercase), a COMMAND KEY(key-name) (without option text), defines the action name from the last key in the KEY() list, converted to lowercase letters. For example, with COMMAND KEY(F10,F12,Control-Z), the name of the action will be "control-z".

Action defaults will be applied by using the key name of the KEY() clause. With a list of keys, the last key name will be used to apply action defaults, because it defines the action name.

**Note:** The keys defined with the KEY() clause will take precedence over accelerators defined with action defaults corresponding to the action name.

By default, COMMAND KEY(key-name) actions are not decorated with a default action (this means that a button in the action frame will not appear for these actions). However, by defining the TEXT attribute within action defaults, the default action view button will be visible. This allows you to decorate existing COMMAND KEY(key-name) clauses with graphical buttons without changing the program code.

To write abstract code without decoration in your programs, use the ON ACTION clause instead of COMMAND [KEY], except if the action view must get the focus.

In TUI mode, actions created with COMMAND [KEY] do not get accelerators from action defaults; Only actions defined with ON ACTION will get accelerators of action defaults.

**Related concepts**

Configuring actions on page 1744
Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with action attributes.

**Action handling basics** on page 1739

This topic describes the basics of action views, action events, and action handlers.

**ON ACTION block**

The **ON ACTION action-name** blocks execute a sequence of instructions when the user triggers a specific action.

A typical action handler block looks like this:

```plaintext
ON ACTION action-name
  instruction
...
```

Action blocks are bound by name to action views (like buttons) in the current form. Action views can be BUTTON, TOOLBAR buttons, or TOPMENU options, and if no explicit action view is defined, actions are rendered with a default action view, depending on the type of front-end.

This example defines an action block to open a typical zoom window and let the user select a customer record:

```plaintext
ON ACTION zoom
  CALL zoom_customers() RETURNING st, rec.cust_id, rec.cust_name
```

In a dialog handling user input such as INPUT, INPUT ARRAY and CONSTRUCT, if an action is specific to a field, add the **INFIELD** clause to have the action automatically enabled when the corresponding field gets the focus:

```plaintext
ON ACTION zoom INFIELD cust_city
  CALL zoom_cities() RETURN st, rec.cust_city
```

In most cases actions are decorated with action defaults in form files, but there can be cases where the **ON ACTION** handler needs to define its own attributes at the program level. This can be done by adding the **ATTRIBUTES()** clause of **ON ACTION**:

```plaintext
ON ACTION custinfo ATTRIBUTES(DISCLOSUREINDICATOR, IMAGE="info")
  CALL show_customer_info()
```

For more details about action handlers, and action configuration, see **Dialog actions** on page 1739.

**Related concepts**

**Configuring actions** on page 1744

Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with action attributes.

**Action handling basics** on page 1739

This topic describes the basics of action views, action events, and action handlers.

**ON IDLE block**

The **ON IDLE seconds** clause defines a set of instructions that must be executed after a given period of user inactivity. This interaction block can be used, for example, to quit the dialog after the user has not interacted with the program for a specified period of time.

Do not mix **ON TIMER** and **ON IDLE** clauses.

As **ON IDLE** can fire field input validation, it is therefore not recommended in dialogs allowing input.

The parameter of **ON IDLE** must be an integer literal or variable. If the value is zero, the dialog timeout is disabled.

It is not recommended to use the **ON IDLE** trigger with a short timeout period such as 1 or 2 seconds; The purpose of this trigger is to give the control back to the program after a relatively long period of inactivity (10, 30 or 60
This is typically the case when the end user leaves the workstation, or gets a phone call. The program can then execute some code before the user gets the control back.

```
ON IDLE 30
  IF ask_question("Do you want to reload information from the database?") THEN
    -- Fetch data back from the db server
  END IF
```

**Important:** The timeout value is taken into account when the dialog initializes its internal data structures. If you use a program variable instead of an integer constant, any change of the variable will have no effect if the variable is changed after the dialog has initialized. If you want to change the value of the timeout variable, it must be done before the dialog block.

**Related concepts**
- Get program control if user is inactive on page 1712
- Execute some code after a given number of seconds, when the user does not interact with the program.

**ON TIMER block** on page 1466

**ON TIMER block**
The `ON TIMER seconds` clause defines a set of instructions that must be executed at regular intervals. This interaction block can be used, for example, to check if a message has arrived in a queue, and needs to be processed.

Do not mix `ON TIMER` and `ON IDLE` clauses.

As `ON TIMER` can fire field input validation, it is therefore not recommended in dialogs allowing input.

The parameter of `ON TIMER` must be an integer literal or variable. If the value is zero, the dialog timeout is disabled.

It is not recommended to use the `ON TIMER` trigger with a short timeout period, such as 1 or 2 seconds. The purpose of this trigger is to give the control back to the program after a reasonable period of time, such as 10, 20 or 60 seconds.

```
ON TIMER 30
  CALL check_for_messages()
```

**Important:** The timer value is taken into account when the dialog initializes its internal data structures. If you use a program variable instead of an integer constant, a change of the variable has no effect if the change takes place after the dialog has initialized. If you want to change the value of the timeout variable, it must be done before the dialog block.

**Related concepts**
- Get program control on a regular (timed) basis on page 1713
- Execute some code after a given number of seconds, with or without user interaction with the program.

**ON IDLE block** on page 1465

**MENU control instructions**

**SHOW/HIDE OPTION instruction**

**Syntax:**

```
\{ HIDE \} \{ SHOW \} \{ OPTION \} \{ ALL \} \{ option-name \, ... \}
```
Usage

The `SHOW OPTION` instruction will show/enable action views corresponding to the listed menu options. The default action views (buttons in action frame) are made visible and the explicit action views (`BUTTON` in form) are enabled. The `HIDE OPTION` instruction will hide default action views and disable explicit action views.

Use the `ALL` keyword reference all menu options. In a menu that contains many options, you typically do a `HIDE OPTIONS ALL` followed by `HIDE OPTION` to show a subset of the menu options.

The `SHOW OPTION` and `HIDE OPTION` instructions are provided for backward compatibility. To hide and show default action views, use the `DIALOG.setActionHidden()` method instead. In GUI applications, it is recommended that you disable actions, instead of hiding them from the end user.

Example

```
MENU "Customers"
   BEFORE MENU
       HIDE OPTION ALL
       SHOW OPTION "Add", "Exit"
   ...
```

`EXIT MENU instruction`

**Syntax**

```
EXIT MENU
```

**Usage**

`EXIT MENU` statement terminates the `MENU` block and continues the program flow with the statement after the menu block.

**Example**

```
MENU "Stock"
   ...
   COMMAND "Exit"
       EXIT MENU
END MENU
```

**Related concepts**

`EXIT block-name` on page 383

The `EXIT block-name` instruction transfers control out of the current program block.

`CONTINUE MENU instruction`

**Syntax**

```
CONTINUE MENU
```

**Usage**

The `CONTINUE MENU` instruction resumes the execution of a classical `MENU` dialog.

`CONTINUE MENU` skips the instructions remaining in the current code block (that belongs to the current dialog), redisplays the menu options, and gives the control back to the user.

**Example**

```
MENU "Stock"
```
COMMAND "Exit"
  IF question("Exit the program?")==FALSE THEN
    CONTINUE MENU
  END IF
  CALL commit_changes()
  EXIT MENU
END MENU

Related concepts
CONTINUE block-name on page 382
The CONTINUE block-name instruction resumes execution of a loop or dialog statement.

Examples
MENU dialog examples.
Example 1: MENU with abstract action options

MAIN
  MENU
    ON ACTION new
      CALL newFile()
    ON ACTION open
      CALL openFile()
    ON ACTION save
      CALL saveFile()
    ON ACTION import
      LOAD FROM "infile.dat" INSERT INTO table
    ON ACTION quit
      EXIT PROGRAM
  END MENU
END MAIN

Example 2: MENU with text-mode options

MAIN
  MENU "File"
    COMMAND KEY ( CONTROL-N ) "New" "Creates New File" HELP 101
      CALL newFile()
    COMMAND KEY ( CONTROL-O ) "Open" "Open existing File" HELP 102
      CALL openFile()
    COMMAND KEY ( CONTROL-S ) "Save" "Save Current File" HELP 103
      CALL saveFile()
    COMMAND "Import"
      LOAD FROM "infile.dat" INSERT INTO table
    COMMAND KEY ( CONTROL-Q ) "Quit" "Quit Program" HELP 201
      EXIT PROGRAM
  END MENU
END MAIN

Example 3: MENU with STYLE="dialog"

This code example implements typical message box utility functions implemented with MENU dialogs:

FUNCTION mbox_ynnc(title,msg)
  DEFINE title, msg STRING
  DEFINE res SMALLINT
  MENU title ATTRIBUTES (STYLE="dialog",COMMENT=msg)
    ON ACTION yes LET res = 1
    ON ACTION no LET res = 0
    ON ACTION cancel LET res = -1
  END MENU
  RETURN res
Record input (INPUT)

The INPUT instruction provides single record input control in an application form.

Understanding the INPUT instruction
The INPUT instruction controls a single record input from form fields.

The INPUT statement binds program variables to screen-records for data entry in form fields. The INPUT statement uses the current form in the current window. Before executing the INPUT statement, record data must be fetched from the database table into the program variables using the input statement.

During the INPUT statement execution, the user can edit the record fields, while the program controls the behavior of the instruction with control blocks.

When the user moves from field to field or changes values, dialog control blocks such as BEFORE FIELD are executed.

When the user clicks on an action view (button), or when an asynchronous event occurs, dialog interaction blocks like ON ACTION are executed.

The code inside an INPUT dialog can use control instructions, dialog control functions, and the ui.Dialog class, to implement the dialog behavior.

To terminate the INPUT execution, the user can validate (or cancel) the dialog to commit (or invalidate) the modifications made in the record.

When the statement completes execution, the form is deactivated. After the user terminates the input (for example, with the "accept" key), the program must test the INT_FLAG variable to check if the dialog was validated (or canceled), and then can use the INSERT or UPDATE SQL statements to modify the appropriate database tables.

Related concepts
Dialog programming basics on page 1706
This section describes basic dialog programming concepts.

Syntax of the INPUT instruction
The INPUT statement supports data entry in fields of the current form.

Syntax

```plaintext
INPUT { BY NAME { variable | record.* } [,]...
      WITHOUT DEFAULTS ]$
      { variable | record.* } [,]...
      WITHOUT DEFAULTS ]$
      FROM field-list
```
where \textit{dialog-control-block} is one of:

\begin{itemize}
\item \texttt{BEFORE INPUT}
\item \texttt{AFTER INPUT}
\item \texttt{BEFORE FIELD field-spec [,...]}  
\item \texttt{AFTER FIELD field-spec [,...]}  
\item \texttt{ON CHANGE field-spec [,...]}  
\item \texttt{ON IDLE seconds}
\item \texttt{ON TIMER seconds}
\item \texttt{ON ACTION action-name}
\item \texttt{INFIELD field-spec}
\item \texttt{ATTRIBUTES ( action-attributes-input )}
\item \texttt{ON KEY ( key-name [,...])}
\end{itemize}

\where \textit{action-attributes-input} is:

\begin{itemize}
\item \texttt{TEXT = string}
\item \texttt{COMMENT = string}
\item \texttt{IMAGE = string}
\item \texttt{ACCELERATOR = string}
\item \texttt{DEFAULTVIEW = \{ YES \| NO \| AUTO \}}
\item \texttt{VALIDATE = NO}
\item \texttt{CONTEXTMENU = \{ YES \| NO \| AUTO \}}
\end{itemize}

\where \textit{dialog-statement} is one of:

\begin{itemize}
\item \texttt{statement}
\item \texttt{ACCEPT INPUT}
\item \texttt{CONTINUE INPUT}
\item \texttt{EXIT INPUT}
\item \texttt{NEXT FIELD}
\item \texttt{CURRENT}
\item \texttt{NEXT}
\item \texttt{PREVIOUS}
\item \texttt{field-spec}
\end{itemize}

\where \textit{field-list} defines a list of fields with one or more of:

\begin{itemize}
\item \texttt{field-name}
\item \texttt{table-name.*}
\item \texttt{table-name.field-name}
\item \texttt{screen-array[line].*}
\item \texttt{screen-array[line].field-name}
\item \texttt{screen-record.*}
\item \texttt{screen-record.field-name}
\end{itemize}
where field-spec identifies a unique field with one of:

```
field-name
| table-name.field-name
| screen-array.field-name
| screen-record.field-name
```

where display-attribute is:

```
BLACK | BLUE | CYAN | GREEN
| MAGENTA | RED | WHITE | YELLOW
| BOLD | DIM | INVISIBLE | NORMAL
| REVERSE | BLINK | UNDERLINE
```

where control-attribute is:

```
ACCEPT [ = boolean ]
| CANCEL [ = boolean ]
| FIELD ORDER FORM
| HELP = help-number
| NAME = "dialog-name"
| UNBUFFERED [ = boolean ]
| WITHOUT DEFAULTS [ = boolean ]
```

1. variable is a program variable that will be filled by the INPUT statement.
2. record.* is a record variable that will be filled by the INPUT statement.
3. help-number is an integer that allows you to associate a help message number with the instruction.
4. field-name is the identifier of a field of the current form.
5. table-name is the identifier of a database table of the current form.
6. screen-record is the identifier of a screen record of the current form.
7. screen-array is the screen array that will be used in the form.
8. line is a screen array line in the form.
9. key-name is a hot-key identifier (like F11 or Control-z).
10. dialog-name is the identifier of the dialog.
11. seconds is an integer literal or variable that defines a number of seconds.
12. action-name identifies an action that can be executed by the user.
13. statement is any instruction supported by the language.
14. boolean is a boolean expression evaluated when the dialog starts.
15. action-attributes are dialog-specific action attributes.

**INPUT programming steps**

Follow this procedure to use the INPUT dialog instruction.

To implement the INPUT statement:

1. Create a form specification file, with an optional screen record.
   The screen record identifies the presentation elements to be used by the runtime system to display the records.
   If you omit the declaration of the screen record in the form file, the runtime system will use the default screen records created by the form compiler for each table listed in the TABLES section and for the FORMONLY pseudo-table.

2. Make sure that the program controls interruption handling with **DEFER INTERRUPT**, to manage the validation/cancellation of the interactive dialog.
3. Define a program RECORD with the DEFINE instruction.
   The members of the program record must correspond to the elements of the screen record, by number and data
   types.
4. Open and display the form, using OPEN WINDOW WITH FORM or the OPEN FORM/DISPLAY FORM
   instructions.
5. If needed, fill the program record with data, for example, with a result set cursor.
6. Set the INT_FLAG variable to FALSE.
7. Implement the INPUT dialog block to handle data input.
8. Inside the INPUT statement, control the behavior of the instruction with BEFORE INPUT, BEFORE FIELD,
    AFTER FIELD, AFTER INPUT and ON ACTION blocks.
9. After the interaction statement block, test the INT_FLAG predefined variable to check if the dialog was canceled
    (INT_FLAG=TRUE) or validated (INT_FLAG=FALSE).
    If the INT_FLAG variable is TRUE, you should reset it to FALSE so as not to disturb code that relies on this
    variable to detect interruption events from the GUI front-end or TUI console.

Related concepts
Form specification files on page 1237
Form specification files are the source files defining the layout and content of application forms.

Using simple record inputs
Dialog coding concepts, configuration and code structure.
Variable binding in INPUT
The INPUT instruction binds program variables (typically, members of a RECORD) to the fields of a screen
record of the current form, and synchronizes the data between field input buffers and program variables.

Binding variables and fields by name
The INPUT BY NAME variable-list instruction implicitly binds the fields to the program variables that have
the same identifiers as the form field names. The program variables are typically defined within a record declared
with a LIKE table.* based on a database schema, so as to get the same names as the form fields defined with the
database column references. The runtime system ignores any record name prefix when making the match, only record
member names matter. The unqualified names of the variables and of the fields must be unique and unambiguous
within their respective domains. If they are not, the runtime system generates an exception.

```
SCHEMA stock
DEFINE custrec RECORD LIKE customer.*
...
INPUT BY NAME custrec.*
...
END INPUT
```

Binding variables and fields by position
The INPUT variable-list FROM field-list clause explicitly binds the variables to form fields by
position. The form can include other fields that are not part of the specified variable list, but the number of variables
or record members must equal the number of form fields listed in the FROM clause. Each variable must be of the same
(or a compatible) data type as the corresponding form field. When the user enters data, the runtime system checks the
entered value against the data type of the variable, not the data type of the form field.

```
SCHEMA stock
DEFINE custrec RECORD LIKE customer.*, comment VARCHAR(100)
...
INPUT custrec.*, comment FROM sr_cust.*, cmt
...
```
When using the FROM clause with a screen record followed by a .* (dot star), keep in mind that program variables are bound to screen record fields by position, so you must make sure that the program variables are defined (or listed) in the same order as the screen array fields.

**Serial column support**

The program variables can be of any data type. The runtime system will adapt input and display rules to the variable type. If a variable is declared with the LIKE clause and uses a column defined as SERIAL / SERIAL8 / BIGSERIAL, the runtime system will treat the field as if it was defined with the NOENTRY attribute in the form file. Since values of serial columns are automatically generated by the database server, no user input is required for such fields.

**The UNBUFFERED mode**

The variables act as data model to display data or to get user input through the INPUT instruction. Always use the variables if you want to change some field values by program. When using the UNBUFFERED attribute, the instruction is sensitive to program variable changes. If you need to display new data during the INPUT execution, just assign the values to the program variables; the runtime system will automatically display the values to the screen:

```
INPUT p_items.* FROM s_items.* ATTRIBUTES ( UNBUFFERED )
ON CHANGE code
    IF p_items.code = "A34" THEN
        LET p_items.desc = "Item A34"
    END IF
END INPUT
```

**Handling default field values**

When the INPUT instruction executes, any column default values are displayed in the screen fields, unless you specify the WITHOUT DEFAULTS keywords. The column default values are specified in the form specification file with the DEFAULT attribute, or in the database schema files.

If you specify the WITHOUT DEFAULTS option. However, the form fields display the current values of the variables when the INPUT statement begins. This option is available with both the BY NAME and the FROM binding clauses.

```
LET p_items.code = "A34"
INPUT p_items.* FROM s_items.* WITHOUT DEFAULTS
BEFORE INPUT
    MESSAGE "it is recommended that you see A34 in field 'code'..."
END INPUT
```

**Using PHANTOM fields**

If the program record has the same structure as a database table (this is the case when the record is defined with a LIKE clause), you may not want to display/use some of the columns. You can achieve this by using PHANTOM fields in the screen record definition. Phantom fields will only be used to bind program variables, and will not be transmitted to the front-end for display.

**Related concepts**

- [Binding variables to form fields](#) on page 1715

Some dialogs need program variables to store form field values.

**INPUT instruction configuration**

This section describes the options that can be specified in the ATTRIBUTES clause of the INPUT instruction. The options of the ATTRIBUTES clause override all default attributes and temporarily override any display attributes
that the OPTIONS or the OPEN WINDOW statement specified for these fields. With the INPUT statement, the INVISIBLE attribute is ignored.

NAME option
The NAME attribute can be used to name the INPUT dialog. This attribute is used to identify the INPUT dialog.

HELP option
The HELP clause specifies the number of the help message to display if the user invokes the help while the focus is in any field used by the instruction. The predefined 'help' action is automatically created by the runtime system. You can bind action views to the 'help' action.
The HELP clause overrides the HELP attribute.

WITHOUT DEFAULTS option
Indicates if the fields controlled by INPUT must be filled (FALSE) or not (TRUE) with the column default values defined in the form specification file or the database schema files. The runtime system assumes that the field satisfies the REQUIRED attribute when WITHOUT DEFAULTS is used. If the WITHOUT DEFAULT option is not used, all fields defined with the REQUIRED attribute must be visited and modified. Fields not defined as NOT NULL can be left empty.

FIELD ORDER FORM option
By default, the tabbing order is defined by the variable binding list in the instruction description. You can control the tabbing order by using the FIELD ORDER FORM attribute. When this attribute is used, the tabbing order is defined by the TABINDEX attribute of the form fields. If this attribute is used, the Dialog.fieldOrder FGLPROFILE entry is ignored.
The OPTIONS instruction can also change the behavior of the INPUT instruction, with the INPUT WRAP or FIELD ORDER FORM options.

UNBUFFERED option
Indicates that the dialog must be sensitive to program variable changes. When using this option, you bypass the traditional BUFFERED mode.

When using the traditional "buffered" mode, program variable changes are not automatically displayed to form fields; You need to execute a DISPLAY TO or DISPLAY BY NAME. Additionally, if an action is triggered, the value of the current field is not validated and is not copied into the corresponding program variable. The only way to get the text of the current field is to use GET_FLDBUF().

If the "unbuffered" mode is used, program variables and form fields are automatically synchronized. You don't need to display explicitly values with a DISPLAY TO or DISPLAY BY NAME. When an action is triggered, the value of the current field is validated and is copied into the corresponding program variable.

ACCEPT option
The ACCEPT attribute can be set to FALSE to avoid the automatic creation of the accept default action. This option can be used for example when you want to write a specific validation procedure, by using ACCEPT INPUT.

CANCEL option
The CANCEL attribute can be set to FALSE to avoid the automatic creation of the cancel default action. This is useful for example when you only need a validation action (accept), or when you want to write a specific cancellation procedure, by using EXIT INPUT.

If the CANCEL=FALSE option is set, no close action will be created, and you must write an ON ACTION close control block to create an explicit action.
Related concepts
Syntax of the INPUT instruction on page 1482
The INPUT statement supports data entry in fields of the current form.

Default actions in INPUT
When an INPUT instruction executes, the runtime system creates a set of default actions.
Depending on the invoked default action, field validation occurs and different INPUT control blocks are executed.
This table lists the default actions created for this dialog:

Table 367: Default actions created for the INPUT dialog

<table>
<thead>
<tr>
<th>Default action</th>
<th>Description</th>
</tr>
</thead>
</table>
| accept         | Validates the INPUT dialog (validates fields and leaves the dialog)  
Creation can be avoided with ACCEPT attribute. |
| cancel         | Cancels the INPUT dialog (no validation, INT_FLAG is set to TRUE)  
Creation can be avoided with CANCEL attribute. |
| close          | By default, cancels the INPUT dialog (no validation, INT_FLAG is set to TRUE) 
Default action view is hidden. See Implementing the close action on page 1772. |
| help           | Shows the help topic defined by the HELP clause.  
Only created when a HELP clause is defined. |

The accept and cancel default actions can be avoided with the ACCEPT and CANCEL dialog control attributes:

```
INPUT BY NAME field1 ATTRIBUTES ( CANCEL=FALSE )
...
```

Related concepts
Dialog programming basics on page 1706
This section describes basic dialog programming concepts.

INPUT control blocks
INPUT control blocks execution order
This table shows the order in which the runtime system executes the control blocks in the INPUT instruction, depending on the user action:

Table 368: Control Block Execution Order for INPUT

<table>
<thead>
<tr>
<th>Context / User action</th>
<th>Control Block execution order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entering the dialog</td>
<td>1. BEFORE INPUT</td>
</tr>
<tr>
<td></td>
<td>2. BEFORE FIELD (first field)</td>
</tr>
</tbody>
</table>
### Context / User action

<table>
<thead>
<tr>
<th>Context / User action</th>
<th>Control Block execution order</th>
</tr>
</thead>
</table>
| Moving from field A to field B | 1. **ON CHANGE** (if value has changed for field A)  
2. **AFTER FIELD** (for field A)  
3. **BEFORE FIELD** (for field B) |
| Changing the value of a field with a specific field like checkbox | 1. **ON CHANGE** |
| Validating the dialog | 1. **ON CHANGE** (if value has changed in current field)  
2. **AFTER FIELD**  
3. **AFTER INPUT** |
| Canceling the dialog | 1. **AFTER INPUT** |

### BEFORE INPUT block

**BEFORE INPUT block in singular and parallel INPUT, INPUT ARRAY dialogs**

In a singular INPUT, INPUT ARRAY instruction, or when used as parallel dialog, the **BEFORE INPUT** is only executed once when the dialog is started.

The **BEFORE INPUT** block is executed once at dialog startup, before the runtime system gives control to the user. This block can be used to display messages to the user, initialize program variables and set up the dialog instance by deactivating unused fields or actions the user is not allowed to execute.

```
INPUT BY NAME cust_rec.* ...
BEFORE INPUT
   MESSAGE "Input customer information"
   CALL DIALOG.setActionActive("check_info", is_super_user() )
   CALL DIALOG.setFieldActive("cust_comment", is_super_user() )
...
```

The fields are initialized with the defaults values before the **BEFORE INPUT** block is executed. When the INPUT instruction uses the **WITHOUT DEFAULTS** option, the default values are taken from the program variables bound to the fields, otherwise (with defaults), the **DEFAULT** attributes of the form fields are used.

Use the **NEXT FIELD** control instruction in the **BEFORE INPUT** block, to jump to a specific field when the dialog starts.

**BEFORE INPUT block in INPUT and INPUT ARRAY of procedural DIALOG**

In an INPUT or INPUT ARRAY sub-dialog of a procedural DIALOG instruction, the **BEFORE INPUT** block is executed when the focus goes to a group of fields driven by the sub-dialog. This trigger is only invoked if a field of the sub-dialog gets the focus, and none of the other fields had the focus.

When the focus is in a list driven by an INPUT ARRAY sub-dialog, moving to a different row will not invoke the **BEFORE INPUT** block.

**BEFORE INPUT** is executed after the **BEFORE DIALOG** block and before the **BEFORE ROW**, **BEFORE FIELD** blocks.

In this example, the **BEFORE INPUT** block is used to set up a specific action and display a message:

```
INPUT BY NAME p_order.*
BEFORE INPUT
   CALL DIALOG.setActionActive("validate_order", TRUE)
```
Related concepts
BEFORE CONSTRUCT block on page 1573
BEFORE DISPLAY block on page 1509
AFTER INPUT block on page 1490

AFTER INPUT block

AFTER INPUT block in singular and parallel INPUT, INPUT ARRAY dialogs
In a singular INPUT, INPUT ARRAY instruction, or when used as parallel dialog, the AFTER INPUT is only executed once when dialog ends.

The AFTER INPUT block is executed after the user has validated or canceled the INPUT or INPUT ARRAY dialog with the accept or cancel default actions, or when the ACCEPT INPUT instruction is executed.

The AFTER INPUT block is not executed when the EXIT INPUT instruction is performed.

In singular and parallel dialogs, this block is typically used to implement global dialog validation rules for several fields. If the values entered by the user do not satisfy the constraints, use the NEXT FIELD instruction to force the dialog to continue. The CONTINUE INPUT instruction can be used instead of NEXT FIELD, when no particular field has to be selected.

Before checking the validation rules, make sure that the INT_FLAG variable is FALSE: because if the user cancels the dialog, the validation rules must be skipped.

INPUT BY NAME cust_rec.*
    WITHOUT DEFAULTS ATTRIBUTES ( UNBUFFERED )
    ...

AFTER INPUT
    IF NOT INT_FLAG THEN
        IF cust_rec.cust_address IS NOT NULL
            AND cust_rec.cust_zipcode IS NULL THEN
                ERROR "Address is incomplete, enter a zipcode."
                NEXT FIELD zipcode
            END IF
        END IF
    END IF
END INPUT

To limit the validation to fields that have been modified by the end user, you can call the FIELD_TOUCHED() function or the DIALOG.getFieldTouched() method to check if a field has changed during the dialog execution. This will make your validation code faster if the user has only modified a couple of fields in a large form.

AFTER INPUT block in INPUT and INPUT ARRAY of procedural DIALOG
In an INPUT or INPUT ARRAY sub-dialog of a procedural DIALOG instruction, the AFTER INPUT block is executed when the focus is lost by a group of fields driven by an INPUT or INPUT ARRAY sub-dialog. This trigger is invoked if a field of the sub-dialog loses the focus, and a field of a different sub-dialog gets the focus. When the focus is in a list driven by an INPUT ARRAY sub-dialog, moving to a different row will not invoke the AFTER INPUT block.

If the focus leaves the current group and goes to an action view, this trigger is not executed, because the focus has not gone to another sub-dialog yet.

AFTER INPUT is executed after the AFTER FIELD, AFTER ROW blocks and before the AFTER DIALOG block.

Executing a NEXT FIELD in the AFTER INPUT control block will keep the focus in the group of fields. Within an INPUT ARRAY sub-dialog, NEXT FIELD will keep the focus in the list and stay in the current row. You typically use this behavior to control user input.
In this example, the AFTER INPUT block is used to validate data and disable an action that can only be used in the current group:

```plaintext
INPUT BY NAME p_order.*
AFTER INPUT
  IF NOT check_order_data(DIALOG) THEN
    NEXT FIELD CURRENT
  END IF
CALL DIALOG.setFieldActive("validate_order", FALSE)
```

Related concepts
AFTER DISPLAY block on page 1510
AFTER CONSTRUCT block on page 1574
BEFORE INPUT block on page 1489

BEFORE FIELD block

In dialog instructions INPUT, INPUT ARRAY, CONSTRUCT or in a DISPLAY ARRAY using the FOCUSONFIELD attribute, the BEFORE FIELD block is executed every time the specified field gets the focus.

For single record inputs driven by INPUT or query by example (QBEs) driven by CONSTRUCT, the BEFORE FIELD block is executed when moving the focus from field to field.

For editable lists driven by INPUT ARRAY, the BEFORE FIELD block is executed when moving the focus from field to field.

For record lists driven by DISPLAY ARRAY using the FOCUSONFIELD attribute, the BEFORE FIELD block is executed when moving the focus from field to field. However, the fields will not be editable as in an INPUT ARRAY.

**Important:** The BEFORE FIELD block is also executed when performing a NEXT FIELD instruction.

The BEFORE FIELD keywords must be followed by a list of form field specification. The screen-record name can be omitted.

BEFORE FIELD is executed after BEFORE INPUT, BEFORE CONSTRUCT, BEFORE ROW and BEFORE INSERT.

Use this block to do some field value initialization, or to display a message to the user:

```plaintext
INPUT BY NAME p_cust.* ...
BEFORE FIELD cust_status
  LET p_cust.cust_comment = NULL
  MESSAGE "Enter customer status"
```

When using the default FIELD ORDER CONSTRAINT mode, the dialog executes the BEFORE FIELD block of the field corresponding to the first variable of an INPUT or INPUT ARRAY, even if that field is not editable (NOENTRY, hidden or disabled). The block is executed when you enter the dialog and every time you create a new row in the case of INPUT ARRAY. This behavior is supported for backward compatibility. The block is not executed when using the FIELD ORDER FORM mode, the mode recommended for DIALOG instructions.

With the FIELD ORDER FORM mode, for each dialog executing for the first time with a specific form, the BEFORE FIELD block will be invoked for the first field of the initial tabbing list defined by the form, even if that field was hidden or moved around in a table. The dialog then behaves as if a NEXT FIELD first-visible-column had been executed in the BEFORE FIELD of that field.

When form-level validation occurs and a field contains an invalid value, the dialog gives the focus to the field, but no BEFORE FIELD trigger will be executed.

Related concepts
Form-level validation rules on page 1727
Form-level validation rules can be defined for each field controlled by a dialog.

AFTER FIELD block on page 1492
**ON CHANGE block**

The **ON CHANGE** block can be used to detect when a field has been changed by user input. The **ON CHANGE** block is executed, if the value has changed since the field got the focus and the modification flag is set.

The **ON CHANGE** block can be used in **INPUT**, **INPUT ARRAY** and **CONSTRUCT** dialogs.

For editable fields defined as **EDIT**, **TEXTEDIT** or **BUTTONEDIT**, the **ON CHANGE** block is executed when leaving a field, if the value of the specified field has changed since the field got the focus and the modification flag is set for the field. The field is left when user validates the dialog, when moving to another field, or when moving to another row in an **INPUT ARRAY**. However, if the text edit field is defined with the **COMPLETER** attribute to enable **autocompletion**, the **ON CHANGE** trigger will be fired after a short period of time, when the user has typed characters in.

For editable fields defined as **CHECKBOX**, **COMBOBOX**, **DATEEDIT**, **DATETIMEEDIT**, **TIMEEDIT**, **RADIOGROUP**, **SPINEDIT**, **SLIDER** or **URL-based WEBCOMPONENT** (when the **COMPONENTTYPE** attribute is not used), the **ON CHANGE** block is invoked immediately when the user changes the value with the widget edition feature. For example, when toggling the state of a **CHECKBOX**, when selecting an item in a **COMBOBOX** list, or when choosing a date in the calendar of a **DATEEDIT**. Note that for such item types, when **ON CHANGE** is fired, the modification flag is always set.

```plaintext
ON CHANGE order_checked -- Defined as CHECKBOX
CALL setup_dialog(DIALOG)
```

**Note:** If both an **ON CHANGE** block and **AFTER FIELD** block are defined for a field, the **ON CHANGE** block is executed before the **AFTER FIELD** block.

When changing by program the value of the current field in an **ON ACTION** block, the **ON CHANGE** block will be executed when leaving the field, if the value is different from the previous value and the modification flag is set (after previous user input or when the touched flag has been changed by program).

In an **INPUT** or **INPUT ARRAY**, the field value change is related to the value of the variable bound to the field. In a **CONSTRUCT** dialog, the field value change is related to the input buffer / displayed value.

**Note:** With a **NEXT FIELD** instruction, the state of the field value change is reset, as if the user had left and reentered the field. When using **NEXT FIELD** in a **ON CHANGE** block or in an **ON ACTION** block, the **ON CHANGE** block will only be re-executed, if the value has changed since **NEXT FIELD**, and the modification flag is set. Therefore, **ON CHANGE** should not be used for field validation with a **NEXT FIELD**, because **ON CHANGE** will not get triggered again, if the (invalid) value has not changed; Field validation rules must be implemented in **AFTER FIELD** blocks and/or **AFTER INPUT** blocks.

**Related concepts**

- **Form item types** on page 1266
- **The form item types** defines the purpose of form elements.
- **AFTER FIELD block** on page 1492

**AFTER FIELD block**

In dialog instructions **INPUT**, **INPUT ARRAY**, **CONSTRUCT** or in a **DISPLAY ARRAY** using the **FOCUSONFIELD** attribute, the **AFTER FIELD** block is executed every time the focus leaves the specified field.

For single record inputs driven by **INPUT** or query by example (**QBEs**) driven by **CONSTRUCT**, the **AFTER FIELD** block is executed when moving the focus from field to field.

For editable lists driven by **INPUT ARRAY**, the **AFTER FIELD** block is executed when moving the focus from field to field in the same row, or when moving to another row in the same column.

For record lists driven by **DISPLAY ARRAY** using the **FOCUSONFIELD** attribute, the **AFTER FIELD** block is executed when moving the focus from field to field. However, the fields will not be editable as in an **INPUT ARRAY**.

The **AFTER FIELD** keywords must be followed by a list of form field specifications. The screen-record name can be omitted.
AFTER FIELD is executed before AFTER INSERT, ON ROW CHANGE, AFTER ROW, AFTER INPUT or AFTER CONSTRUCT.

When a NEXT FIELD instruction is executed in an AFTER FIELD block, the cursor moves to the specified field, which can be the current field. This can be used to prevent the user from moving to another field / row during data input. Note that the BEFORE FIELD block is also executed when NEXT FIELD is invoked.

The AFTER FIELD block of the current field is not executed when performing a NEXT FIELD; only BEFORE INPUT, BEFORE CONSTRUCT, BEFORE ROW, and BEFORE FIELD of the target item might be executed, depending on the sub-dialog type.

When ACCEPT DIALOG, ACCEPT INPUT, or ACCEPT CONSTRUCT is performed, the AFTER FIELD trigger of the current field is executed.

Use the AFTER FIELD block to implement field validation rules:

```
INPUT BY NAME p_item.* ...
AFTER FIELD item_quantity
   IF p_item.item_quantity <= 0 THEN
      ERROR "Item quantity cannot be negative or zero"
      LET p_item.item_quantity = 0
   NEXT FIELD item_quantity
END IF
```

Related concepts

ACCEPT DIALOG instruction on page 1640
ON CHANGE block on page 1492

INPUT interaction blocks

ON ACTION block

The ON ACTION action-name blocks execute a sequence of instructions when the user triggers a specific action. A typical action handler block looks like this:

```
ON ACTION action-name
   instruction
   ...
```

Action blocks are bound by name to action views (like buttons) in the current form. Action views can be BUTTON, TOOLBAR buttons, or TOPMENU options, and if no explicit action view is defined, actions are rendered with a default action view, depending on the type of front-end.

This example defines an action block to open a typical zoom window and let the user select a customer record:

```
ON ACTION zoom
   CALL zoom_customers() RETURNING st, rec.cust_id, rec.cust_name
```

In a dialog handling user input such as INPUT, INPUT ARRAY and CONSTRUCT, if an action is specific to a field, add the INFIELD clause to have the action automatically enabled when the corresponding field gets the focus:

```
ON ACTION zoom INFIELD cust_city
   CALL zoom_cities() RETURN st, rec.cust_city
```

In most cases actions are decorated with action defaults in form files, but there can be cases where the ON ACTION handler needs to define its own attributes at the program level. This can be done by adding the ATTRIBUTES() clause of ON ACTION:

```
ON ACTION custinfo ATTRIBUTES(DISCLOSEINDICATOR, IMAGE="info")
   CALL show_customer_info()
```
For more details about action handlers, and action configuration, see Dialog actions on page 1739.

**Related concepts**

- Configuring actions on page 1744
- Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with *action attributes*.
- Action handling basics on page 1739
  This topic describes the basics of action views, action events, and action handlers.

**ON IDLE block**

The **ON IDLE** clause defines a set of instructions that must be executed after a given period of user inactivity. This interaction block can be used, for example, to quit the dialog after the user has not interacted with the program for a specified period of time.

Do not mix **ON TIMER** and **ON IDLE** clauses.

As **ON IDLE** can fire field input validation, it is therefore not recommended in dialogs allowing input.

The parameter of **ON IDLE** must be an integer literal or variable. If the value is zero, the dialog timeout is disabled.

It is not recommended to use the **ON IDLE** trigger with a short timeout period such as 1 or 2 seconds; The purpose of this trigger is to give the control back to the program after a relatively long period of inactivity (10, 30 or 60 seconds). This is typically the case when the end user leaves the workstation, or gets a phone call. The program can then execute some code before the user gets the control back.

```plaintext
ON IDLE 30
IF ask_question("Do you want to reload information from the database?") THEN
  -- Fetch data back from the db server
END IF
```

**Important:** The timeout value is taken into account when the dialog initializes its internal data structures. If you use a program variable instead of an integer constant, any change of the variable will have no effect if the variable is changed after the dialog has initialized. If you want to change the value of the timeout variable, it must be done before the dialog block.

**Related concepts**

- Get program control if user is inactive on page 1712
  Execute some code after a given number of seconds, when the user does not interact with the program.

**ON TIMER block** on page 1466

**ON KEY block**

An **ON KEY** block defines an action with a hidden action view (no default button is visible), that executes a sequence of instructions when the user presses the specified key.

**Note:** The **ON KEY** block is supported for backward compatibility with TUI mode applications. In new developments, consider using **ON ACTION** with accelerators defined in action defaults.

An **ON KEY** block can specify up to four accelerator keys. Each key creates a specific action object that will be identified by the key name in lowercase.

For example, **ON KEY(F5,F6)** creates two actions with the names `f5` and `f6`. Each action object will get an `acceleratorName` attribute assigned, with the corresponding accelerator name. The specified keys must be one of the virtual keys.

**Note:** The `KEY()` clause allows a comma-separated list of keys. Up to four keys can be specified. For new developments, consider using a single key, or prefer **ON ACTION** handlers with a single accelerator definition in action defaults.
In GUI mode, action defaults are applied for `ON KEY` actions by using the name of the action (the key name). You can define secondary accelerator keys, as well as default decoration attributes like button text and image, by using the key name as action identifier. The action name is always in lowercase letters.

Check carefully `ON KEY CONTROL-?` statements to avoid having duplicate accelerators for multiple actions due to the accelerators defined by action defaults. Additionally, `ON KEY` statements used with `ESC`, `TAB`, `UP`, `DOWN`, `LEFT`, `RIGHT`, `HELP`, `NEXT`, `PREVIOUS`, `INSERT`, `CONTROL-M`, `CONTROL-X`, `CONTROL-V`, `CONTROL-C` and `CONTROL-A` should be avoided for use in GUI programs, because it's very likely to clash with default accelerators defined in the factory action defaults file provided by default.

By default, `ON KEY` actions are not decorated with a default button in the action frame (the default action view). You can show the default button by configuring a `text` attribute with the action defaults.

```
ON KEY (CONTROL-Z)
    CALL open_zoom()
```

**Related concepts**

- **Configuring actions** on page 1744
  Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with `action attributes`.

- **Default action views** on page 1743
  A default action view is created to render an action handler when no explicit action view exists for it.

**ON TIMER block**

The `ON TIMER seconds` clause defines a set of instructions that must be executed at regular intervals. This interaction block can be used, for example, to check if a message has arrived in a queue, and needs to be processed.

Do not mix `ON TIMER` and `ON IDLE` clauses.

As `ON TIMER` can fire field input validation, it is therefore not recommended in dialogs allowing input.

The parameter of `ON TIMER` must be an integer literal or variable. If the value is zero, the dialog timeout is disabled.

It is not recommended to use the `ON TIMER` trigger with a short timeout period, such as 1 or 2 seconds. The purpose of this trigger is to give the control back to the program after a reasonable period of time, such as 10, 20 or 60 seconds.

```
ON TIMER 30
    CALL check_for_messages()
```

**Important:** The timer value is taken into account when the dialog initializes its internal data structures. If you use a program variable instead of an integer constant, a change of the variable has no effect if the change takes place after the dialog has initialized. If you want to change the value of the timeout variable, it must be done before the dialog block.

**Related concepts**

- **Get program control on a regular (timed) basis** on page 1713
  Execute some code after a given number of seconds, with or without user interaction with the program.

- **ON IDLE block** on page 1465

**INPUT control instructions**

**ACCEPT INPUT instruction**

The `ACCEPT INPUT` instruction validates the `INPUT` instruction and exits the dialog block if no error is raised.

The `AFTER FIELD`, `ON CHANGE`, etc. control blocks will be executed.

The statements after the `ACCEPT INPUT` instruction will not be executed.

```
INPUT BY NAME cust_rec.*
    ...
    ON ACTION process_order
```
The `INPUT` instruction creates the default accept action to let the user validate the dialog. Use of the `ACCEPT INPUT` instruction is recommended only in specific cases when the default accept action is not appropriated.

**Related concepts**
- [EXIT INPUT instruction](#) on page 1496
- [CONTINUE INPUT instruction](#) on page 1496

**Related reference**
- [INPUT control blocks execution order](#) on page 1488

**CONTINUE INPUT instruction**

CONTINUE INPUT skips all subsequent statements in the current control block and gives the control back to the dialog. This instruction is useful when program control is nested within multiple conditional statements, and you want to return the control to the dialog.

If this instruction is called in a control block that is not `AFTER INPUT`, further control blocks might be executed according to the context. CONTINUE INPUT instructs the dialog to continue as if the code in the control block was terminated (it's a kind of `GOTO end_of_control_block`). However, when executed in `AFTER INPUT`, the focus returns to the most recently occupied field in the current form, giving the user another chance to enter data in that field. In this case the BEFORE FIELD of the current field will be invoked.

As alternative, use the NEXT FIELD control instruction to give the focus to a specific field and force the dialog to continue. However, unlike CONTINUE INPUT, the NEXT FIELD instruction will skip the further control blocks that are normally executed.

**Related concepts**
- [EXIT INPUT instruction](#) on page 1496
- [ACCEPT INPUT instruction](#) on page 1495

**Related reference**
- [INPUT control blocks execution order](#) on page 1488

**EXIT INPUT instruction**

The EXIT INPUT instruction terminates the INPUT instruction and resumes the program execution at the instruction following the INPUT block.

Performing an EXIT INPUT instruction during a dialog is equivalent to canceling the dialog: No field validation will occur, and the AFTER FIELD or AFTER INPUT blocks will not be executed.

The dialog is exited immediately.

**Note:** The INT_FLAG will not be set to TRUE as when the cancel action is fired.

**Related concepts**
- [ACCEPT INPUT instruction](#) on page 1495
- [CONTINUE INPUT instruction](#) on page 1496

**Related reference**
- [INPUT control blocks execution order](#) on page 1488

**CLEAR instruction in dialogs**

The CLEAR field-list and CLEAR SCREEN ARRAY screen-array.* instructions clear the value buffer of specified form fields. The buffers are directly changed in the current form, and the program variables bound to the dialog are left unchanged. CLEAR can be used outside any dialog instruction, such as the DISPLAY BY NAME / TO instructions.
When a dialog is configured with the UNBUFFERED mode, there is no reason to clear field buffers since any variable assignment will synchronize field buffers. Actually, changing the field buffers with DISPLAY or CLEAR instruction in an UNBUFFERED dialog will have no visual effect, because the variables bound to the dialog will be used to reset the field buffer just before giving control back to the user. To clear fields of an UNBUFFERED dialog, just set to NULL the variables bound to the dialog. However, when using a CONSTRUCT, no program variables are associated with the dialog and no UNBUFFERED concept exits, and the CLEAR or DISPLAY TO / BY NAME instructions are the only way to modify the CONSTRUCT fields.

A screen array with a screen-line specification doesn’t make much sense in a GUI application using TABLE containers, you can therefore use the CLEAR SCREEN ARRAY instruction to clear all rows of a list.

Related concepts
Static display (DISPLAY/ERROR/MESSAGE/CLEAR) on page 1452
This section explains the instructions displaying static information to application forms, such as DISPLAY, ERROR, MESSAGE, CLEAR.

NEXT FIELD instruction

Understanding the NEXT FIELD instruction

The NEXT FIELD field-name instruction gives the focus to the specified field. This instruction can be used to control field input, in BEFORE FIELD, ON CHANGE or AFTER FIELD blocks, it can also force a DISPLAY ARRAY or INPUT ARRAY to stay in the current row when NEXT FIELD is used in the AFTER ROW block.

If it exists, the BEFORE FIELD block of the corresponding field is executed.

In editable dialogs, the purpose of the NEXT FIELD instruction is to give the focus to an editable field. Make sure that the field specified in NEXT FIELD is active and editable, or use NEXT FIELD CURRENT.

In a DISPLAY ARRAY using the FOCUSONFIELD attribute, NEXT FIELD can be used in conjunction with DIALOG.setCurrentRow(), to set the focus to a specific cell in the list.

Instead of the NEXT FIELD instruction, you can use the DIALOG.nextField("field-name") method to register a field, for example when the name is not known at compile time. However, this method only registers the field. It does not stop code execution, like the NEXT FIELD instruction does. You must execute a CONTINUE DIALOG to get the same behavior as NEXT FIELD.

Form field identification with NEXT FIELD

With the NEXT FIELD instruction, fields are identified by the form field name specification, not the program variable name used by the dialog.

Form fields are bound to program variables with the binding clause of the dialog instruction (INPUT variable-list FROM field-list, INPUT BY NAME variable-list, CONSTRUCT BY NAME sql ON column-list, CONSTRUCT sql ON column-list FROM field-list, INPUT ARRAY array-name FROM screen-array.*).

The field name specification can be any of the following:

- field-name
- table-name.field-name
- screen-record-name.field-name
- FORMONLY.field-name

Here are some examples:

- "cust_name"
- "customer.cust_name"
- "cust_screen_record.cust_name"
- "item_screen_array.item_label"
- "formonly.total"
When no field name prefix is used, the first form field matching that simple field name is used.

When using a prefix in the field name specification, it must match the field prefix assigned by the dialog field binding method used at the beginning of the interactive statement: When no screen-record has been explicitly specified in the field binding clause (for example, when using INPUT BY NAME variable-list), the field prefix must be the database table name or FORMONLY, as defined in the form file, or any valid screen-record using that field. When the FROM clause of the dialog specifies an explicit screen-record (for example, in INPUT variable-list FROM screen-record.* / field-list-with-screen-record-prefix OR INPUT ARRAY array-name FROM screen-array.*), the field prefix must be the screen-record name used in the FROM clause.

Abstract field identification is supported with the CURRENT, NEXT and PREVIOUS keywords. These keywords represent the current, next and previous fields respectively. When using FIELD ORDER FORM, the NEXT and PREVIOUS options follow the tabbing order defined by the form. Otherwise, they follow the order defined by the input binding list (with the FROM or BY NAME clause).

In a procedural dialog, if the focus is in the first field of an INPUT or CONSTRUCT sub-dialog, NEXT FIELD PREVIOUS will jump out of the current sub-dialog and set the focus to the previous sub-dialog. If the focus is in the last field of an INPUT or CONSTRUCT sub-dialog, NEXT FIELD NEXT will jump out of the current sub-dialog and set the focus to the next sub-dialog. NEXT FIELD NEXT or NEXT FIELD PREVIOUS also jumps to another sub-dialog when the focus is in a DISPLAY ARRAY sub-dialog. However, when using an INPUT ARRAY sub-dialog, NEXT FIELD NEXT from within the last column will loop to the first column of the current row, and NEXT FIELD PREVIOUS from within the first column will jump to the last column of the current row - the focus stays in the current INPUT ARRAY sub-dialog. When another sub-dialog gets the focus because of a NEXT FIELD NEXT/PREVIOUS, the newly-selected field depends on the sub-dialog type, following the tabbing order as if the end-user had pressed the tab or Shift-Tab key combination.

**NEXT FIELD to a non-editable field**

Non-editable fields are:

- Fields defined with the NOENTRY attribute in the form.
- Fields defined as PHANTOM fields in the form.
- Fields disabled at runtime with DIALOG.setFieldActive().
- Fields using a widget that does not allow input, such as a LABEL.

In an INPUT, INPUT ARRAY or CONSTRUCT instruction, if a NEXT FIELD instruction specifies a non-editable field, the BEFORE FIELD block of that field is executed. Then the dialog tries to give the focus to that field. Since the field cannot get the focus, the dialog will perform the last pressed navigation key (Tab, Shift-Tab, Left, Right, Up, Down, Accept) and execute the related control blocks, including the AFTER FIELD block of the non-editable field. If no last key is identified, the dialog considers Tab as fallback and moves to the next editable field as defined by the FIELD ORDER mode used by the dialog.

**Note:** In an INPUT ARRAY, doing a NEXT FIELD to a non-editable field which has no editable fields in the last navigation direction, the dialog may move to the next or previous row and give the focus to the first or last editable field of that row.

When selecting a non-editable field with NEXT FIELD NEXT, the runtime system will re-select the current field since it is the next editable field in the dialog. As a result the end user sees no change.

**Important:** Doing a NEXT FIELD to a non-editable field can lead to infinite loops in the dialog; To stay in the current field / row, use NEXT FIELD CURRENT instead.

**NEXT FIELD in procedural DIALOG blocks**

In a procedural dialog block, the NEXT FIELD field-name instruction gives the focus to the specified field controlled by INPUT, INPUT ARRAY or CONSTRUCT, or to a read-only list when using DISPLAY ARRAY.

When using a DISPLAY ARRAY sub-dialog, it is possible to give the focus to the list, by specifying the name of the first column as argument for NEXT FIELD.
If the target field specified in the NEXT FIELD instruction is inside the current sub-dialog, neither AFTER FIELD nor AFTER ROW will be invoked for the field or list you are leaving. However, the BEFORE FIELD control blocks of the destination field (or the BEFORE ROW in case of read-only list) will be executed.

If the target field specified in the NEXT FIELD instruction is outside the current sub-dialog, the AFTER FIELD, AFTER INSERT, AFTER ROW, and AFTER INPUT, AFTER DISPLAY, AFTER /CONSTRUCT control blocks will be invoked for the field or list you are leaving. Form-level validation rules will also be checked, as if the user had selected the new sub-dialog himself. This guarantees the current sub-dialog is left in a consistent state. The BEFORE INPUT, BEFORE DISPLAY, BEFORE CONSTRUCT, BEFORE ROW and the BEFORE FIELD control blocks of the destination field / list will then be executed.

**NEXT FIELD in record list control blocks**

When using NEXT FIELD in AFTER ROW or in ON ROW CHANGE of a DISPLAY ARRAY or INPUT ARRAY, the dialog will stay in the current row and give control back to the user.

This behavior allows you to implement data input rules:

```
AFTER ROW
    IF NOT int_flag AND arr_count()<=arr_curr() THEN
        IF arr[arr_curr()].it_count * arr[arr_curr()].it_value > maxval THEN
            ERROR "Amount of line exceeds max value."
            NEXT FIELD item_count
        END IF
    END IF
```

If a condition is not met for a given field, do a NEXT FIELD to that field to point the user to the problem. If a global condition for the current row is not met, use NEXT FIELD CURRENT to stay in the current row and current field.

**Related concepts**

Giving the focus to a form element on page 1731
How to force the focus to move or stay in a specific form element using program code.

Understanding multiple dialogs on page 1585
*Multiple dialogs* are defined with DIALOG blocks inside a FUNCTION.

**Examples**

INPUT dialog examples.

**Example 1: INPUT with binding by field position**

Form definition file "form1.per":

```
SCHEMA office

LAYOUT
GRID
{
    Customer id: [f001    ]
    First Name : [f002                    ]
    Last Name  : [f003                    ]
}
END
END

TABLES
    customer
END

ATTRIBUTES
    f001 = customer.id;
    f002 = customer.fname;
    f003 = customer.lname, UPSHIFT;
```
Program source code:

```
SCHEMA office
MAIN
    DEFINE custrec RECORD LIKE customer.*
    OPTIONS INPUT WRAP
    OPEN FORM f FROM "form1"
    DISPLAY FORM f
    LET INT_FLAG = FALSE
    INPUT custrec.* FROM sr_cust.*
    IF INT_FLAG = FALSE THEN
        DISPLAY custrec.*
        LET INT_FLAG = FALSE
    END IF
END MAIN
```

Example 2: INPUT with binding by field name

Form definition file "form1.per":

```
SCHEMA office
LAYOUT
GRID
{
    Customer id: [f001 ]
    First Name : [f002 ]
    Last Name  : [f003 ]
}
END

TABLES
customer
END

ATTRIBUTES
    f001 = customer.id;
    f002 = customer.fname;
    f003 = customer.lname, UPSHIFT;
END

INSTRUCTIONS
    SCREEN RECORD sr_cust(customer.*);
END
```

Program source code:

```
SCHEMA shop
```
MAIN

DEFINE custrec RECORD LIKE customer.*
DEFINE upd INTEGER

DATABASE shop
OPTIONS INPUT WRAP
OPEN FORM f FROM "form1"
DISPLAY FORM f

LET custrec.id = arg_val(1)
LET upd = (custrec.id < 0)

LET INT_FLAG = FALSE
INPUT BY NAME custrec.* ATTRIBUTES(UNBUFFERED, WITHOUT DEFAULTS=upd)
BEFORE INPUT
   MESSAGE "Enter customer information..."
   IF upd THEN
      SELECT fname, lname INTO custrec.fname, customer.lname
      FROM customer WHERE customer.id = custrec.id
   END IF
AFTER FIELD fname
   IF FIELD_TOUCHED(custrec.fname) AND custrec.fname IS NULL THEN
      LET custrec.lname = NULL
   END IF
AFTER INPUT
   MESSAGE "Input terminated..."
END INPUT

IF INT_FLAG = FALSE THEN
   DISPLAY custrec.*
   LET INT_FLAG = FALSE
END IF

END MAIN

Record list (DISPLAY ARRAY)
The DISPLAY ARRAY instruction provides record list navigation in an application form, with optional record modification actions.

Understanding the DISPLAY ARRAY instruction
The DISPLAY ARRAY dialog controls a list of records.

DISPLAY ARRAY is designed to browse a list of records, binding a static or dynamic array model to a screen array of the current displayed form.

A DISPLAY ARRAY instruction supports additional features such as drag & drop, tree-view management, built-in sort and search, multi-row selection and list modification triggers. For a detailed description of these features, see Table views on page 1853.

Use the DISPLAY ARRAY instruction to let the end user browse in a list of rows, after fetching a result set from the database. The result set is produced with a database cursor executing a SELECT statement. The SELECT SQL statement is usually completed at runtime with a WHERE clause produced from a CONSTRUCT dialog. When the DISPLAY ARRAY statement completes execution, the program must test the INT_FLAG on page 569 variable to check if the dialog was validated or canceled. If INT_FLAG is FALSE, the program can get the current row from arr_curr().

Depending on the type of DISPLAY ARRAY, you must implement dialog data blocks.

When the user browses the list, dialog control blocks such as BEFORE ROW are executed.
When the user clicks on an action view (button), or when an asynchronous event occurs, dialog interaction blocks like ON ACTION are executed.

The code inside a DISPLAY ARRAY dialog can use control instructions, dialog control functions, and the ui.Dialog class, to implement the dialog behavior.

**Related concepts**

Dialog programming basics on page 1706

This section describes basic dialog programming concepts.

**Syntax of DISPLAY ARRAY instruction**

The DISPLAY ARRAY instruction controls the display of a program array on the screen.

### Syntax

```
DISPLAY ARRAY array TO screen-array.*
[ HELP help-number ]
[ ATTRIBUTES ( [ display-attribute ] control-attribute ]
[ [ dialog-control-block ] [ ] ]
END DISPLAY ]
```

where **dialog-control-block** is one of:

```
[ BEFORE DISPLAY ]
[ AFTER DISPLAY ]
[ BEFORE ROW ]
[ AFTER ROW ]
[ ON IDLE seconds ]
[ ON TIMER seconds ]
[ ON ACTION action-name ]
[ [ ATTRIBUTES ( action-attributes-display-array ) ] ]
[ ON FILL BUFFER ]
[ ON SELECTION CHANGE ]
[ ON SORT ]
[ ON APPEND ]
[ ON INSERT ]
[ ON UPDATE ]
[ ON DELETE ]
[ ON EXPAND ( row-index ) ]
[ ON COLLAPSE ( row-index ) ]
[ ON DRAG_START ( dnd-object ) ]
[ ON DRAG_FINISH ( dnd-object ) ]
[ ON DRAG_ENTER ( dnd-object ) ]
[ ON DRAG_OVER ( dnd-object ) ]
[ ON DROP ( dnd-object ) ]
[ ON KEY ( key-name [, ]... ) ]
```

where **action-attributes-display-array** is:

```
[ TEXT = string ]
[ COMMENT = string ]
[ IMAGE = string ]
[ ACCELERATOR = string ]
[ DEFAULTVIEW = [ YES | NO | AUTO ] ]
[ CONTEXTMENU = [ YES | NO | AUTO ] ]
[ ROWBOUND ]
```
where `action-attributes-listmod-triggers` is:

```markdown
| TEXT = string |
| COMMENT = string |
| IMAGE = string |
| ACCELERATOR = string |
| DEFAULTVIEW = { YES | NO | AUTO } |
| CONTEXTMENU = { YES | NO | AUTO } |
```

where `dialog-statement` is one of:

```markdown
| statement |
| EXIT DISPLAY |
| CONTINUE DISPLAY |
| ACCEPT DISPLAY |
```

where `display-attribute` is:

```markdown
| BLACK | BLUE | CYAN | GREEN |
| MAGENTA | RED | WHITE | YELLOW |
| BOLD | DIM | INVISIBLE | NORMAL |
| REVERSE | BLINK | UNDERLINE |
```

where `control-attribute` is:

```markdown
| ACCEPT [ = boolean ] |
| CANCEL [ = boolean ] |
| KEEP CURRENT ROW [ = boolean ] |
| HELP = help-number |
| COUNT = row-count |
| UNBUFFERED [ = boolean ] |
| DETAILACTION = action-name |
| DOUBLECLICK = action-name |
| ACCESSORYTYPE = { DETAILBUTTON | DISCLOSUREINDICATOR | CHECKMARK } |
| FOCUSONFIELD |
```

1. `array` is a static or dynamic array containing the records you want to display.
2. `screen-array` is the name of the screen array used to display data.
3. `help-number` is an integer that associates a help message number with the instruction.
4. `action-name` identifies an action that can be executed by the user.
5. `seconds` is an integer literal or variable that defines a number of seconds.
6. `row-index` identifies the program variable which holds the row index corresponding to the tree view node that has been expanded or collapsed.
7. `dnd-object` references a `ui.DragDrop` variable defined in the scope of the dialog.
8. `key-name` is an hot-key identifier (such as `F11` or `Control-z`).
9. `statement` is any instruction supported by the language.
10. `row-count` defines the total number of rows for a static array.
11. `boolean` is a boolean expression that evaluates to `TRUE` or `FALSE`.
12. `action-attributes` are dialog-specific action attributes.
DISPLAY ARRAY programming steps

Follow this procedure to use the DISPLAY ARRAY dialog instruction.

To implement a DISPLAY ARRAY statement:

1. Create a form specification file containing a screen array. The screen array identifies the presentation elements to be used by the runtime system to display the rows.
2. Make sure that the program controls interruption handling with DEFER INTERRUPT, to manage the validation/cancellation of the interactive dialog.
3. Define an array of records with the DEFINE instruction. The members of the program array must correspond to the elements of the screen array, by number and data types. Static or a dynamic arrays can be used for the full list mode, but the paged mode requires a dynamic array. For new developments, use dynamic arrays in both cases.
4. Open and display the form, using OPEN WINDOW WITH FORM or the OPEN FORM/DISPLAY FORM instructions.
5. If you want to use the full list mode, fill the program array with data, typically with a result set cursor, counting the number of program records being filled with retrieved data.
6. Set the INT_FLAG on page 569 variable to FALSE.
7. Implement the DISPLAY ARRAY dialog block. When using a static array, specify the number of rows with the COUNT attribute in the ATTRIBUTES clause, or call the SET_COUNT() function before the dialog block. With dynamic arrays, the number of rows is automatically known by the dialog. Consider using the UNBUFFERED mode in new developments.
8. If you want to use the paged mode, define the total number of rows with the COUNT attribute (can be -1 for infinite number of rows), and add the ON FILL BUFFER clause that will contain the code to fill the dynamic array with the expected rows from fgl_dialog_getBufferStart() to fgl_dialog_getBufferLength().
9. If multi-row selection is needed, call the ui.Dialog.setSelectionMode() method in BEFORE DISPLAY to enable this mode.
10. Inside the DISPLAY ARRAY block, control the behavior of the instruction with BEFORE ROW, AFTER ROW and ON ACTION blocks.
11. After the interaction statement block, test the INT_FLAG predefined variable to check if the dialog was canceled (INT_FLAG=TRUE) or validated (INT_FLAG=FALSE). If the INT_FLAG variable is TRUE, reset it to FALSE in order not to disturb code that relies on this variable to detect interruption events from the GUI front-end or TUI console.
12. If needed, get the current row with the ARR_CURR() built-in function after dialog execution. During dialog execution, you can also use DIALOG.getCurrentRow().

Related concepts
Form specification files on page 1237
Form specification files are the source files defining the layout and content of application forms.

Using record lists
Dialog coding concepts, configuration and code structure.

Variable binding in DISPLAY ARRAY

The DISPLAY ARRAY statement binds the members of the array of record to the screen array fields specified with the TO keyword. Array members and screen array fields are bound by position (not by name). The number of members in the program array must match the number of fields in the screen record (that is, in a single row of the screen array).

```
SCHEMA stock
DEFINE cust_arr DYNAMIC ARRAY OF customer.*
...
DISPLAY ARRAY cust_arr TO sr.*
    ATTRIBUTES(UNBUFFERED)
    ...
END DISPLAY
```
The array is usually defined with a flat list of members with `ARRAY OF RECORD / END RECORD`. However, the array can be structured with sub-records and still be used with a `DISPLAY ARRAY` dialog. This is especially useful when you need to define arrays from database tables, and additional information needs to be managed at runtime (for example to hold image resource for each row, to be displayed with the `IMAGECOLUMN` attribute):

```sql
SCHEMA shop
DEFINE a_items DYNAMIC ARRAY OF RECORD
    item_data RECORD LIKE items.*,
    it_image STRING,
    it_count INTEGER
END RECORD
...
DISPLAY ARRAY a_items TO sr.*
...```

When using the `UNBUFFERED` attribute, the instruction is sensitive to program variable changes. This means that you do not have to `DISPLAY` the values; setting the program variable used by the dialog automatically displays the data in the corresponding form field.

```sql
ON ACTION change
    LET arr[arr_curr()].field1 = newValue()
```

If the program array has the same structure as a database table (this is the case when the array is defined with a `DEFINE LIKE` clause), you may not want to display/use some of the columns. You can achieve this by using `PHANTOM` fields in the screen array definition. Phantom fields will only be used to bind program variables, and will not be transmitted to the front-end for display.

**Related concepts**
- **Binding variables to form fields** on page 1715
  Some dialogs need program variables to store form field values.

**DISPLAY ARRAY instruction configuration**

This section describes the options that can be specified in the `ATTRIBUTES` clause of the `DISPLAY ARRAY` instruction. The options of the `ATTRIBUTES` clause override all default attributes and temporarily overrides any display attributes that the `OPTIONS` or the `OPEN WINDOW` statement specified for these fields. With the `DISPLAY ARRAY` statement, the `INVISIBLE` attribute is ignored.

**HELP option**

The `HELP` clause specifies the number of a help message to display if the user invokes the help in the `DISPLAY ARRAY` dialog. The predefined 'help message' to display is automatically created by the runtime system. You can bind action views to the 'help' action.

The `HELP` clause overrides the `HELP` attribute.

**COUNT option**

When using a dynamic array, the number of rows to be displayed is defined by the number of elements in the dynamic array; the `COUNT` attribute is ignored.

When using a static array or the paged mode, the number of rows to be displayed is defined by the `COUNT` attribute. You can also use the `SET_COUNT()` built-in function, but it is supported for backward compatibility only. If you don't know the total number of rows for the paged mode, you can specify `-1` for the `COUNT` attribute (or in the `SET_COUNT()` call before the dialog block). With `COUNT=-1`, the dialog will ask for rows by executing `ON FILL BUFFER` until less rows are provided than are required, or if you reset the number of rows to a value higher than `-1` with `ui.Dialog.setArrayLength()`. 
**KEEP CURRENT ROW option**

Depending on the list container used in the form, the current row may be highlighted during the execution of the dialog, and cleared when the instruction ends. You can change this default behavior by using the **KEEP CURRENT ROW** attribute, to force the runtime system to keep the current row highlighted.

**ACCEPT option**

The **ACCEPT** attribute can be set to **FALSE** to avoid the automatic creation of the default accept action. Use this attribute when you want to avoid dialog validation, or if you need to write a specific validation procedure using **ACCEPT DISPLAY**.

**CANCEL option**

The **CANCEL** attribute can be set to **FALSE** to avoid the automatic creation of the cancel default action. Use this attribute when you only need a validation action (accept), or when you want to write a specific cancellation procedure using **EXIT DISPLAY**.

If the **CANCEL=FALSE** option is set, no close action will be created, and you must write an **ON ACTION close** control block to create an explicit action.

**DOUBLECLICK option**

The **DOUBLECLICK** option can be used to define the action that will be fired when the user chooses a row from the list. Different configuration options are available to control the row selection action of desktop and mobile devices. For more details, see **Defining the action for a row choice** on page 1867.

**DETAILACTION option**

**Important:** This feature is only for mobile platforms.

The **DETAILACTION** attribute can be used to define the action that will be fired when the user selects the detail button of a row. The detail button is typically shown with an (i) icon on iOS devices. Note that the **DOUBLECLICK** attribute can be used to distinguish the action when the user selects the row instead of the detail button in the row. For more details, see **Row configuration on iOS devices** on page 1876.

**FOCUSONFIELD option**

**Important:** This feature is not supported on mobile platforms.

When the **FOCUSONFIELD** option is used, the **DISPLAY ARRAY** allows focus at the field (or cell) level. It it then possible to implement **BEFORE FIELD** and **AFTER FIELD** blocks, as well as using **NEXT FIELD** instructions. However, the dialog still manages a read-only list. For more details, see **Field-level focus in DISPLAY ARRAY** on page 1840.

**ACCESSORYTYPE option**

**Important:** This feature is only for mobile platforms.

The **ACCESSORYTYPE** attribute can be used to define the decoration of rows, typically used on an iOS device. Values can be **DETAILBUTTON,** **DISCLOSUREINDICATOR,** **CHECKMARK** to respectively get an (i), > or check mark icon. For more details, see **Row configuration on iOS devices** on page 1876.

**Related concepts**

- **Syntax of DISPLAY ARRAY instruction** on page 1502
- The **DISPLAY ARRAY** instruction controls the display of a program array on the screen.

**Default actions in DISPLAY ARRAY**

When a **DISPLAY ARRAY** instruction executes, the runtime system creates a set of **default actions.**
Field validation occurs and different DISPLAY ARRAY control blocks are executed, based on the invoked default action.

This table lists the default actions created for this dialog:

**Table 369: Default actions created for the DISPLAY ARRAY dialog**

<table>
<thead>
<tr>
<th>Default action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>accept</td>
<td>Validates the DISPLAY ARRAY dialog (validates current row selection)</td>
</tr>
<tr>
<td></td>
<td>Creation can be avoided with ACCEPT = FALSE attribute.</td>
</tr>
<tr>
<td>cancel</td>
<td>Cancels the DISPLAY ARRAY dialog (no validation, INT_FLAG is set to TRUE)</td>
</tr>
<tr>
<td></td>
<td>Creation can be avoided with CANCEL = FALSE attribute.</td>
</tr>
<tr>
<td>close</td>
<td>By default, cancels the DISPLAY ARRAY dialog (no validation, INT_FLAG is set to TRUE)</td>
</tr>
<tr>
<td></td>
<td>Default action view is hidden. See Implementing the close action on page 1772.</td>
</tr>
<tr>
<td>help</td>
<td>Shows the help topic defined by the HELP clause.</td>
</tr>
<tr>
<td></td>
<td>Only created when a HELP clause is defined.</td>
</tr>
<tr>
<td>nextrow</td>
<td>Moves to the next row in a list displayed in one row of fields.</td>
</tr>
<tr>
<td></td>
<td>See note (1).</td>
</tr>
<tr>
<td>prevrow</td>
<td>Moves to the previous row in a list displayed in one row of fields.</td>
</tr>
<tr>
<td></td>
<td>See note (1).</td>
</tr>
<tr>
<td>firstrow</td>
<td>Moves to the first row in a list displayed in one row of fields.</td>
</tr>
<tr>
<td></td>
<td>See note (1).</td>
</tr>
<tr>
<td>lastrow</td>
<td>Moves to the last row in a list displayed in one row of fields.</td>
</tr>
<tr>
<td></td>
<td>See note (1).</td>
</tr>
<tr>
<td>find</td>
<td>Opens the fglfind dialog window to let the user enter a search value, and searches for the row matching the value.</td>
</tr>
<tr>
<td></td>
<td>See note (2).</td>
</tr>
<tr>
<td>findnext</td>
<td>Searches for the next row matching the value entered during the fglfind dialog.</td>
</tr>
<tr>
<td></td>
<td>See note (2).</td>
</tr>
</tbody>
</table>

Notes:

1. The action is only created with a DISPLAY ARRAY or INPUT ARRAY using a screen record bound to a set of form fields in a GRID container, and this set of fields show only a single row of the list. The action is not created when using a screen array bound to a list container such as TABLE, TREE and SCROLLGRID, or to a set of fields in a GRID container, that repeat on several lines to show more than one single row.

2. The action is only created if the context allows built-in find.

The accept and cancel default actions can be avoided with the ACCEPT and CANCEL dialog control attributes:

```display array arr to sr.* attributes( cancel=false, ... )```
Related concepts
Dialog programming basics on page 1706
This section describes basic dialog programming concepts.

DISPLAY ARRAY data blocks
Data blocks are dialog triggers that are invoked when the dialog controller needs data to feed the view with values.
Such blocks are typically used when record list data is provided dynamically, with the display array paged mode or when implementing dynamic tree-views.

ON FILL BUFFER block
The ON FILL BUFFER block is used to fill a page of rows into the dynamic array, based on an offset and a number of rows.
This data block is only used in DISPLAY ARRAY dialog blocks.
The offset can be retrieved with the FGL_DIALOG_GETBUFFERSTART() built-in function and the number of rows to provide is defined by the FGL_DIALOG_GETBUFFERLENGTH() built-in function.
The ON FILL BUFFER block is executed when the runtime system needs data rows to fill the current page of the list dialog. This can happen before a BEFORE DISPLAY of a singular DISPLAY ARRAY, or before the BEFORE DIALOG block of a DIALOG/END DIALOG instruction containing DISPLAY ARRAY sub-dialogs.
For more details about ON FILL BUFFER usage, see Paged mode of DISPLAY ARRAY on page 1843.

ON EXPAND block
The ON EXPAND block is executed when a tree view node is expanded (opened).
This data block is used to implement dynamic trees in a DISPLAY ARRAY, where nodes are added, depending on the nodes opened by the end user.
For more details, see Dynamic filling of very large trees on page 1896.

Related concepts
ON COLLAPSE block on page 1508

ON COLLAPSE block
The ON COLLAPSE block is executed when a tree view node is collapsed (i.e. closed).
This data block is used to implement dynamic trees in a DISPLAY ARRAY, where nodes are removed from view by the user closing or collapsing them.
For more details, see Dynamic filling of very large trees on page 1896.

Related concepts
ON EXPAND block on page 1508

DISPLAY ARRAY control blocks
DISPLAY ARRAY control blocks execution order
This table shows the order in which the runtime system executes the control blocks in the DISPLAY ARRAY instruction, based on the user action:

<table>
<thead>
<tr>
<th>Table 370: Control blocks execution order in DISPLAY ARRAY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context / User action</td>
</tr>
<tr>
<td>-----------------------</td>
</tr>
<tr>
<td>Entering the dialog</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Context / User action</td>
</tr>
<tr>
<td>-----------------------</td>
</tr>
</tbody>
</table>
| Moving to a different row | 1. AFTER ROW (the current row)  
|                       | 2. BEFORE ROW (the new row)    |
| Validating the dialog  | 1. AFTER ROW                  |
|                       | 2. AFTER DISPLAY               |
| Canceling the dialog  | 1. AFTER ROW                  |
|                       | 2. AFTER DISPLAY               |
| Firing the insert or append action for the ON INSERT block | 1. AFTER ROW  
|              | 2. ON INSERT                   |
|                       | 3. BEFORE ROW                  |
| Firing the delete action for the ON DELETE block | 1. AFTER ROW  
|              | 2. ON DELETE                   |
|                       | 3. BEFORE ROW                  |

**Related concepts**

ON APPEND block on page 1516  
ON UPDATE block on page 1518

**BEFORE DISPLAY block**

**BEFORE DISPLAY block in singular and parallel DISPLAY ARRAY dialogs**

In a singular DISPLAY ARRAY instruction, or when used as parallel dialog, the BEFORE DISPLAY is only executed once when the dialog is started.

The BEFORE DISPLAY block is executed once at dialog startup, before the runtime system gives control to the user. This block can be used to display messages to the user, initialize program variables, and set up the dialog instance by deactivating actions the user is not allowed to execute.

```plaintext
DISPLAY ARRAY p_items TO s_items.*  
BEFORE DISPLAY  
    CALL DIALOG.setActionActive("clear_item_list", is_super_user())
```

**BEFORE DISPLAY block DISPLAY ARRAY of procedural DIALOG**

In a DISPLAY ARRAY sub-dialog of a procedural DIALOG instruction, the BEFORE DISPLAY block is executed when a DISPLAY ARRAY list gets the focus.

BEFORE DISPLAY is executed before the BEFORE ROW block.

In this example the BEFORE DISPLAY block enables an action and displays a message:

```plaintext
DISPLAY ARRAY p_items TO s_items.*  
BEFORE DISPLAY  
    CALL DIALOG.setActionActive("print_list", TRUE)  
    MESSAGE "You are now in the list of items"
```

**Related concepts**

BEFORE INPUT block on page 1489  
BEFORE CONSTRUCT block on page 1573  
AFTER DISPLAY block on page 1510
**AFTER DISPLAY block**

**AFTER DISPLAY block in singular and parallel DISPLAY ARRAY dialogs**

In a singular `DISPLAY ARRAY` instruction, or when used as parallel dialog, the `AFTER DISPLAY` is only executed once when the dialog is ended.

You typically implement dialog finalization in this block.

```
DISPLAY ARRAY p_items TO s_items.*
    AFTER DISPLAY
        DISPLAY "Current row is: ", arr_curr()
```

**AFTER DISPLAY block in DISPLAY ARRAY of procedural DIALOG**

In a `DISPLAY ARRAY` sub-dialog of a procedural `DIALOG` instruction, the `AFTER DISPLAY` block is executed when a `DISPLAY ARRAY` list loses the focus and the focus goes to another sub-dialog.

If the focus leaves the current group and goes to an action view, this trigger is not executed, because the focus has not gone to another sub-dialog yet.

`AFTER DISPLAY` is executed after the `AFTER ROW` block.

In this example, the `AFTER DISPLAY` block disables an action that is specific to the current list:

```
DISPLAY ARRAY p_items TO s_items.*
    AFTER DISPLAY
        CALL DIALOG.setActionActive("clear_item_list", FALSE)
```

**Related concepts**

- AFTER INPUT block on page 1490
- AFTER CONSTRUCT block on page 1574
- BEFORE DISPLAY block on page 1509

**BEFORE ROW block**

**BEFORE ROW block in singular and parallel DISPLAY ARRAY, INPUT ARRAY dialogs**

In a singular `DISPLAY ARRAY`, `INPUT ARRAY` instruction, or when used as parallel dialog, the `BEFORE ROW` block is executed each time the user moves to another row. This trigger can also be executed in other situations, such as when you delete a row, or when the user tries to insert a row but the maximum number of rows in the list is reached.

You typically do some dialog setup / message display in the `BEFORE ROW` block, because it indicates that the user selected a new row or entered in the list.

When the dialog starts, `BEFORE ROW` will be executed for the current row, but only if there are data rows in the array.

When called in this block, `DIALOG.getCurrentRow() / arr_curr()` return the index of the current row.

In this example, the `BEFORE ROW` block gets the new row number and displays it in a message:

```
DISPLAY ARRAY ...
    ...
    BEFORE ROW
        MESSAGE "We are on row # ", arr_curr()
    ...
```
BEFORE ROW block in DISPLAY ARRAY and INPUT ARRAY of procedural DIALOG

In an INPUT or INPUT ARRAY sub-dialog of a procedural DIALOG instruction, the BEFORE ROW block is executed when a DISPLAY ARRAY or INPUT ARRAY list gets the focus, or when the user moves to another row inside a list. This trigger can also be executed in other situations, for example when you delete a row, or when the user tries to insert a row but the maximum number of rows in the list is reached.

You typically do some dialog setup / message display in the BEFORE ROW block, because it indicates that the user selected a new row. Do not use this trigger to detect focus changes; Use the BEFORE DISPLAY or BEFORE INPUT blocks instead.

In DISPLAY ARRAY, BEFORE ROW is executed after the BEFORE DISPLAY block. In INPUT ARRAY, BEFORE ROW is executed before the BEFORE INSERT and BEFORE FIELD blocks and after the BEFORE INPUT blocks.

When the procedural dialog starts, BEFORE ROW will only be executed if the list has received the focus and there is a current row (the array is not empty). If you have other elements in the form which can get the focus before the list, BEFORE ROW will not be triggered when the dialog starts. You must pay attention to this, because this behavior is different to the behavior of singular DISPLAY ARRAY or INPUT ARRAY. In singular dialogs, the BEFORE ROW block is always executed when the dialog starts (and when there are rows in the array).

When called in this block, DIALOG.getCurrentRow() / arr_curr() return the index of the current row.

In this example the BEFORE ROW block displays a message with the current row number:

```
DISPLAY ARRAY p_items TO s_items.*
BEFORE ROW
  MESSAGE "We are in items, on row ", DIALOG.getCurrentRow("s_items")
```

Related concepts

BEFORE INPUT block on page 1489
BEFORE DISPLAY block on page 1509

AFTER ROW block

AFTER ROW block in singular and parallel DISPLAY ARRAY, INPUT ARRAY dialogs

In a singular DISPLAY ARRAY, INPUT ARRAY instruction, or when used as parallel dialog, the AFTER ROW block is executed each time the user moves to another row, before the current row is left. This trigger can also be executed in other situations, such as when you delete a row, or when the user inserts a new row.

A NEXT FIELD instruction executed in the AFTER ROW control block will keep the user entry in the current row. Use this behavior to implement row validation and prevent the user from leaving the list or moving to another row.

When called in this block, DIALOG.getCurrentRow() / arr_curr() returns the index of the row that you are leaving.

AFTER ROW block in DISPLAY ARRAY and INPUT ARRAY of procedural DIALOG

In an INPUT or INPUT ARRAY sub-dialog of a procedural DIALOG instruction, the AFTER ROW block is executed when a DISPLAY ARRAY or INPUT ARRAY list loses the focus, or when the user moves to another row in a list. This trigger can also be executed in other situations, for example when you delete a row, or when the user inserts a new row.

AFTER ROW is executed after the AFTER FIELD, AFTER INSERT and before AFTER DISPLAY or AFTER INPUT blocks.

When called in this block, DIALOG.getCurrentRow() / arr_curr() returns the index of the of the row that you are leaving.

For both INPUT ARRAY and DISPLAY ARRAY sub-dialogs, a NEXT FIELD executed in the AFTER ROW control block will keep the focus in the list and stay in the current row. Use this feature to implement row validation and prevent the user from leaving the list or moving to another row.
AFTER ROW and temporary rows in INPUT ARRAY

Important: After creating a temporary row at the end of a list driven by INPUT ARRAY, if you leave that row to go to a previous row without data input (setting the touched flag), or when the cancel action is invoked, the temporary row will be automatically removed. The AFTER ROW block will be executed for the temporary row, but `ui.Dialog.getCurrentRow()//arr_curr()` will be one row greater than `ui.Dialog.getArrayLength()//ARR_COUNT()`. In this case, it is recommended that you ignore the AFTER ROW event. For example, it is recommended that you avoid executing a NEXT FIELD or CONTINUE INPUT instruction, and trying to access the dynamic array with a row index that is greater than the total number of rows, otherwise the runtime system will adapt the total number of rows to the actual number of rows in the program array.

In this example, the AFTER ROW block checks the current row index and verifies a variable value to force the focus to stay in the current row if the value is wrong:

```plaintext
INPUT ARRAY p_items FROM s_items.*
...
AFTER ROW
   LET r = DIALOG.getCurrentRow("s_items")
   IF r <= DIALOG.getArrayLength("s_items") THEN
      IF NOT item_is_valid_quantity(p_item[r].item_quantity) THEN
         ERROR "Item quantity is not valid"
         NEXT FIELD item_quantity
      END IF
   END IF
END IF
```

Another way to handle the case of temporary rows in AFTER ROW is to use a flag to know if the AFTER INSERT block was executed. The AFTER INSERT block is not executed if the temporary row is automatically removed. By setting a first value in BEFORE INSERT and changing the flag in AFTER INSERT, you can detect if the row was permanently added to the list:

```plaintext
INPUT ARRAY p_items FROM s_items.*
...
BEFORE INSERT
   LET op = "T"
...
AFTER INSERT
   LET op = "I"
...
AFTER ROW
   IF op == "I" THEN
      IF NOT item_is_valid_quantity(p_item[arr_curr()].item_quantity) THEN
         ERROR "Item quantity is not valid"
         NEXT FIELD item_quantity
      END IF
      WHENEVER ERROR CONTINUE
      INSERT INTO items (item_num, item_name, item_quantity) VALUES ( p_item[arr_curr()].* )
      WHENEVER ERROR STOP
      IF SQLCA.SQLCODE<0 THEN
         ERROR "Could not insert the record into database!"
         NEXT FIELD CURRENT
      ELSE
         MESSAGE "Record has been inserted successfully"
      END IF
   END IF
...
```

Related concepts
NEXT FIELD instruction on page 1497
BEFORE ROW block on page 1510
**BEFORE FIELD block**

In dialog instructions INPUT, INPUT ARRAY, CONSTRUCT or in a DISPLAY ARRAY using the FOCUSONFIELD attribute, the BEFORE FIELD block is executed every time the specified field gets the focus.

For single record inputs driven by INPUT or query by example (QBEs) driven by CONSTRUCT, the BEFORE FIELD block is executed when moving the focus from field to field.

For editable lists driven by INPUT ARRAY, the BEFORE FIELD block is executed when moving the focus from field to field in the same row, or when moving to another row in the same column.

For record lists driven by DISPLAY ARRAY using the FOCUSONFIELD attribute, the BEFORE FIELD block is executed when moving the focus from field to field. However, the fields will not be editable as in an INPUT ARRAY.

**Important:** The BEFORE FIELD block is also executed when performing a NEXT FIELD instruction.

The BEFORE FIELD keywords must be followed by a list of form field specification. The screen-record name can be omitted.

**BEFORE FIELD is executed after** BEFORE INPUT, BEFORE CONSTRUCT, BEFORE ROW and BEFORE INSERT.

Use this block to do some field value initialization, or to display a message to the user:

```plaintext
INPUT BY NAME p_cust.* ...
   BEFORE FIELD cust_status
       LET p_cust.cust_comment = NULL
       MESSAGE "Enter customer status"
```

When using the default FIELD ORDER CONSTRAINT mode, the dialog executes the BEFORE FIELD block of the field corresponding to the first variable of an INPUT or INPUT ARRAY, even if that field is not editable (NOENTRY, hidden or disabled). The block is executed when you enter the dialog and every time you create a new row in the case of INPUT ARRAY. This behavior is supported for backward compatibility. The block is **not** executed when using the FIELD ORDER FORM, the mode recommended for DIALOG instructions.

With the FIELD ORDER FORM mode, for each dialog executing for the first time with a specific form, the BEFORE FIELD block will be invoked for the first field of the initial tabbing list defined by the form, even if that field was hidden or moved around in a table. The dialog then behaves as if a NEXT FIELD first-visible-column had been executed in the BEFORE FIELD of that field.

When form-level validation occurs and a field contains an invalid value, the dialog gives the focus to the field, but no BEFORE FIELD trigger will be executed.

**Related concepts**

Form-level validation rules on page 1727

Form-level validation rules can be defined for each field controlled by a dialog.

**AFTER FIELD block on page 1492**

**AFTER FIELD block**

In dialog instructions INPUT, INPUT ARRAY, CONSTRUCT or in a DISPLAY ARRAY using the FOCUSONFIELD attribute, the AFTER FIELD block is executed every time the focus leaves the specified field.

For single record inputs driven by INPUT or query by example (QBEs) driven by CONSTRUCT, the AFTER FIELD block is executed when moving the focus from field to field.

For editable lists driven by INPUT ARRAY, the AFTER FIELD block is executed when moving the focus from field to field in the same row, or when moving to another row in the same column.

For record lists driven by DISPLAY ARRAY using the FOCUSONFIELD attribute, the AFTER FIELD block is executed when moving the focus from field to field. However, the fields will not be editable as in an INPUT ARRAY.
The \texttt{AFTER FIELD} keywords must be followed by a list of form field specifications. The screen-record name can be omitted.

\texttt{AFTER FIELD} is executed before \texttt{AFTER INSERT, ON ROW CHANGE, AFTER ROW, AFTER INPUT} or \texttt{AFTER CONSTRUCT}.

When a \texttt{NEXT FIELD} instruction is executed in an \texttt{AFTER FIELD} block, the cursor moves to the specified field, which can be the current field. This can be used to prevent the user from moving to another field / row during data input. Note that the \texttt{BEFORE FIELD} block is also executed when \texttt{NEXT FIELD} is invoked.

The \texttt{AFTER FIELD} block of the current field is not executed when performing a \texttt{NEXT FIELD}; only \texttt{BEFORE INPUT}, \texttt{BEFORE CONSTRUCT}, \texttt{BEFORE ROW}, and \texttt{BEFORE FIELD} of the target item might be executed, depending on the sub-dialog type.

When \texttt{ACCEPT DIALOG}, \texttt{ACCEPT INPUT}, or \texttt{ACCEPT CONSTRUCT} is performed, the \texttt{AFTER FIELD} trigger of the current field is executed.

Use the \texttt{AFTER FIELD} block to implement field validation rules:

\begin{verbatim}
INPUT BY NAME p_item.* ...
  AFTER FIELD item_quantity
    IF p_item.item_quantity <= 0 THEN
      ERROR "Item quantity cannot be negative or zero"
      LET p_item.item_quantity = 0
    NEXT FIELD item_quantity
END IF
\end{verbatim}

\textbf{Related concepts}

\texttt{ACCEPT DIALOG instruction} on page 1640
\texttt{ON CHANGE block} on page 1492

\textbf{DISPLAY ARRAY interaction blocks}

\texttt{ON ACTION block}

The \texttt{ON ACTION} \texttt{action-name} blocks execute a sequence of instructions when the user triggers a specific action.

A typical action handler block looks like this:

\begin{verbatim}
ON ACTION action-name
  instruction ...
\end{verbatim}

Action blocks are bound by name to action views (like buttons) in the current form. Action views can be \texttt{BUTTON}, \texttt{TOOLBAR} buttons, or \texttt{TOPMENU} options, and if no explicit action view is defined, actions are rendered with a default action view, depending on the type of front-end.

This example defines an action block to open a typical zoom window and let the user select a customer record:

\begin{verbatim}
ON ACTION zoom
  CALL zoom_customers() RETURNING st, rec.cust_id, rec.cust_name
\end{verbatim}

In a dialog handling user input such as \texttt{INPUT, INPUT ARRAY} and \texttt{CONSTRUCT}, if an action is specific to a field, add the \texttt{INFIELD} clause to have the action automatically enabled when the corresponding field gets the focus:

\begin{verbatim}
ON ACTION zoom INFIELD cust_city
  CALL zoom_cities() RETURN st, rec.cust_city
\end{verbatim}

In most cases actions are decorated with action defaults in form files, but there can be cases where the \texttt{ON ACTION} handler needs to define its own attributes at the program level. This can be done by adding the \texttt{ATTRIBUTES()} clause of \texttt{ON ACTION}:

\begin{verbatim}
ON ACTION custinfo ATTRIBUTES(DISCLOSUREINDICATOR, IMAGE="info")
\end{verbatim}
CALL show_customer_info()

For more details about action handlers, and action configuration, see Dialog actions on page 1739.

**Related concepts**
- Configuring actions on page 1744
- Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with action attributes.
- Action handling basics on page 1739
- This topic describes the basics of action views, action events, and action handlers.

**ON IDLE block**

The ON IDLE seconds clause defines a set of instructions that must be executed after a given period of user inactivity. This interaction block can be used, for example, to quit the dialog after the user has not interacted with the program for a specified period of time.

Do not mix ON TIMER and ON IDLE clauses.

As ON IDLE can fire field input validation, it is therefore not recommended in dialogs allowing input.

The parameter of ON IDLE must be an integer literal or variable. If the value is zero, the dialog timeout is disabled.

It is not recommended to use the ON IDLE trigger with a short timeout period such as 1 or 2 seconds; The purpose of this trigger is to give the control back to the program after a relatively long period of inactivity (10, 30 or 60 seconds). This is typically the case when the end user leaves the workstation, or gets a phone call. The program can then execute some code before the user gets the control back.

```plaintext
ON IDLE 30
  IF ask_question("Do you want to reload information from the database?") THEN
      -- Fetch data back from the db server
  END IF
```

**Important:** The timeout value is taken into account when the dialog initializes its internal data structures. If you use a program variable instead of an integer constant, any change of the variable will have no effect if the variable is changed after the dialog has initialized. If you want to change the value of the timeout variable, it must be done before the dialog block.

**Related concepts**
- Get program control if user is inactive on page 1712
  Execute some code after a given number of seconds, when the user does not interact with the program.
- ON TIMER block on page 1466

**ON KEY block**

An ON KEY (key-name) block defines an action with a hidden action view (no default button is visible), that executes a sequence of instructions when the user presses the specified key.

**Note:** The ON KEY block is supported for backward compatibility with TUI mode applications. In new developments, consider using **ON ACTION** with accelerators defined in action defaults.

An ON KEY block can specify up to four accelerator keys. Each key creates a specific action object that will be identified by the key name in lowercase.

For example, ON KEY(F5,F6) creates two actions with the names f5 and f6. Each action object will get an acceleratorName attribute assigned, with the corresponding accelerator name. The specified keys must be one of the virtual keys.

**Note:** The KEY() clause allows a comma-separated list of keys. Up to four keys can be specified. For new developments, consider using a single key, or prefer **ON ACTION** handlers with a single accelerator definition in action defaults.
In GUI mode, action defaults are applied for \texttt{ON KEY} actions by using the name of the action (the key name). You can define secondary accelerator keys, as well as default decoration attributes like button text and image, by using the key name as action identifier. The action name is always in lowercase letters.

Check carefully \texttt{ON KEY CONTROL-?} statements to avoid having duplicate accelerators for multiple actions due to the accelerators defined by action defaults. Additionally, \texttt{ON KEY} statements used with \texttt{ESC, TAB, UP, DOWN, LEFT, RIGHT, HELP, NEXT, PREVIOUS, INSERT, CONTROL-M, CONTROL-X, CONTROL-V, CONTROL-C} and \texttt{CONTROL-A} should be avoided for use in GUI programs, because it's very likely to clash with default accelerators defined in the factory action defaults file provided by default.

By default, \texttt{ON KEY} actions are not decorated with a default button in the action frame (the default action view). You can show the default button by configuring a \texttt{text} attribute with the action defaults.

```
ON KEY (CONTROL-Z)
  CALL open_zoom()
```

**Related concepts**

- **Configuring actions** on page 1744
  Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with \textit{action attributes}.

- **Default action views** on page 1743
  A default action view is created to render an action handler when no explicit action view exists for it.

**\textbf{ON TIMER} block**

The \texttt{ON TIMER \textit{seconds}} clause defines a set of instructions that must be executed at regular intervals. This interaction block can be used, for example, to check if a message has arrived in a queue, and needs to be processed.

Do not mix \texttt{ON TIMER} and \texttt{ON IDLE} clauses.

As \texttt{ON TIMER} can fire field input validation, it is therefore not recommended in dialogs allowing input.

The parameter of \texttt{ON TIMER} must be an integer literal or variable. If the value is zero, the dialog timeout is disabled.

It is not recommended to use the \texttt{ON TIMER} trigger with a short timeout period, such as 1 or 2 seconds. The purpose of this trigger is to give the control back to the program after a reasonable period of time, such as 10, 20 or 60 seconds.

```
ON TIMER 30
  CALL check_for_messages()
```

**Important:** The timer value is taken into account when the dialog initializes its internal data structures. If you use a program variable instead of an integer constant, a change of the variable has no effect if the change takes place after the dialog has initialized. If you want to change the value of the timeout variable, it must be done before the dialog block.

**Related concepts**

- **Get program control on a regular (timed) basis** on page 1713
  Execute some code after a given number of seconds, with or without user interaction with the program.

- **ON IDLE block** on page 1465

**\textbf{ON APPEND} block**

Similar to the \texttt{ON INSERT} control block, the \texttt{ON APPEND} trigger can be used to enable row creation during a \texttt{DISPLAY ARRAY} dialog. If this block is defined, the dialog will automatically create the append action. This action can be decorated, enabled and disabled as a regular action.

If the dialog defines an \texttt{ON ACTION append} interaction block and the \texttt{ON APPEND} block is used, the compiler will stop with error -8408.
When the user fires the append action, the dialog first executes the user code of the \textit{AFTER ROW} block if defined. Then the dialog moves to the end of the list, and creates a new row after the last existing row. After creating the row, the dialog executes the user code of the \textit{ON APPEND} block.

The dialog handles only row creation actions and navigation, you must program the record input with a regular \texttt{INPUT} statement, to let the end user enter data for the newly-created row. This is typically done with an \texttt{INPUT} binding explicitly array fields to the screen record fields. The new current row in the program array is identified with \texttt{arr\_curr()}, and the current screen line in the form is defined by \texttt{scr\_line()}:

\begin{verbatim}
DISPLAY ARRAY arr TO sr.*
...
ON APPEND
  INPUT arr[arr\_curr()].* FROM sr[scr\_line()].* ;
...
\end{verbatim}

Pay attention to the semicolon ending the \texttt{INPUT} instruction, which is usually needed here to solve a language grammar conflict when nested dialog instructions are implemented.

After the user code is executed, the dialog gets the control back and processes the new row as follows:

- If the \texttt{INT\_FLAG} global variable is \texttt{FALSE} and \texttt{STATUS} is zero, the new row is kept in the program array, and the \textit{BEFORE ROW} block is executed for the newly-created row.
- If the \texttt{INT\_FLAG} global variable is \texttt{TRUE} or \texttt{STATUS} is different from zero, the new row is removed from the program array, and the \textit{BEFORE ROW} block is executed for the row that existed at the current position, before the new row was created.

The \texttt{DISPLAY ARRAY} dialog always resets \texttt{INT\_FLAG} to \texttt{FALSE} and \texttt{STATUS} to zero before executing the user code of the \texttt{ON APPEND} block.

The append action is disabled if the maximum number of rows is reached.

If needed, the \texttt{ON APPEND} handler can be configured with action attributes by added an \texttt{ATTRIBUTES()} clause, as with user-defined action handlers:

\begin{verbatim}
ON APPEND ATTRIBUTES(TEXT=%"custlist.delete", IMAGE="listdel")
\end{verbatim}

\textbf{Related concepts}

- \texttt{Record input (INPUT)} on page 1482
  The \texttt{INPUT} instruction provides single record input control in an application form.
- \texttt{ON UPDATE block} on page 1518
- \texttt{ON DELETE block} on page 1519

\texttt{ON INSERT block}

Similar to the \texttt{ON APPEND} control block, the \texttt{ON INSERT} trigger can be used to enable row creation during a \texttt{DISPLAY ARRAY} dialog. If this block is defined, the dialog will automatically create the insert action. This action can be decorated, enabled and disabled as a regular action.

If the dialog defines an \texttt{ON ACTION insert} interaction block and the \texttt{ON INSERT} block is used, the compiler will stop with error -8408.

When the user fires the insert action, the dialog first executes the user code of the \texttt{AFTER ROW} block if defined. Then the new row is created: The insert action creates a new row before current row in the list. After creating the row, the dialog executes the user code of the \texttt{ON INSERT} block.

The dialog handles only row creation actions and navigation, you must program the record input with a regular \texttt{INPUT} statement, to let the end user enter data for the newly-created row. This is typically done with an \texttt{INPUT} binding explicitly array fields to the screen record fields. The new current row in the program array is identified with \texttt{arr\_curr()}, and the current screen line in the form is defined by \texttt{scr\_line()}:

\begin{verbatim}
DISPLAY ARRAY arr TO sr.*
\end{verbatim}
ON INSERT
  INPUT arr[arr_curr()].* FROM sr[scr_line()].* ;
...

Pay attention to the semicolon ending the \texttt{INPUT} instruction, which is usually needed here to solve a language grammar conflict when nested dialog instructions are implemented.

After the user code is executed, the dialog gets the control back and processes the new row as follows:

- If the \texttt{INT\_FLAG} global variable is \texttt{FALSE} and \texttt{STATUS} is zero, the new row is kept in the program array, and the \texttt{BEFORE ROW} block is executed for the new created row.
- If the \texttt{INT\_FLAG} global variable is \texttt{TRUE} or \texttt{STATUS} is different from zero, the new row is removed from the program array, and the \texttt{BEFORE ROW} block is executed for the row that was existing at the current position, before the new row was created.

The \texttt{DISPLAY ARRAY} dialog always resets \texttt{INT\_FLAG} to \texttt{FALSE} and \texttt{STATUS} to zero before executing the user code of the \texttt{ON INSERT} block.

The insert action is disabled if the maximum number of rows is reached.

If needed, the \texttt{ON INSERT} handler can be configured with action attributes by added an \texttt{ATTRIBUTES()} clause, as with user-defined action handlers:

\begin{verbatim}
ON INSERT ATTRIBUTES(TEXT="$\text{custlist.delete}$", IMAGE="listdel")
\end{verbatim}

\textbf{Related concepts}

\texttt{Record input (INPUT)} on page 1482

The \texttt{INPUT} instruction provides single record input control in an application form.

\texttt{ON UPDATE block} on page 1518

\texttt{ON DELETE block} on page 1519

\textbf{ON UPDATE block}

The \texttt{ON UPDATE} trigger can be used to enable row modification during a \texttt{DISPLAY ARRAY} dialog. If this block is defined, the dialog will automatically create the update action. This action can be decorated, enabled and disabled as regular actions.

You typically configure the \texttt{TABLE} container in the form by defining the \texttt{DOUBLECLICK} attribute to "update", in order to trigger the update action when the user double-clicks on a row.

If the dialog defines an \texttt{ON ACTION update} interaction block and the \texttt{ON UPDATE} block is used, the compiler will stop with error -8408.

When the user fires the \texttt{update} action, the dialog executes the user code of the \texttt{ON UPDATE} block.

The dialog handles only the row modification action and navigation, you must program the record input with a regular \texttt{INPUT} statement, to let the end user modify the data of the current row. This is typically done with an \texttt{INPUT} binding explicitly array fields to the screen record fields, with the \texttt{WITHOUT DEFAULTS} clause. The current row in the program array is identified with \texttt{arr_curr()}, and the current screen line in the form is defined by \texttt{scr_line()}:

\begin{verbatim}
DISPLAY ARRAY arr TO sr.*
...
  ON UPDATE
    INPUT arr[arr_curr()].* WITHOUT DEFAULTS FROM sr[scr_line()].* ;
  ...
\end{verbatim}

Pay attention to the semicolon ending the \texttt{INPUT} instruction, which is usually needed here to solve a language grammar conflict when nested dialog instructions are implemented.

After the user code is executed, the dialog gets the control back and processes the current row as follows:
• If the `INT_FLAG` global variable is `FALSE` and `STATUS` is zero, the modified values of the current row are kept in the program array.

• If the `INT_FLAG` global variable is `TRUE` or `STATUS` is different from zero, the old values of the current row are restored in the program array.

The `DISPLAY ARRAY` dialog always resets `INT_FLAG` to `FALSE` and `STATUS` to zero before executing the user code of the `ON UPDATE` block.

If needed, the `ON UPDATE` handler can be configured with action attributes by added an `ATTRIBUTES()` clause, as with user-defined action handlers:

```
ON UPDATE ATTRIBUTES(TEXT=%"custlist.delete", IMAGE="listdel")
```

Related concepts

Record input (INPUT) on page 1482
The `INPUT` instruction provides single record input control in an application form.

ON INSERT block on page 1517
ON APPEND block on page 1516
ON DELETE block on page 1519

ON DELETE block
The `ON DELETE` trigger can be used to enable row deletion during a `DISPLAY ARRAY` dialog. If this block is defined, the dialog will automatically create the delete action. This action can be decorated, enabled and disabled as regular actions.

If the dialog defines an `ON ACTION delete` interaction block and the `ON DELETE` block is used, the compiler will stop with error -8408.

When the user fires the delete action, the dialog executes the user code of the `ON DELETE` block.

The dialog handles only the row deletion action and navigation, you can typically program a validation dialog box to let the user confirm the deletion. The current row in the program array is identified with `arr_curr()`:

```
DISPLAY ARRAY arr TO sr.*
...
ON DELETE
  IF fgl_winQuestion("Delete",
    "Do you want to delete this record?",
    "yes", "no|yes", "help", 0) == "no"
  THEN
    LET int_flag = TRUE
  END IF
...
```

After the user code is executed, the dialog gets the control back and processes the current row as follows:

• If the `INT_FLAG` global variable is `FALSE` and `STATUS` is zero, the current row is deleted from the program array, and the `BEFORE ROW` block is executed for the next row in the list.

• If the `INT_FLAG` global variable is `TRUE` or `STATUS` is different from zero, the current row is kept in the program array, and the `BEFORE ROW` block is executed again for the current row.

The `DISPLAY ARRAY` dialog always resets `INT_FLAG` to `FALSE` and `STATUS` to zero before executing the user code of the `ON DELETE` block.

If needed, the `ON DELETE` handler can be configured with action attributes by adding an `ATTRIBUTES()` clause, as with user-defined action handlers:

```
ON DELETE ATTRIBUTES(TEXT=%"custlist.delete", IMAGE="listdel")
```
ON SELECTION CHANGE block

The ON SELECTION CHANGE trigger can be used to enable multi-row selection and detect when rows are selected or de-selected by the end user during a DISPLAY ARRAY dialog. If this block is defined, multi-row selection is automatically enabled. However, the feature can be enabled/disabled with the setSelectionMode() dialog method.

Related concepts

Multiple row selection on page 1851
Multiple row selection allows the end user to select several rows within a list of records.

ON SORT block

Basics

The ON SORT interaction block can be used to detect when rows have to be sorted in a DISPLAY ARRAY or INPUT ARRAY dialog.

ON SORT is used in two different contexts:

1. In a regular full-list DISPLAY ARRAY / INPUT ARRAY dialog, the ON SORT trigger can be used to detect that a list sort was performed.
2. In a DISPLAY ARRAY using paged mode (ON FILL BUFFER), use ON SORT to detect a sort request from the user and re-fetch the rows from the database in the required order.

ON SORT in regular full-list DISPLAY ARRAY or INPUT ARRAY

In a regular DISPLAY ARRAY / INPUT ARRAY dialog not using paged mode, the ON SORT trigger can be used to detect that a list sort was performed.

When the ON SORT block executes in this context, the (visual) sort is already done by the runtime system and the ON SORT block is only used to execute post-sort tasks, such as displaying current row information.

To display the row position information, use the arrayToVisualIndex() dialog method to convert the current program row number to the visual row number:

```dialog
DISPLAY ARRAY arr TO sr.* ...
... 
ON SORT
  MESSAGE SFMT( "Row: %1/%2", 
    DIALOG.arrayToVisualIndex( "sr", DIALOG.getCurrentRow("sr") ),
    DIALOG.getArrayLength( "sr" )
  )
... 
```

If needed, you can get the sort column and sort order with the getSortKey() and isSortReverse() dialog methods:

```dialog
DISPLAY ARRAY arr TO sr.* ...
... 
ON SORT
  MESSAGE SFMT( "Sort on %1, %2 order", 
    DIALOG.getSortKey("sr"),
    IIF( DIALOG.isSortReverse("sr"), "descending", "ascending" )
  )
```

If needed, you can get the sort column and sort order with the getSortKey() and isSortReverse() dialog methods:
ON SORT in DISPLAY ARRAY using the paged mode

In a DISPLAY ARRAY implementing paged mode with ON FILL BUFFER trigger, built-in row sorting is not available because data is provided by pages.

Use the ON SORT trigger, to detect a sort request and perform a new SQL query to re-order the rows. In this context, the sort column and sort order are available with the getSortKey() and isSortReverse() dialog methods:

```
DEFINE key STRING, rev BOOLEAN

DISPLAY ARRAY arr TO sr.* ...
... 
ON SORT
   -- Re-execute the SQL statement to fill the page of rows in ON FILL BUFFER
   -- Assuming that form field names match table column names
   LET key = DIALOG.getSortKey("sa")
   LET rev = DIALOG.isSortReverse("sa")
   IF key IS NULL THEN
      CALL execute_sql( NULL )
   ELSE
      CALL execute_sql( "ORDER BY " || key || IIF(rev," DESC"," " ) )
   END IF
```

See Paged mode of DISPLAY ARRAY on page 1843 for more details about the paged mode in DISPLAY ARRAY and how to implement sort in this type of record list dialog.

Related concepts
List ordering on page 1862
List controllers implement a built-in sort. This feature can be disabled if not required.

ON DRAG_START block

The ON DRAG_START block is executed when the end user begins the drag operation. If this dialog trigger has not been defined, default dragging is enabled for this dialog.

In the ON DRAG_START block, the program typically specifies the type of drag & drop operation by calling ui.DragDrop.setOperation() with "move" or "copy". This call will define the default and unique drag operation. If needed, the program can allow another type of drag operation with ui.DragDrop.addPossibleOperation(). The end user can then choose to move or copy the dragged object, if the drag & drop target allows it.

If the dragged object can be dropped outside the program, the MIME type and drag/drop data must be defined with ui.DragDrop.setMimeType() and ui.DragDrop.setBuffer() methods.

Example:

```
DEFINE dnd ui.DragDrop
... 
DISPLAY ARRAY arr TO sr.* ...
... 
ON DRAG_START (dnd)
   CALL dnd.setOperation("move") -- Move is the default operation
   CALL dnd.addPossibleOperation("copy") -- User can toggle to copy if needed
   CALL dnd.setMimeType("text/plain")
   CALL dnd.setBuffer(arr[arr_curr()].cust_name)
... 
END DISPLAY
```
Related concepts
Handle drag & drop data with MIME types on page 1919
How to handle MIME types with drag & drop?
The DragDrop class on page 2570
The ui.DragDrop class is used to control the events related to drag & drop events.
Drag & drop on page 1917
Explains programming techniques for the drag & drop feature.

ON DRAG_FINISHED block
Execution of the ON DRAG_FINISHED block notifies the dialog where the drag started and that the drop operation has been completed or terminated.

Call ui.DragDrop.getOperation() to get the final type of operation of the drop. On successful completion, the method returns "move" or "copy"; otherwise, the function returns NULL. If NULL is returned, the ON DRAG_FINISHED trigger can be ignored.

In cases of successful moves to a target out of the current DISPLAY ARRAY, the application must remove the transferred data from the source model. For example, if a row was moved from dialog A to B, dialog A will get an ON DRAG_FINISHED execution after the row was dropped into B, which removes the row from the list A.

The ON DRAG_FINISHED interaction block is optional.

```
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr TO sr.* ...
...
ON DRAG_START (dnd)
  LET last_dragged_row = arr_curr()
...
ON DRAG_FINISHED (dnd)
  IF dnd.getOperation() == "move" THEN
    CALL DIALOG.deleteRow(last_dragged_row)
  END IF
...
END DISPLAY
```

Related concepts
The DragDrop class on page 2570
The ui.DragDrop class is used to control the events related to drag & drop events.
Drag & drop on page 1917
Explains programming techniques for the drag & drop feature.

ON DRAG_ENTER block
When the ON DROP control block is defined, the ON DRAG_ENTER block will be executed when the mouse cursor enters the visual boundaries of the drop target dialog. Entering the target dialog is accepted by default if no ON DRAG_ENTER block is defined. However, when ON DROP is defined, it is recommended that you also define ON DRAG_ENTER to prevent the drop of objects with an unsupported MIME type coming from other applications.

The program can decide to disallow or allow a specific drop operation with a call to ui.DragDrop.setOperation(): passing a NULL to the method will prevent the drop.

To check what MIME type is available in the drag & drop buffer, the program uses the ui.DragDrop.selectMimeType() method. This method takes the MIME type as a parameter and returns TRUE if the passed MIME type is used. You can call this method several times to check the availability of different MIME types.
You may also define the visual effect when hovering over the target list with `ui.DragDrop.setFeedback()`.

```plaintext
DEFINE dnd ui.DragDrop
... DISPLAY ARRAY arr TO sr.* ...
... ON DRAG_ENTER (dnd)
  IF dnd.selectMimeType("text/plain") THEN
    CALL dnd.setOperation("copy")
    CALL dnd.setFeedback("all")
  ELSE
    CALL dnd.setOperation(NULL)
  END IF
ON DROP (dnd)
... END DISPLAY
```

Once the mouse has entered the target area, subsequent mouse cursor moves can be detected with the `ON DRAG_OVER` trigger.

When using a table or tree-view as drop target, you can control the visual effect when the mouse moves over the rows, depending on the type of drag & drop you want to achieve.

Basically, a dragged object can be:

1. Inserted in between two rows (visual effect must show where the object will be inserted)
2. Copied/merged to the current row (visual effect must show the row under the mouse)
3. Dropped somewhere on the target widget (the exact location inside the widget does not matter)

The visual effect can be defined with the `ui.DragDrop.setFeedback()` method, typically called in the `ON DRAG_ENTER` block.

The values to pass to the `setFeedback()` method to get the desired visual effects described are respectively:

1. `insert` (default)
2. `select`
3. `all`

```plaintext
DEFINE dnd ui.DragDrop
... DISPLAY ARRAY arr TO sr.* ...
... ON DRAG_ENTER (dnd)
  IF canDrop() THEN
    CALL dnd.setOperation(NULL)
  ELSE
    CALL dnd.setFeedback("select")
  END IF
... END DISPLAY
```

### Related concepts

- **The DragDrop class** on page 2570
  The `ui.DragDrop` class is used to control the events related to drag & drop events.

- **Drag & drop** on page 1917
  Explains programming techniques for the drag & drop feature.

#### `ON DRAG_OVER` block

When the `ON DROP` control block is defined, the `ON DRAG_OVER` block will be executed after `ON DRAG_ENTER`, when the mouse cursor is moving over the drop target, or when the drag & drop operation has changed (toggling copy/move).
ON DRAG_OVER will be called only once per row, even if the mouse cursor moves over the row.

In the ON DRAG_OVER block, the method `ui.DragDrop.getLocationRow()` returns the index of the row in the target array, and can be used to allow or disallow the drop. When using a tree-view, you must also check the index returned by the `ui.DragDrop.getLocationParent()` method to detect if the object was dropped as a sibling or as a child node, and allow/disallow the drop operation accordingly.

The program can change the drop operation at any execution of the ON DRAG_OVER block. You can disallow or allow a specific drop operation with a call to `ui.DragDrop.setOperation()`; passing a NULL to the method will disallow the drop.

The current operation (returned by `ui.DragDrop.getOperation()`) is the value set in previous ON DRAG_ENTER or ON DRAG_OVER events, or the operation selected by the end user, if it can toggle between copy and move. Thus, ON DRAG_OVER can occur even if the mouse position has not changed.

If dropping has been prevented with `ui.DragDrop.setOperation(NULL)` in the previous ON DRAG_OVER event, the program can reset the operation to allow a drop with a call to `ui.DragDrop.setOperation()` with the operation parameter "move" or "copy".

ON DRAG_OVER will not be called if drop has been disabled in ON DRAG_ENTER with `ui.DragDrop.setOperation(NULL)`.

ON DRAG_OVER is optional, and must only be defined if the operation or the acceptance of the drag object depends on the target row of the drop target.

```plaintext
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr TO sr.* ...
...
ON DRAG_ENTER (dnd)
...
ON DRAG_OVER (dnd)
  IF arr[dnd.getLocationRow()].acceptsCopy THEN
    CALL dnd.setOperation("copy")
  ELSE
    CALL dnd.setOperation(NULL)
  END IF
ON DROP (dnd)
...
END DISPLAY
```

During a drag & drop process, the end user (or the target application) can decide to modify the type of the operation, to indicate whether the dragged object has to be copied or moved from the source to the target. For example, in a typical file explorer, by default files are moved when doing a drag & drop on the same disk. To make a copy of a file, you must press the Ctrl key while doing the drag & drop with the mouse.

In the drop target dialog, you can detect such operation changes in the ON DRAG_OVER trigger and query the `ui.DragDrop` object for the current operation with `ui.DragDrop.getOperation()`. In the drag source dialog, you typically check `ui.DragDrop.getOperation()` in the ON DRAG_FINISHED trigger to know what type of operation occurred, to keep ("copy" operation) or delete ("move" operation) the original dragged object.

This example tests the current operation in the drop target list and displays a message accordingly:

```plaintext
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr TO sr.* ...
...
ON DRAG_ENTER (dnd)
...
ON DRAG_OVER (dnd)
  CASE dnd.getOperation()
    WHEN "move"
      MESSAGE "The object will be moved to row ", dnd.getLocationRow()
  END CASE
ON DROP (dnd)
...
END DISPLAY
```
WHEN "copy"
  MESSAGE "The object will be copied to row ", dnd.getLocationRow()
END CASE

... ON DROP (dnd)
... END DISPLAY

Related concepts

The DragDrop class on page 2570
The `ui.DragDrop` class is used to control the events related to drag & drop events.

Tree views on page 1889
Describes how to implement tree views.

Drag & drop on page 1917
Explains programming techniques for the drag & drop feature.

**ON DROP block**

To enable drop actions on a list, you must define the `ON DROP` block; otherwise the list will not accept drop actions.

The `ON DROP` block is executed after the end user has released the mouse button to drop the dragged object. `ON DROP` will not occur if drop has been disallowed in the previous `ON DRAG_OVER` event or in `ON DRAG_ENTER` with a call to `ui.DragDrop.setOperation(NULL)`.

When `ON DROP` executes, the MIME type of the dragged object can be checked with `ui.DragDrop.getSelectedMimeType()`. Then call the `ui.DragDrop.getBuffer()` method to retrieve drag & drop data from external applications.

Ideally, the drop operation is accepted (there is no need for additional calls to `ui.DragDrop.setOperation()`).

In this block, the `ui.DragDrop.getLocationRow()` method returns the index of the row in the target array, and can be used to execute the code to get the drop data / object into the row that has been chosen by the user.

```
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr TO sr.* ...
...
  ON DROP (dnd)
    LET arr[dnd.getLocationRow()].capacity == dnd.getBuffer()
  ...
END DISPLAY
```

If the drag & drop operations are local to the same list or tree-view controller, you can use the `ui.DragDrop.dropInternal()` method to simplify the code. This method implements the typical move of the dragged rows or tree-view node. This is especially useful in case of a tree-view, but is also the preferred way to move rows around in simple tables.

This `ON DROP` code example uses the `dropInternal()` method:

```
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr_tree TO sr_tree.* ...
...
  ON DROP (dnd)
    CALL dnd.dropInternal()
  ...
END DISPLAY
```

If you want to implement by hand the code to drop a node in a tree-view, you must check the index returned by the `ui.DragDrop.getLocationParent()` method to detect if the object was dropped as a `sibling` or as
a child node, and execute the code corresponding to the drop operation. If the drop target row index returned by
getLocationRow() is a child of the parent row index returned by getLocationParent(), the new row
must be inserted before getLocationRow(); otherwise the new row must be added as a child of the parent node
identified by getLocationParent().

Related concepts
The DragDrop class on page 2570
The ui.DragDrop class is used to control the events related to drag & drop events.

Drag & drop on page 1917
Explains programming techniques for the drag & drop feature.

DISPLAY ARRAY control instructions
CONTINUE DISPLAY instruction
CONTINUE DISPLAY skips all subsequent statements in the current control block and gives the control back to the
dialog.

The CONTINUE DISPLAY instruction is useful when program control is nested within multiple conditional
statements, and you want to return the control to the dialog. If this instruction is called in a control block that is not
AFTER DISPLAY, further control blocks might be executed depending on the context.

Actually, CONTINUE DISPLAY just instructs the dialog to continue as if the code in the control block was
terminated (it's a kind of GOTO end_of_control_block). However, when executed in AFTER DISPLAY, the
focus returns to the current row in the list, giving the user another chance to browse and select a row. In this case the
BEFORE ROW of the current row will be invoked.

Related concepts
ACCEPT DISPLAY instruction on page 1526
EXIT DISPLAY instruction on page 1526

EXIT DISPLAY instruction
Use the EXIT DISPLAY to terminate the DISPLAY ARRAY instruction and resume the program execution at the
instruction immediately following the DISPLAY ARRAY block.

Note: The INT_FLAG will not be set to TRUE as when the cancel action is fired.

Related concepts
ACCEPT DISPLAY instruction on page 1526
CONTINUE DISPLAY instruction on page 1526

ACCEPT DISPLAY instruction
The ACCEPT DISPLAY instruction validates the DISPLAY ARRAY instruction and exits the dialog block.
The AFTER DISPLAY control block will be executed. Statements after ACCEPT DISPLAY will not be executed.

Related concepts
EXIT DISPLAY instruction on page 1526
CONTINUE DISPLAY instruction on page 1526

Examples
DISPLAY ARRAY dialog usage examples.
Example 1: DISPLAY ARRAY using full list mode
Form definition file "custlist.per":

```
SCHEMA shop

LAYOUT TABLE
{
```
### Program source code:

```
SCHEMA shop

MAIN

    DEFINE cnt INTEGER
    DEFINE arr DYNAMIC ARRAY OF RECORD LIKE customer.*

    DATABASE shop

    OPEN FORM f1 FROM "custlist"
    DISPLAY FORM f1

    DECLARE c1 CURSOR FOR
        SELECT id, fname, lname FROM customer
    LET cnt = 1
    FOREACH c1 INTO arr[cnt].*
        LET cnt = cnt + 1
    END FOREACH
    CALL arr.deleteElement(cnt)

    DISPLAY ARRAY arr TO srec.*
    ON ACTION print
        DISPLAY "Print a report"
    END DISPLAY

END MAIN

Example 2: DISPLAY ARRAY using paged mode

Form definition file "custlist.per":
```

```
```
Id       Name         LastName
[f001    |f002        |f003        ]
[f001    |f002        |f003        ]
[f001    |f002        |f003        ]
[f001    |f002        |f003        ]
[f001    |f002        |f003        ]
[f001    |f002        |f003        ]

END

TABLES
customer
END

ATTRIBUTES
  f001 = customer.id;
  f002 = customer.fname;
  f003 = customer.lname;
END

INSTRUCTIONS
  SCREEN RECORD srec(customer.*);
END

Program source code:

SCHEMA shop

MAIN

  DEFINE arr DYNAMIC ARRAY OF RECORD LIKE customer.*
  DEFINE cnt, ofs, len, row, i INTEGER

  DATABASE shop

  OPEN FORM f1 FROM "custlist"
  DISPLAY FORM f1

  DECLARE c1 SCROLL CURSOR FOR
    SELECT id, fname, lname FROM customer
  OPEN c1
  DISPLAY ARRAY arr TO srec.* ATTRIBUTES(COUNT=-1)
  ON FILL BUFFER
    LET ofs = fgl_dialog_getBufferStart()
    LET len = fgl_dialog_getBufferLength()
    LET row = ofs
    FOR i=1 TO len
      FETCH ABSOLUTE row c1 INTO arr[i].*
      IF SQLCA.SQLCODE!=0 THEN
        CALL DIALOG.setArrayLength("srec",row-1)
        EXIT FOR
      END IF
      LET row = row + 1
    END FOR
  AFTER DISPLAY
    IF NOT int_flag THEN
      DISPLAY "Selected customer is #"
        || arr[arr_curr()-ofs+1].id
    END IF
  END DISPLAY
END MAIN
Example 3: DISPLAY ARRAY using modification triggers

Form definition file "custlist.per":

```plaintext
SCHEMA shop

LAYOUT
TABLE
{
  Id       Name         LastName
[f001    |f002        |f003        ]
[f001    |f002        |f003        ]
[f001    |f002        |f003        ]
[f001    |f002        |f003        ]
[f001    |f002        |f003        ]
[f001    |f002        |f003        ]
}
END
END TABLES
customer
END

ATTRIBUTES
  f001 = customer.id;
  f002 = customer.fname;
  f003 = customer.lname;
END

INSTRUCTIONS
  SCREEN RECORD srec(customer.*);
END
```

Program source code:

```plaintext
SCHEMA shop

MAIN

DEFINE arr DYNAMIC ARRAY OF RECORD LIKE customer.*
DEFINE cnt, ofs, len, row, i INTEGER

DATABASE shop

OPEN FORM f1 FROM "custlist"
DISPLAY FORM f1

DECLARE c1 CURSOR FOR
  SELECT id, fname, lname FROM customer
LET cnt = 1
FOREACH c1 INTO arr[cnt].*
  LET cnt = cnt + 1
END FOREACH
CALL arr.deleteElement(cnt)

DISPLAY ARRAY arr TO srec.* ATTRIBUTES(UNBUFFERED)
  ON UPDATE
    INPUT arr[arr_curr()].* WITHOUT DEFAULTS FROM srec[scr_line()].*
  ON INSERT
    INPUT arr[arr_curr()].* FROM srec[scr_line()].*
  ON APPEND
```
Example 4: DISPLAY ARRAY with structured array

Database table definition:

```sql
CREATE TABLE items
(
  it_num INTEGER NOT NULL PRIMARY KEY,
  it_code CHAR(5) NOT NULL UNIQUE,
  it_desc VARCHAR(200) NOT NULL
);

INSERT INTO items VALUES ( 432, "XB345", "Core piece AAC" );
INSERT INTO items VALUES ( 832, "AF445", "Left wheel 2cm" );
INSERT INTO items VALUES ( 833, "AF446", "Right wheel 2cm" );
INSERT INTO items VALUES ( 512, "EE111", "Top cover 123cm" );
INSERT INTO items VALUES ( 513, "EE121", "Top cover 50cm" );
```

Form definition file "itemlist.per":

```plaintext
SCHEMA shop
LAYOUT
TABLE
{
  Code  Description
  [c1   |c2                     ]
  [c1   |c2                     ]
  [c1   |c2                     ]
}
END
END
TABLES
items
END

ATTRIBUTES
PHANTOM items.it_num;
c1 = items.it_code, IMAGECOLUMN=it_image;
c2 = items.it_desc;
PHANTOM FORMONLY.it_image;
PHANTOM FORMONLY.it_count;
END

INSTRUCTIONS
SCREEN RECORD sr
(
  items.it_num,
  items.it_code,
  items.it_desc,
  FORMONLY.it_image,
```
The "shop.sch" schema file:

```
items^it_num^258^4^1^
items^it_code^256^5^2^
items^it_desc^269^200^3^
```

Program source code:

**Note:** The `a_items` array is defined with an `item_data` sub-record that is defined for a database table.

```
SCHEMA shop

DEFINE a_items DYNAMIC ARRAY OF RECORD
  item_data RECORD LIKE items.*,
  it_image STRING,
  it_count INTEGER
END RECORD

MAIN

DEFINE x INTEGER

DATABASE test1

OPEN FORM f1 FROM "itemlist"
DISPLAY FORM f1

DECLARE c1 CURSOR FOR
  SELECT * FROM items ORDER BY it_code
LET x=1
FOREACH c1 INTO a_items[x].item_data.*
  LET a_items[x].it_image = "smiley"
  LET a_items[x].it_count = x * 5
  LET x=x+1
END FOREACH

CALL a_items.deleteElement(x)

DISPLAY ARRAY a_items TO sr.* ATTRIBUTES(UNBUFFERED)
BEFORE ROW
  MESSAGE SFMT("Item count: %1",a_items[arr_curr()].it_count)
END DISPLAY

END MAIN
```

**Editable record list (INPUT ARRAY)**

The INPUT ARRAY instruction provides always-editable record list handling in an application form.

**Understanding the INPUT ARRAY instruction**

The INPUT ARRAY dialog controls a list of records than can by directly edited.

INPUT ARRAY is designed to browse and modify a list of record, binding a static or dynamic array model to a screen array of the current displayed form.

**Important:** This feature is not supported on mobile platforms.

An INPUT ARRAY instruction supports additional features, built-in sort and search, multi-row selection and list modification triggers. For a detailed description of these features, see Table views on page 1853.
Use the INPUT ARRAY instruction to let the end user update, delete and create new records in a list, after fetching a result set from the database. The result set is produced with a database cursor executing a SELECT statement. The SELECT SQL statement is usually completed at runtime with a WHERE clause produced from a CONSTRUCT dialog.

The INPUT ARRAY instruction associates a program array of records with a screen-array defined in a form so that the user can update the list of records. The INPUT ARRAY statement activates the current form (the form that was most recently displayed or the form in the current window).

During the INPUT ARRAY execution, the user can edit or delete existing rows, insert new rows, and move inside the list of records. The program controls the behavior of the instruction with dialog control blocks such as BEFORE DELETE, BEFORE INSERT, etc.

When the user clicks on an action view (button), or when an asynchronous event occurs, dialog interaction blocks like ON ACTION are executed.

The code inside an INPUT ARRAY dialog can use control instructions, dialog control functions, and the ui.Dialog class, to implement the dialog behavior.

To terminate the INPUT ARRAY execution, the user can validate (or cancel) the dialog to commit (or invalidate) the modifications made in the list of records.

When the statement completes execution, the program must test the INT_FLAG on page 569 variable to check if the dialog was validated (or canceled) and then use INSERT, DELETE, or UPDATE SQL statements to modify the appropriate database tables. The database can also be updated during the execution of the INPUT ARRAY statement.

Related concepts
Dialog programming basics on page 1706
This section describes basic dialog programming concepts.

Syntax of INPUT ARRAY instruction
The INPUT ARRAY supports data entry by users into a screen array and stores the entered data in an array of records.

Syntax

```
INPUT ARRAY array
  \[- WITHOUT DEFAULTS \]
  FROM screen-array.*
  \[- ATTRIBUTES ( \[- \display-attribute \]
      \[- control-attribute \]
  ) \-
  \[- HELP help-number \]
  \[- dialog-control-block \]
  \[- END INPUT \]
```

where dialog-control-block is one of:

```
\[- BEFORE INPUT \]
\[- AFTER INPUT \]
\[- AFTER DELETE \]
\[- BEFORE ROW \]
\[- AFTER ROW \]
\[- BEFORE FIELD field-spec \[- \[- \]
\[- AFTER FIELD field-spec \[- \[- \]
\[- ON ROW CHANGE \]
\[- ON CHANGE field-spec \[- \[- \]
\[- ON IDLE seconds \]
\[- ON TIMER seconds \]
\[- ON ACTION action-name \]
  \[- INFIELD field-spec \]
  \[- ATTRIBUTES ( action-attributes-input-array ) \]
```
ON KEY (key-name [, ...])
BEFORE INSERT
AFTER INSERT
BEFORE DELETE

dialog-statement

where action-attributes-input-array is:

TEXT = string
COMMENT = string
IMAGE = string
ACCELERATOR = string
DEFAULTVIEW = YES | NO | AUTO
VALIDATE = NO
CONTEXTMENU = YES | NO | AUTO
ROWBOUND

where dialog-statement is one of:

statement
ACCEPT INPUT
CONTINUE INPUT
EXIT INPUT
NEXT FIELD CURRENT | NEXT | PREVIOUS field-spec
CANCEL DELETE
CANCEL INSERT

where field-spec identifies a unique field with one of:

field-name
table-name.field-name
screen-array.field-name
screen-record.field-name

where display-attribute is:

BLACK | BLUE | CYAN | GREEN
MAGENTA | RED | WHITE | YELLOW
BOLD | DIM | INVISIBLE | NORMAL
REVERSE | BLINK | UNDERLINE

where control-attribute is:

ACCEPT = boolean
APPEND ROW = boolean
AUTO APPEND = boolean
CANCEL = boolean
COUNT = row-count
DELETE ROW = boolean
FIELD ORDER FORM
HELP = help-number
INSERT ROW = boolean
KEEP CURRENT ROW = boolean
MAXCOUNT = max-row-count
UNBUFFERED = boolean
WITHOUT DEFAULTS = boolean
1. `array` is the array of records that will be filled by the `INPUT ARRAY` statement.
2. `help-number` is an integer that allows you to associate a help message number with the instruction.
3. `field-name` is the identifier of a field of the current form.
4. `table-name` is the identifier of a database table of the current form.
5. `screen-record` is the identifier of a screen record of the current form.
6. `screen-array` is the screen array that will be used in the form.
7. `action-name` identifies an action that can be executed by the user.
8. `seconds` is an integer literal or variable that defines a number of seconds.
9. `key-name` is a hot-key identifier (like `F11` or `Control-z`).
10. `statement` is any instruction supported by the language.
11. `row-count` defines the initial number of rows for a static array.
12. `max-row-count` is the maximum number of rows that can be created.
13. `boolean` is a boolean expression that evaluates to `TRUE` or `FALSE`.
14. `action-attributes` are dialog-specific action attributes.

**INPUT ARRAY programming steps**

Follow this procedure to use the `INPUT ARRAY` dialog instruction.

To implement an `INPUT ARRAY` statement:

1. Create a form specification file containing a screen array. The screen array identifies the presentation elements to be used by the runtime system to display the rows.
2. Make sure that the program controls interruption handling with `DEFER INTERRUPT`, to manage the validation/cancellation of the interactive dialog.
3. Define an array of records with the `DEFINE` instruction. The members of the program array must correspond to the elements of the screen array, by number and data types. If you want to input data from a reduced set of columns, you must define a second screen array, containing the limited list of form fields, in the form file. You can then use the second screen array in an `INPUT ARRAY a FROM sa.*` instruction.
4. Open and display the form, using an `OPEN WINDOW WITH FORM` or the `OPEN FORM/DISPLAY FORM` instructions.
5. If needed, fill the program array with data, for example with a result set cursor, counting the number of program records being filled with retrieved data.
6. Set the `INT_FLAG` variable to `FALSE`.
7. Write the `INPUT ARRAY` dialog block to handle data input.
8. Inside the `INPUT ARRAY` dialog, control the behavior of the instruction with control blocks such as `BEFORE INPUT`, `BEFORE INSERT`, `BEFORE DELETE`, `BEFORE ROW`, `BEFORE FIELD`, `AFTER INSERT`, `AFTER DELETE`, `AFTER FIELD`, `AFTER ROW`, `AFTER INPUT` and `ON ACTION` blocks.
9. Get the new number of rows with the `ARR_COUNT()` built-in function or with `DIALOG.getArrayLength()`.
10. After the interaction statement block, test the `INT_FLAG` predefined variable to check if the dialog was canceled (`INT_FLAG=TRUE`) or validated (`INT_FLAG=FALSE`). If the `INT_FLAG` variable is `TRUE`, it is recommended that you reset it to `FALSE` to not disturb code that relies on this variable to detect interruption events from the GUI front-end or TUI console.

**Related concepts**

Form specification files on page 1237
Form specification files are the source files defining the layout and content of application forms.

**Using editable record lists**
Dialog coding concepts, configuration and code structure.

**Variable binding in INPUT ARRAY**
The `INPUT ARRAY` statement binds the members of the `array` of record to the `screen array` fields specified with the `FROM` keyword. Array members and screen array fields are bound by position (not by name). The number of members in the program array must match the number of fields in the screen record (that is, in a single row of the screen array).

```plaintext
SCHEMA stock
DEFINE cust_arr DYNAMIC ARRAY OF customer.*
...
INPUT ARRAY cust_arr FROM sr.*
    ATTRIBUTES(UNBUFFERED)
...
END INPUT
```

The array is usually defined with a flat list of members with `ARRAY OF RECORD / END RECORD`. However, the array can be structured with sub-records and still be used with an `INPUT ARRAY` dialog. This is especially useful when you need to define arrays from database tables, and additional information needs to be managed at runtime (for example to hold image resource for each row, to be displayed with the `IMAGECOLUMN` attribute):

```plaintext
SCHEMA shop
DEFINE a_items DYNAMIC ARRAY OF RECORD
    item_data RECORD LIKE items.*,
    it_image STRING,
    it_count INTEGER
END RECORD
...
INPUT ARRAY a_items FROM sr.*
...
```

When using a static array, the initial number of rows is defined by the `COUNT` attribute and the size of the array determines how many rows can be inserted. When using a dynamic array, the initial number of rows is defined by the number of elements in the dynamic array (the `COUNT` attribute is ignored), and the maximum rows is unlimited. For both static and dynamic arrays, the maximum number of rows the user can enter can be defined with the `MAXCOUNT` attribute.

The `FROM` clause binds the screen records in the screen array to the program records of the program array. The form can include other fields that are not part of the specified screen array, but the number of member variables in each record of the program array must equal the number of fields in each row of the screen array. When the user enters data, the runtime system checks the entered value against the data type of the variable, not the data type of the screen field.

The variables of the record array are the interface to display data or to get the user input through the `INPUT ARRAY` instruction. Always use the variables if you want to change some field values by program. When using the `UNBUFFERED` attribute, the instruction is sensitive to program variable changes. If you need to display new data during the `INPUT ARRAY` execution, use the `UNBUFFERED` attribute and assign the values to the program array row; the runtime system will automatically display the values to the screen:

```plaintext
INPUT ARRAY p_items FROM s_items.*
    ATTRIBUTES(UNBUFFERED)
    ON CHANGE code
        IF p_items[arr_curr()].code = "A34" THEN
            LET p_items[arr_curr()].desc = "Item A34"
        END IF
    END ON
END INPUT
```
The runtime system adapts input and display rules to the data type of the array record members. If a member is declared with the `DEFINE LIKE` instruction and uses a column defined as `SERIAL/SERIAL8/BIGSERIAL`, the runtime system will treat the field as if it was defined with the `NOENTRY` attribute in the form file. Since values of serial columns are automatically generated by the database server, no user input is required for such fields.

The default order in which the focus moves from field to field in the screen array is determined by the declared order of the corresponding member variables, in the array of the record definition. The program `OPTIONS` instruction can also change the behavior of the `INPUT ARRAY` instruction, with the `INPUT WRAP` or `FIELD ORDER FORM` options. For more details, see Defining the tabbing order on page 1729.

By default the `INPUT ARRAY` instruction clears the program array when starting, unless you specify the `WITHOUT DEFAULTS` keywords or option. With this option, the dialog displays the program array rows in the screen fields. Unlike the `INPUT` dialog, the column default values defined in the form specification file with the `DEFAULT` attribute or in the `database schema files` are always used when a new row is inserted in the list.

If the program array has the same structure as a database table (this is the case when the array is defined with a `DEFINE LIKE` clause), you may not want to display/use some of the columns. You can achieve this by using `PHANTOM` fields in the screen array definition. Phantom fields will only be used to bind program variables, and will not be transmitted to the front-end for display.

**Related concepts**

- Binding variables to form fields on page 1715
- Some dialogs need program variables to store form field values.

**INPUT ARRAY instruction configuration**

This section describes the options that can be specified in the `ATTRIBUTES` clause of the `INPUT ARRAY` instruction. The options of the `ATTRIBUTES` clause override all default attributes and temporarily override any display attributes that the `OPTIONS` or the `OPEN WINDOW` statement specified for these fields. With the `INPUT ARRAY` statement, the `INVISIBLE` attribute is ignored.

**HELP option**

The `HELP` clause specifies the number of a help message to display if the user invokes the help the `INPUT ARRAY` dialog. The predefined 'help' action is automatically created by the runtime system. You can bind action views to the 'help' action.

The `HELP` clause overrides the `HELP` attribute.

**WITHOUT DEFAULTS option**

The `WITHOUT DEFAULTS` clause defines whether the program array elements are populated (and to be displayed) when the dialog begins. Once the dialog is started, existing rows are always handled as records to be updated in the database (i.e. `WITHOUT DEFAULTS=FALSE`), while newly-created rows are handled as records to be inserted in the database (i.e. `WITHOUT DEFAULTS=TRUE`). In other words, the `REQUIRED` and `DEFAULT` attributes defined in the form are only used for newly-created rows.

It is unusual to implement an `INPUT ARRAY` with no `WITHOUT DEFAULTS` option, because the data of the program variables would be cleared and the list empty. So, you typically use the `WITHOUT DEFAULTS=FALSE` clause in `INPUT ARRAY`. In a singular `INPUT ARRAY`, the default is `WITHOUT DEFAULTS=FALSE`.

**FIELD ORDER FORM option**

By default, the form tabbing order is defined by the variable list in the binding specification. You can control the tabbing order by using the `FIELD ORDER FORM` attribute. When this attribute is used, the tabbing order is defined by the `TABINDEX` attribute of the form items. With `FIELD ORDER FORM`, if you jump from one field to another with the mouse, the `BEFORE FIELD/AFTER FIELD` triggers of intermediate fields are not executed (actually, the `Dialog.fieldOrder FGLPROFILE entry` is ignored.)
If the form uses a **TABLE** container, the front-end resets the tab indexes when the user moves columns around. This way, the visual column order always corresponds to the input tabbing order. The order of the columns in an editable list can be important; you may want to freeze the table columns with the **UNMOVABLECOLUMNS** attribute.

**UNBUFFERED option**

The **UNBUFFERED** attribute indicates that the dialog must be sensitive to program variable changes. When using this option, you bypass the traditional "buffered" mode.

When using the traditional "buffered" mode, program variable changes are not automatically displayed to form fields; you need to execute a **DISPLAY TO** or **DISPLAY BY NAME**. Additionally, if an action is triggered, the value of the current field is not validated and is not copied into the corresponding program variable. The only way to get the text of the current field is to use **GET_FLDBUF()**.

If the "unbuffered" mode is used, program variables and form fields are automatically synchronized. You don't need to display explicitly values with a **DISPLAY TO** or **DISPLAY BY NAME**. When an action is triggered, the value of the current field is validated and is copied into the corresponding program variable.

**COUNT option**

The **COUNT** attribute defines the number of valid rows in the **static array** to be displayed as default rows. If you do not use the **COUNT** attribute, the runtime system cannot determine how much data to display, so the screen array remains empty. You can also use the **SET_COUNT()** built-in function, but it is supported for backward compatibility only. The **COUNT** option is ignored when using a **dynamic array**. If you specify the **COUNT** attribute, the **WITHOUT DEFAULTS** option is not required because it is implicit. If the **COUNT** attribute is greater than **MAXCOUNT**, the runtime system will take **MAXCOUNT** as the actual number of rows. If the value of **COUNT** is negative or zero, it defines an empty list.

**MAXCOUNT option**

The **MAXCOUNT** attribute defines the **maximum** number of rows that can be inserted in the program array. This attribute allows you to give an upper limit of the total number of rows the user can enter, when using both **static** or **dynamic arrays**.

When binding a **static** array, **MAXCOUNT** is used as upper limit if it is lower or equal to the actual declared static array size. If **MAXCOUNT** is greater than the array size, the size of the static array is used as the upper limit. If **MAXCOUNT** is lower than the **COUNT** attribute (or to the **SET_COUNT()** parameter), the actual number of rows in the array will be reduced to **MAXCOUNT**.

When binding a **dynamic** array, the user can enter an infinite number of rows unless the **MAXCOUNT** attribute is used. If **MAXCOUNT** is lower than the actual size of the dynamic array, the number of rows in the array will be reduced to **MAXCOUNT**.

If **MAXCOUNT** is negative or equal to zero, the user cannot insert rows.

**ACCEPT option**

The **ACCEPT** attribute can be set to **FALSE** to avoid the automatic creation of the accept default action. This option can be used for example when you want to write a specific validation procedure, by using **ACCEPT INPUT**.

**CANCEL option**

The **CANCEL** attribute can be set to **FALSE** to avoid the automatic creation of the cancel default action. This is useful for example when you only need a validation action (accept), or when you want to write a specific cancellation procedure, by using **EXIT INPUT**.

If the **CANCEL=FALSE** option is set, no **close** action will be created, and you must write an **ON ACTION close** control block to create an explicit action.
APPEND ROW option
The APPEND ROW attribute can be set to FALSE to avoid the append default action, and prevent the user adding rows at the end of the list. If APPEND ROW=FALSE, it is still possible to insert rows in the middle of the list. Use the INSERT ROW attribute to disallow the user from inserting rows. To deny automatic temporary row creation if only APPEND ROW=FALSE is used, set AUTO APPEND is not set to FALSE.

INSERT ROW option
The INSERT ROW attribute can be set to FALSE to avoid the insert default action, and prevent the user inserting new rows in the middle of the list. However, even if INSERT ROW is FALSE, it is still possible to append rows at the end of the list. Use the APPEND ROW attribute to disallow the user from appending rows. To deny automatic temporary row creation if only INSERT ROW=FALSE is used, set AUTO APPEND is not set to FALSE.

DELETE ROW option
The DELETE ROW attribute can be set to FALSE to avoid the delete default action, and prevent the user removing rows from the list.

AUTO APPEND option
By default, an INPUT ARRAY controller creates a temporary row when needed (for example, when the user deletes the last row of the list, a new row will be automatically created). You can prevent this default behavior by setting the AUTO APPEND attribute to FALSE. When this attribute is set to FALSE, the only way to create a new temporary row is to execute the append action.

If both the APPEND ROW and INSERT ROW attributes are set to FALSE, the dialog automatically behaves as if AUTO APPEND equals FALSE.

KEEP CURRENT ROW option
Depending on the list container used in the form, the current row may be highlighted during the execution of the dialog, and cleared when the instruction ends. You can change this default behavior by using the KEEP CURRENT ROW attribute, to force the runtime system to keep the current row highlighted.

Related concepts
Syntax of INPUT ARRAY instruction on page 1532
The INPUT ARRAY supports data entry by users into a screen array and stores the entered data in an array of records.

Default actions in INPUT ARRAY
When an INPUT ARRAY instruction executes, the runtime system creates a set of default actions.

Field validation occurs and different INPUT ARRAY control blocks are executed based on the invoked default action.

This table lists the default actions created for this dialog:

Table 371: Default actions for INPUT ARRAY

<table>
<thead>
<tr>
<th>Default action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>accept</td>
<td>Validates the INPUT ARRAY dialog (validates fields and leaves the dialog)</td>
</tr>
<tr>
<td></td>
<td>Creation can be avoided with ACCEPT = FALSE attribute.</td>
</tr>
<tr>
<td>cancel</td>
<td>Cancels the INPUT ARRAY dialog (no validation, INT_FLAG is set to TRUE)</td>
</tr>
<tr>
<td></td>
<td>Creation can be avoided with CANCEL = FALSE attribute.</td>
</tr>
<tr>
<td>Default action</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>close</td>
<td>By default, cancels the INPUT ARRAY dialog (no validation, INT_FLAG is set to TRUE). Default action view is hidden. See Implementing the close action on page 1772.</td>
</tr>
<tr>
<td>insert</td>
<td>Inserts a new row before current row. Creation can be avoided with INSERT ROW = FALSE attribute.</td>
</tr>
<tr>
<td>append</td>
<td>Appends a new row at the end of the list. Creation can be avoided with APPEND ROW = FALSE attribute.</td>
</tr>
<tr>
<td>delete</td>
<td>Deletes the current row. Creation can be avoided with DELETE ROW = FALSE attribute.</td>
</tr>
<tr>
<td>help</td>
<td>Shows the help topic defined by the HELP clause. Only created when a HELP clause is defined.</td>
</tr>
<tr>
<td>nextrow</td>
<td>Moves to the next row in a list displayed in one row of fields. See note (1).</td>
</tr>
<tr>
<td>prevrow</td>
<td>Moves to the previous row in a list displayed in one row of fields. See note (1).</td>
</tr>
<tr>
<td>firstrow</td>
<td>Moves to the first row in a list displayed in one row of fields. See note (1).</td>
</tr>
<tr>
<td>lastrow</td>
<td>Moves to the last row in a list displayed in one row of fields. See note (1).</td>
</tr>
<tr>
<td>find</td>
<td>Opens the fglfind dialog window to let the user enter a search value, and seeks the row matching the value. See note (2).</td>
</tr>
<tr>
<td>findnext</td>
<td>Seeks the next row matching the value entered during the fglfind dialog. See note (2).</td>
</tr>
</tbody>
</table>

Notes:

1. The action is only created with a DISPLAY ARRAY or INPUT ARRAY using a screen record bound to a set of form fields in a GRID container, and this set of fields show only a single row of the list. The action is not created when using a screen array bound to a list container such as TABLE, TREE and SCROLLGRID, or to a set of fields in a GRID container, that repeat on several lines to show more than one single row.

2. The action is only created if the context allows built-in find.

The insert, append, delete, accept and cancel default actions can be avoided with dialog control attributes:

```
INPUT ARRAY arr TO sr.* ATTRIBUTES( INSERT ROW=FALSE, CANCEL=FALSE, ... )
...
```

Related concepts

Dialog programming basics on page 1706
This section describes basic dialog programming concepts.

**INPUT ARRAY control blocks**

**INPUT ARRAY control blocks execution order**

This table shows the order in which the runtime system executes the control blocks in the INPUT ARRAY instruction, based on the user action:

**Table 372: Control block execution order for INPUT ARRAY**

<table>
<thead>
<tr>
<th>Context / User action</th>
<th>Control Block execution order</th>
</tr>
</thead>
</table>
| Entering the dialog                      | 1. BEFORE INPUT  
2. BEFORE ROW  
3. BEFORE FIELD                                                   |
| Moving to a different row from field A to field B | 1. **ON CHANGE** (if value has changed for field A)  
2. **AFTER FIELD** (for field A in the row you leave)  
3. **AFTER INSERT** (if the row you leave was inserted or appended)  
   or  
   **ON ROW CHANGE** (if values have changed in the row you leave)  
4. **AFTER ROW** (for the row you leave)  
5. BEFORE ROW (the new current row)  
6. BEFORE FIELD (for field B in the new current row) |
| Moving from field A to field B in the same row | 1. **ON CHANGE** (if value has changed for field A)  
2. **AFTER FIELD** (for field A)  
3. **BEFORE FIELD** (for field B) |
| Deleting a row                            | 1. **BEFORE DELETE** (for the row to be deleted)  
2. **AFTER DELETE** (for the deleted row)  
3. **AFTER ROW** (for the deleted row)  
4. **BEFORE ROW** (for the new current row)  
5. **BEFORE FIELD** (field in the new current row) |
| Inserting a new row between rows          | 1. **ON CHANGE** (if value has changed in the field you leave)  
2. **AFTER FIELD** (for the row you leave)  
3. **AFTER INSERT** (if the row you leave was inserted or appended)  
   or  
   **ON ROW CHANGE** (if values have changed in the row you leave)  
4. **AFTER ROW** (for the row you leave)  
5. **BEFORE INSERT** (for the new created row)  
6. **BEFORE FIELD** (for the new created row) |
<table>
<thead>
<tr>
<th>Context / User action</th>
<th>Control Block execution order</th>
</tr>
</thead>
</table>
| Appending a new row at the end | 1. ON CHANGE (if value has changed in the current field)  
2. AFTER FIELD (for the row you leave)  
3. AFTER_INSERT (if the row you leave was inserted or appended)  
or  
ON_ROW_CHANGE (if values have changed in the row you leave)  
4. AFTER_ROW (for the row you leave)  
5. BEFORE_ROW (for the new created row)  
6. BEFORE_INSERT (for the new created row)  
7. BEFORE_FIELD (for the new created row) |
| Validating the dialog | 1. ON_CHANGE  
2. AFTER_FIELD  
3. AFTER_INSERT (if the current row was inserted or appended)  
or  
ON_ROW_CHANGE (if values have changed in the current row)  
4. AFTER_ROW  
5. AFTER_INPUT |
| Canceling the dialog | 1. AFTER_ROW  
2. AFTER_INPUT |

**BEFORE INPUT block**

**BEFORE INPUT block in singular and parallel INPUT, INPUT ARRAY dialogs**

In a singular INPUT, INPUT ARRAY instruction, or when used as parallel dialog, the **BEFORE_INPUT** is only executed once when the dialog is started.

The **BEFORE INPUT** block is executed once at dialog startup, before the runtime system gives control to the user. This block can be used to display messages to the user, initialize program variables and set up the dialog instance by deactivating unused fields or actions the user is not allowed to execute.

```plaintext
INPUT BY_NAME cust_rec.* ...  
BEFORE_INPUT  
MESSAGE "Input customer information"  
CALL DIALOG.setActionActive("check_info", is_super_user() )  
CALL DIALOG.setFieldActive("cust_comment", is_super_user() )  
... 
```

The fields are initialized with the defaults values before the **BEFORE_INPUT** block is executed. When the **INPUT** instruction uses the **WITHOUT_DEFAULTS** option, the default values are taken from the program variables bound to the fields, otherwise (with defaults), the **DEFAULT** attributes of the form fields are used.

Use the **NEXT_FIELD** control instruction in the **BEFORE_INPUT** block, to jump to a specific field when the dialog starts.
BEFORE INPUT block in INPUT and INPUT ARRAY of procedural DIALOG

In an INPUT or INPUT ARRAY sub-dialog of a procedural DIALOG instruction, the BEFORE INPUT block is executed when the focus goes to a group of fields driven by the sub-dialog. This trigger is only invoked if a field of the sub-dialog gets the focus, and none of the other fields had the focus.

When the focus is in a list driven by an INPUT ARRAY sub-dialog, moving to a different row will not invoke the BEFORE INPUT block.

BEFORE INPUT is executed after the BEFORE DIALOG block and before the BEFORE ROW, BEFORE FIELD blocks.

In this example, the BEFORE INPUT block is used to set up a specific action and display a message:

```
INPUT BY NAME p_order.*
BEFORE INPUT
  CALL DIALOG.setActionActive("validate_order", TRUE)
```

Related concepts
BEFORE CONSTRUCT block on page 1573
BEFORE DISPLAY block on page 1509
AFTER INPUT block on page 1490

AFTER INPUT block

AFTER INPUT block in singular and parallel INPUT, INPUT ARRAY dialogs

In a singular INPUT, INPUT ARRAY instruction, or when used as parallel dialog, the AFTER INPUT is only executed once when dialog ends.

The AFTER INPUT block is executed after the user has validated or canceled the INPUT or INPUT ARRAY dialog with the accept or cancel default actions, or when the ACCEPT INPUT instruction is executed.

The AFTER INPUT block is not executed when the EXIT INPUT instruction is performed.

In singular and parallel dialogs, this block is typically used to implement global dialog validation rules for several fields. If the values entered by the user do not satisfy the constraints, use the NEXT FIELD instruction to force the dialog to continue. The CONTINUE INPUT instruction can be used instead of NEXT FIELD, when no particular field has to be selected.

Before checking the validation rules, make sure that the INT_FLAG variable is FALSE: because if the user cancels the dialog, the validation rules must be skipped.

```
INPUT BY NAME cust_rec.*
  WITHOUT DEFAULTS ATTRIBUTES (UNBUFFERED)
  ... 
  AFTER INPUT 
    IF NOT INT_FLAG THEN 
      IF cust_rec.cust_address IS NOT NULL 
        AND cust_rec.cust_zipcode IS NULL THEN 
          ERROR "Address is incomplete, enter a zipcode."
          NEXT FIELD zipcode 
      END IF 
    END IF 
  END INPUT 
```

To limit the validation to fields that have been modified by the end user, you can call the FIELD_TOUCHED() function or the DIALOG.getFieldTouched() method to check if a field has changed during the dialog execution. This will make your validation code faster if the user has only modified a couple of fields in a large form.
**AFTER INPUT block in INPUT and INPUT ARRAY of procedural DIALOG**

In an INPUT or INPUT ARRAY sub-dialog of a procedural DIALOG instruction, the AFTER INPUT block is executed when the focus is lost by a group of fields driven by an INPUT or INPUT ARRAY sub-dialog. This trigger is invoked if a field of the sub-dialog loses the focus, and a field of a different sub-dialog gets the focus. When the focus is in a list driven by an INPUT ARRAY sub-dialog, moving to a different row will not invoke the AFTER INPUT block.

If the focus leaves the current group and goes to an action view, this trigger is not executed, because the focus has not gone to another sub-dialog yet.

AFTER INPUT is executed after the AFTER FIELD, AFTER ROW blocks and before the AFTER DIALOG block.

Executing a NEXT FIELD in the AFTER INPUT control block will keep the focus in the group of fields. Within an INPUT ARRAY sub-dialog, NEXT FIELD will keep the focus in the list and stay in the current row. You typically use this behavior to control user input.

In this example, the AFTER INPUT block is used to validate data and disable an action that can only be used in the current group:

```plaintext
INPUT BY NAME p_order.*
AFTER INPUT
   IF NOT check_order_data(DIALOG) THEN
      NEXT FIELD CURRENT
   END IF
   CALL DIALOG.setFieldActive("validate_order", FALSE)
```

**Related concepts**

AFTER DISPLAY block on page 1510
AFTER CONSTRUCT block on page 1574
BEFORE INPUT block on page 1489

**BEFORE ROW block**

**BEFORE ROW block in singular and parallel DISPLAY ARRAY, INPUT ARRAY dialogs**

In a singular DISPLAY ARRAY, INPUT ARRAY instruction, or when used as parallel dialog, the BEFORE ROW block is executed each time the user moves to another row. This trigger can also be executed in other situations, such as when you delete a row, or when the user tries to insert a row but the maximum number of rows in the list is reached.

You typically do some dialog setup / message display in the BEFORE ROW block, because it indicates that the user selected a new row or entered in the list.

When the dialog starts, BEFORE ROW will be executed for the current row, but only if there are data rows in the array.

When called in this block, DIALOG.getCurrentRow() / arr_curr() return the index of the current row.

In this example, the BEFORE ROW block gets the new row number and displays it in a message:

```plaintext
DISPLAY ARRAY ... 
   ... 
   BEFORE ROW 
   MESSAGE "We are on row # ", arr_curr() 
   ... 
```

**BEFORE ROW block in DISPLAY ARRAY and INPUT ARRAY of procedural DIALOG**

In an INPUT or INPUT ARRAY sub-dialog of a procedural DIALOG instruction, the BEFORE ROW block is executed when a DISPLAY ARRAY or INPUT ARRAY list gets the focus, or when the user moves to another row
inside a list. This trigger can also be executed in other situations, for example when you delete a row, or when the user tries to insert a row but the maximum number of rows in the list is reached.

You typically do some dialog setup / message display in the **BEFORE ROW** block, because it indicates that the user selected a new row. Do not use this trigger to detect focus changes; Use the **BEFORE DISPLAY** or **BEFORE INPUT** blocks instead.

In **DISPLAY ARRAY**, **BEFORE ROW** is executed **after** the **BEFORE DISPLAY** block. In **INPUT ARRAY**, **BEFORE ROW** is executed **before** the **BEFORE INSERT** and **BEFORE FIELD** blocks and **after** the **BEFORE INPUT** blocks.

When the procedural dialog starts, **BEFORE ROW** will only be executed if the list has received the focus and there is a current row (the array is not empty). If you have other elements in the form which can get the focus before the list, **BEFORE ROW** will not be triggered when the dialog starts. You must pay attention to this, because this behavior is different to the behavior of singular **DISPLAY ARRAY** or **INPUT ARRAY**. In singular dialogs, the **BEFORE ROW** block is always executed when the dialog starts (and when there are rows in the array).

When called in this block, **DIALOG.getCurrentRow() / arr_curr()** return the index of the current row.

In this example the **BEFORE ROW** block displays a message with the current row number:

```
DISPLAY ARRAY p_items TO s_items.*
BEFORE ROW
  MESSAGE "We are in items, on row ", DIALOG.getCurrentRow("s_items")
```

**Related concepts**

* BEFORE INPUT block on page 1489
* BEFORE DISPLAY block on page 1509

**ON ROW CHANGE block**

The **ON ROW CHANGE** block is executed in a list controlled by an **INPUT ARRAY**, when leaving the current row and when the row has been modified since it got the focus. This is typically used to detect row modification.

The code in **ON ROW CHANGE** will not be executed when leaving new rows created by the user with the default append or insert action. To detect row creation, you must use the **BEFORE INSERT** or **AFTER INSERT** control blocks.

The **ON ROW CHANGE** block is only executed if at least one field value in the current row has changed since the row was entered, and the modification flag of the field is set. The modified field(s) may not be the current field, and several field values can be changed. Values may have been changed by the user or by the program. The modification flag is reset for all fields when entering another row, when going to another sub-dialog, or when leaving the dialog instruction.

**ON ROW CHANGE** is executed **after** the **AFTER FIELD** block and **before** the **AFTER ROW** block.

When called in this block, **DIALOG.getCurrentRow() / arr_curr()** return the index of the current row that has been changed.

You can, for example, code database modifications (**UPDATE**) in the **ON ROW CHANGE** block:

```
INPUT ARRAY p_items FROM s_items.*
...
ON ROW CHANGE
  LET r = DIALOG.getCurrentRow("s_items")
  UPDATE items SET
    items.item_code        = p_items[r].item_code,
    items.item_description = p_items[r].item_description,
    items.item_price       = p_items[r].item_price,
    items.item_updatedate  = TODAY
WHERE items.item_num = p_items[r].item_num
```

**Related concepts**

* Input field modification flag on page 1723
Each input field controlled by a dialog instruction has a modification flag.

**AFTER ROW block** on page 1511

**AFTER ROW block**

**AFTER ROW block in singular and parallel DISPLAY ARRAY, INPUT ARRAY dialogs**

In a singular DISPLAY ARRAY, INPUT ARRAY instruction, or when used as parallel dialog, the **AFTER ROW** block is executed each time the user moves to another row, before the current row is left. This trigger can also be executed in other situations, such as when you delete a row, or when the user inserts a new row.

A **NEXT FIELD** instruction executed in the **AFTER ROW** control block will keep the user entry in the current row. Use this behavior to implement row validation and prevent the user from leaving the list or moving to another row.

When called in this block, `DIALOG.getCurrentRow() / arr_curr()` returns the index of the row that you are leaving.

**AFTER ROW block in DISPLAY ARRAY and INPUT ARRAY of procedural DIALOG**

In an **INPUT** or **INPUT ARRAY** sub-dialog of a procedural DIALOG instruction, the **AFTER ROW** block is executed when a **DISPLAY ARRAY** or **INPUT ARRAY** list loses the focus, or when the user moves to another row in a list. This trigger can also be executed in other situations, for example when you delete a row, or when the user inserts a new row.

**AFTER ROW** is executed **after** the **AFTER FIELD**, **AFTER INSERT** and **before** **AFTER DISPLAY** or **AFTER INPUT** blocks.

When called in this block, `DIALOG.getCurrentRow() / arr_curr()` returns the index of the row that you are leaving.

For both **INPUT ARRAY** and **DISPLAY ARRAY** sub-dialogs, a **NEXT FIELD** executed in the **AFTER ROW** control block will keep the focus in the list and stay in the current row. Use this feature to implement row validation and prevent the user from leaving the list or moving to another row.

**AFTER ROW and temporary rows in INPUT ARRAY**

**Important:** After creating a temporary row at the end of a list driven by INPUT ARRAY, if you leave that row to go to a previous row without data input (setting the touched flag), or when the cancel action is invoked, the temporary row will be automatically removed. The **AFTER ROW** block will be executed for the temporary row, but `ui.DIALOG.getCurrentRow() / arr_curr()` will be one row greater than `ui.DIALOG.getArrayLength() / ARR_COUNT()`. In this case, it is recommended that you ignore the **AFTER ROW** event. For example, it is recommended that you avoid executing a **NEXT FIELD** or **CONTINUE INPUT** instruction, and trying to access the dynamic array with a row index that is greater than the total number of rows, otherwise the runtime system will adapt the total number of rows to the actual number of rows in the program array.

In this example, the **AFTER ROW** block checks the current row index and verifies a variable value to force the focus to stay in the current row if the value is wrong:

```plaintext
INPUT ARRAY p_items FROM s_items.*

...  
AFTER ROW
    LET r = DIALOG.getCurrentRow("s_items")
    IF r <= DIALOG.getArrayLength("s_items") THEN
        IF NOT item_is_valid_quantity(p_item[r].item_quantity) THEN
            ERROR "Item quantity is not valid"
            NEXT FIELD item_quantity
        END IF
    END IF
END IF
```
Another way to handle the case of temporary rows in **AFTER ROW** is to use a flag to know if the **AFTER INSERT** block was executed. The **AFTER INSERT** block is not executed if the temporary row is automatically removed. By setting a first value in **BEFORE INSERT** and changing the flag in **AFTER INSERT**, you can detect if the row was permanently added to the list:

```sql
INPUT ARRAY p_items FROM s_items.*
...
BEFORE INSERT
    LET op = "T"
...
AFTER INSERT
    LET op = "I"
...
AFTER ROW
    IF op == "I" THEN
        IF NOT item_is_valid_quantity(p_item[arr_curr()].item_quantity) THEN
            ERROR "Item quantity is not valid"
        NEXT FIELD item_quantity
    END IF
    WHENEVER ERROR CONTINUE
    INSERT INTO items (item_num, item_name, item_quantity)
    VALUES ( p_item[arr_curr()].* )
    WHENEVER ERROR STOP
    IF SQLCA.SQLCODE<0 THEN
        ERROR "Could not insert the record into database!"
    NEXT FIELD CURRENT
    ELSE
        MESSAGE "Record has been inserted successfully"
    END IF
    END IF
...
```

**Related concepts**

NEXT FIELD instruction on page 1497
BEFORE ROW block on page 1510
ON ROW CHANGE block on page 1544

**BEFORE INSERT block**

The **BEFORE INSERT** block is executed when a new row is created in an **INPUT ARRAY**. You typically use this trigger to set some default values in the newly-created row. A new row can be created by moving down after the last row, by executing a insert action, or by executing an append action.

The **BEFORE INSERT** block is executed after the **BEFORE ROW** block and before the **BEFORE FIELD** block.

When called in this block, `DIALOG.getCurrentRow() / arr_curr()` returns the index of the newly-created row.

To distinguish row insertion from an appended row, compare the current row (`DIALOG.getCurrentRow("screen-array")`) with the total number of rows (`DIALOG.getArrayLength("screen-array")`). If the current row index and the total number of rows correspond, the **BEFORE INSERT** concerns a temporary row, otherwise it concerns an inserted row.

Row creation can be stopped by using the `CANCEL INSERT` instruction inside **BEFORE INSERT**. If possible, it is however better to disable the insert and append actions to prevent the user executing the actions with `DIALOG.setActionActive()`.

In this example, the **BEFORE INSERT** block checks if the user can create rows and prevents new row creation if needed; otherwise, it sets some default values:

```sql
INPUT ARRAY p_items FROM s_items.*
...
BEFORE INSERT
IF NOT user_can_append THEN
   ERROR "You are not allowed to append rows"
   CANCEL INSERT
END IF
LET r = DIALOG.getCurrentRow("s_items")
LET p_items[r].item_num = get_new_serial("items")
LET p_items[r].item_name = "undefined"

Related concepts
Appending rows in INPUT ARRAY on page 1848
Rows appended at the end of an editable list are temporary until they are edited.
BEFORE ROW block on page 1510

AFTER INSERT block
The AFTER INSERT block of INPUT ARRAY is executed when the creation of a new row is validated. In this block, you can for example implement SQL to insert a new row in the database table.
The AFTER INSERT block is executed after the AFTER FIELD block and before the AFTER ROW block.
When called in this block, DIALOG.getCurrentRow() / arr_curr() returns the index of the newly-created row.
When the user appends a new row at the end of the list, then moves UP to another row or validates the dialog, the AFTER INSERT block is only executed if at least one field was edited. If no data entry is detected, the dialog automatically removes the new appended row and thus does not trigger the AFTER INSERT block.
When executing a NEXT FIELD in the AFTER INSERT block, the dialog will keep the focus in the list and stay in the current row. Use this behavior to implement row input validation and prevent the user from leaving the list or moving to another row. However, this will not cancel the row insertion and will not invoke the BEFORE INSERT / AFTER INSERT triggers again. The only way to keep the focus in the current row after the row was inserted is to execute a NEXT FIELD in the AFTER ROW block.
In this example, the AFTER INSERT block inserts a new row in the database and cancels the operation if the SQL command fails:

```
INPUT ARRAY p_items FROM s_items.*
...
   AFTER INSERT
      WHENEVER ERROR CONTINUE
      INSERT INTO items VALUES
         ( p_items[DIALOG.getCurrentRow("s_items")].* )
      WHENEVER ERROR STOP
      IF SQLCA.SQLCODE<>0 THEN
         ERROR SQLERRMESSAGE
         CANCEL INSERT
      END IF
```

Related concepts
NEXT FIELD instruction on page 1497
AFTER ROW block on page 1511

BEFORE DELETE block
The BEFORE DELETE block is executed each time the user deletes a row of an INPUT ARRAY list, before the row is removed from the list.
You typically code the database table synchronization in the BEFORE DELETE block, by executing a DELETE SQL statement using the primary key of the current row. In the BEFORE DELETE block, the row to be deleted still exists in the program array, so you can access its data to identify what record needs to be removed.
The BEFORE DELETE block is executed before the AFTER DELETE block.

If needed, the deletion can be canceled with the CANCEL DELETE instruction.

When called in this block, DIALOG.getCurrentRow() / arr_curr() returns the index of the row that will be deleted.

This example uses the BEFORE DELETE block to remove the row from the database table and cancels the deletion operation if an SQL error occurs:

```
INPUT ARRAY p_items FROM s_items.*
BEFORE DELETE
    LET r = DIALOG.getCurrentRow("s_items")
    WHENEVER ERROR CONTINUE
    DELETE FROM items
        WHERE item_num = p_items[r].item_num
        WHENEVER ERROR STOP
    IF SQLCA.SQLCODE<>0 VALUES
        ERROR SQLERRMESSAGE
        CANCEL DELETE
    END IF
...
```

Related concepts
AFTER DELETE block on page 1548

AFTER DELETE block

The AFTER DELETE block is executed each time the user deletes a row of an INPUT ARRAY list, after the row has been deleted from the list.

The AFTER DELETE block is executed after the BEFORE DELETE block and before the AFTER ROW block for the deleted row and the BEFORE ROW block of the new current row.

When an AFTER DELETE block executes, the program array has already been modified; the deleted row no longer exists in the array (except in the special case when deleting the last row). The arr_curr() function or the ui.Dialog.getCurrentRow() method returns the same index as in BEFORE ROW, but it is the index of the new current row. The AFTER ROW block is also executed just after the AFTER DELETE block.

Important: When deleting the last row of the list, AFTER DELETE is executed for the delete row, and DIALOG.getCurrentRow() / arr_curr() will be one greater than DIALOG.getArrayLength() / ARR_COUNT(). Ensure you avoid accessing a dynamic array with a row index that is greater than the total number of rows, otherwise the runtime system will adapt the total number of rows to the actual number of rows in the program array. When using a static array, you must ignore the values in the rows after ARR_COUNT().

Here the AFTER DELETE block is used to re-number the rows with a new item line number (note that DIALOG.getArrayLength() / ARR_COUNT() may return zero):

```
INPUT ARRAY p_items FROM s_items.*
AFTER DELETE
    LET r = DIALOG.getCurrentRow("s_items")
    FOR i=r TO DIALOG.getArrayLength("s_items")
        LET p_items[i].item_lineno = i
    END FOR
...
```

It is not possible to use the CANCEL_DELETE instruction in an AFTER DELETE block. At this time it is too late to cancel row deletion, as the data row no longer exists in the program array.

Related concepts
BEFORE INSERT block on page 1546
BEFORE FIELD block

In dialog instructions `INPUT`, `INPUT ARRAY`, `CONSTRUCT` or in a `DISPLAY ARRAY` using the `FOCUSONFIELD` attribute, the BEFORE FIELD block is executed every time the specified field gets the focus.

For single record inputs driven by `INPUT` or query by example (QBEs) driven by `CONSTRUCT`, the BEFORE FIELD block is executed when moving the focus from field to field.

For editable lists driven by `INPUT ARRAY`, the BEFORE FIELD block is executed when moving the focus from field to field in the same row, or when moving to another row in the same column.

For record lists driven by `DISPLAY ARRAY` using the `FOCUSONFIELD` attribute, the BEFORE FIELD block is executed when moving the focus from field to field. However, the fields will not be editable as in an `INPUT ARRAY`.

**Important:** The BEFORE FIELD block is also executed when performing a NEXT FIELD instruction.

The BEFORE FIELD keywords must be followed by a list of form field specification. The screen-record name can be omitted.

BEFORE FIELD is executed after BEFORE INPUT, BEFORE CONSTRUCT, BEFORE ROW and BEFORE INSERT.

Use this block to do some field value initialization, or to display a message to the user:

```plaintext
INPUT BY NAME p_cust.* ...  
BEFORE FIELD cust_status  
    LET p_cust.cust_comment = NULL  
MESSAGE "Enter customer status"
```

When using the default FIELD ORDER CONSTRAINT mode, the dialog executes the BEFORE FIELD block of the field corresponding to the first variable of an INPUT or INPUT ARRAY, even if that field is not editable (NOENTRY, hidden or disabled). The block is executed when you enter the dialog and every time you create a new row in the case of INPUT ARRAY. This behavior is supported for backward compatibility. The block is not executed when using the FIELD ORDER FORM, the mode recommended for DIALOG instructions.

With the FIELD ORDER FORM mode, for each dialog executing for the first time with a specific form, the BEFORE FIELD block will be invoked for the first field of the initial tabbing list defined by the form, even if that field was hidden or moved around in a table. The dialog then behaves as if a NEXT FIELD first-visible-column had been executed in the BEFORE FIELD of that field.

When form-level validation occurs and a field contains an invalid value, the dialog gives the focus to the field, but no BEFORE FIELD trigger will be executed.

**Related concepts**

Form-level validation rules on page 1727

Form-level validation rules can be defined for each field controlled by a dialog.

AFTER FIELD block on page 1492

**ON CHANGE block**

The ON CHANGE block can be used to detect when a field has been changed by user input. The ON CHANGE block is executed, if the value has changed since the field got the focus and the modification flag is set.

The ON CHANGE block can be used in `INPUT`, `INPUT ARRAY` and `CONSTRUCT` dialogs.

For editable fields defined as `EDIT`, `TEXTEDIT` or `BUTTONEDIT`, the ON CHANGE block is executed when leaving a field, if the value of the specified field has changed since the field got the focus and the modification flag is set for the field. The field is left when user validates the dialog, when moving to another field, or when moving to another row in an `INPUT ARRAY`. However, if the text edit field is defined with the `COMPLETER` attribute to enable autocompletion, the ON CHANGE trigger will be fired after a short period of time, when the user has typed characters in.
For editable fields defined as CHECKBOX, COMBOBOX, DATEEDIT, DATETIMEEDIT, TIMEEDIT, RADIOGROUP, SPINEDIT, SLIDER or URL-based WEBCOMPONENT (when the COMPONENTTYPE attribute is not used), the ON CHANGE block is invoked immediately when the user changes the value with the widget edition feature. For example, when toggling the state of a CHECKBOX, when selecting an item in a COMBOBOX list, or when choosing a date in the calendar of a DATEEDIT. Note that for such item types, when ON CHANGE is fired, the modification flag is always set.

```
ON CHANGE order_checked -- Defined as CHECKBOX
CALL setup_dialog(DIALOG)
```

**Note:** If both an ON CHANGE block and AFTER FIELD block are defined for a field, the ON CHANGE block is executed before the AFTER FIELD block.

When changing by program the value of the current field in an ON ACTION block, the ON CHANGE block will be executed when leaving the field, if the value is different from the previous value and the modification flag is set (after previous user input or when the touched flag has been changed by program).

In an INPUT or INPUT ARRAY, the field value change is related to the value of the variable bound to the field. In a CONSTRUCT dialog, the field value change is related to the input buffer / displayed value.

**Note:** With a NEXT FIELD instruction, the state of the field value change is reset, as if the user had left and reentered the field. When using NEXT FIELD in an ON CHANGE block or in an ON ACTION block, the ON CHANGE block will only be re-executed, if the value is changed since NEXT FIELD, and the modification flag is set.

Therefore, ON CHANGE should not be used for field validation with a NEXT FIELD, because ON CHANGE will not get triggered again, if the (invalid) value has not changed; Field validation rules must be implemented in AFTER FIELD blocks and/or AFTER INPUT blocks.

**Related concepts**
- **Form item types** on page 1266
  The form item types defines the purpose of form elements.
- **AFTER FIELD block** on page 1492

**AFTER FIELD block**

In dialog instructions INPUT, INPUT ARRAY, CONSTRUCT or in a DISPLAY ARRAY using the FOCUSONFIELD attribute, the AFTER FIELD block is executed every time the focus leaves the specified field.

For single record inputs driven by INPUT or query by example (QBEs) driven by CONSTRUCT, the AFTER FIELD block is executed when moving the focus from field to field.

For editable lists driven by INPUT ARRAY, the AFTER FIELD block is executed when moving the focus from field to field in the same row, or when moving to another row in the same column.

For record lists driven by DISPLAY ARRAY using the FOCUSONFIELD attribute, the AFTER FIELD block is executed when moving the focus from field to field. However, the fields will not be editable as in an INPUT ARRAY.

The AFTER FIELD keywords must be followed by a list of form field specifications. The screen-record name can be omitted.

AFTER FIELD is executed before AFTER INSERT, ON ROW CHANGE, AFTER ROW, AFTER INPUT or AFTER CONSTRUCT.

When a NEXT FIELD instruction is executed in an AFTER FIELD block, the cursor moves to the specified field, which can be the current field. This can be used to prevent the user from moving to another field / row during data input. Note that the BEFORE FIELD block is also executed when NEXT FIELD is invoked.

The AFTER FIELD block of the current field is not executed when performing a NEXT FIELD; only BEFORE INPUT, BEFORE CONSTRUCT, BEFORE ROW, and BEFORE FIELD of the target item might be executed, depending on the sub-dialog type.

When ACCEPT DIALOG, ACCEPT INPUT, or ACCEPT CONSTRUCT is performed, the AFTER FIELD trigger of the current field is executed.
Use the AFTER FIELD block to implement field validation rules:

```
INPUT BY NAME p_item.* ...
   AFTER FIELD item_quantity
       IF p_item.item_quantity <= 0 THEN
           ERROR "Item quantity cannot be negative or zero"
           LET p_item.item_quantity = 0
       END IF
   NEXT FIELD item_quantity
```

Related concepts

- ACCEPT DIALOG instruction on page 1640
- ON CHANGE block on page 1492

INPUT ARRAY interaction blocks

**ON ACTION block**

The ON ACTION `action-name` blocks execute a sequence of instructions when the user triggers a specific action.

A typical action handler block looks like this:

```
ON ACTION action-name
    instruction

```

Action blocks are bound by name to action views (like buttons) in the current form. Action views can be BUTTON, TOOLBAR buttons, or TOPMENU options, and if no explicit action view is defined, actions are rendered with a default action view, depending on the type of front-end.

This example defines an action block to open a typical zoom window and let the user select a customer record:

```
ON ACTION zoom
    CALL zoom_customers() RETURNING st, rec.cust_id, rec.cust_name
```

In a dialog handling user input such as INPUT, INPUT ARRAY and CONSTRUCT, if an action is specific to a field, add the INFIELD clause to have the action automatically enabled when the corresponding field gets the focus:

```
ON ACTION zoom INFIELD cust_city
    CALL zoom_cities() RETURN st, rec.cust_city
```

In most cases actions are decorated with action defaults in form files, but there can be cases where the ON ACTION handler needs to define its own attributes at the program level. This can be done by adding the ATTRIBUTES() clause of ON ACTION:

```
ON ACTION custinfo ATTRIBUTES(DISCLOSUREINDICATOR, IMAGE="info")
    CALL show_customer_info()
```

For more details about action handlers, and action configuration, see Dialog actions on page 1739.

Related concepts

- Configuring actions on page 1744
- Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with action attributes.
- Action handling basics on page 1739
- This topic describes the basics of action views, action events, and action handlers.

**ON IDLE block**

The ON IDLE `seconds` clause defines a set of instructions that must be executed after a given period of user inactivity. This interaction block can be used, for example, to quit the dialog after the user has not interacted with the program for a specified period of time.
Do not mix ON TIMER and ON IDLE clauses.

As ON IDLE can fire field input validation, it is therefore not recommended in dialogs allowing input.

The parameter of ON IDLE must be an integer literal or variable. If the value is zero, the dialog timeout is disabled.

It is not recommended to use the ON IDLE trigger with a short timeout period such as 1 or 2 seconds; The purpose of this trigger is to give the control back to the program after a relatively long period of inactivity (10, 30 or 60 seconds). This is typically the case when the end user leaves the workstation, or gets a phone call. The program can then execute some code before the user gets the control back.

```
ON IDLE 30
   IF ask_question("Do you want to reload information from the database?") THEN
      -- Fetch data back from the db server
   END IF
```

**Important:** The timeout value is taken into account when the dialog initializes its internal data structures. If you use a program variable instead of an integer constant, any change of the variable will have no effect if the variable is changed after the dialog has initialized. If you want to change the value of the timeout variable, it must be done before the dialog block.

**Related concepts**

- Get program control if user is inactive on page 1712
- Execute some code after a given number of seconds, when the user does not interact with the program.
- ON TIMER block on page 1466

**ON KEY block**

An ON KEY (key-name) block defines an action with a hidden action view (no default button is visible), that executes a sequence of instructions when the user presses the specified key.

**Note:** The ON KEY block is supported for backward compatibility with TUI mode applications. In new developments, consider using ON ACTION with accelerators defined in action defaults.

An ON KEY block can specify up to four accelerator keys. Each key creates a specific action object that will be identified by the key name in lowercase.

For example, ON KEY (F5,F6) creates two actions with the names F5 and F6. Each action object will get an acceleratorName attribute assigned, with the corresponding accelerator name. The specified keys must be one of the virtual keys.

**Note:** The KEY() clause allows a comma-separated list of keys. Up to four keys can be specified. For new developments, consider using a single key, or prefer ON ACTION handlers with a single accelerator definition in action defaults.

In GUI mode, action defaults are applied for ON KEY actions by using the name of the action (the key name). You can define secondary accelerator keys, as well as default decoration attributes like button text and image, by using the key name as action identifier. The action name is always in lowercase letters.

Check carefully ON KEY CONTROL-? statements to avoid having duplicate accelerators for multiple actions due to the accelerators defined by action defaults. Additionally, ON KEY statements used with ESC, TAB, UP, DOWN, LEFT, RIGHT, HELP, NEXT, PREVIOUS, INSERT, CONTROL-M, CONTROL-X, CONTROL-V, CONTROL-C and CONTROL-A should be avoided for use in GUI programs, because it's very likely to clash with default accelerators defined in the factory action defaults file provided by default.

By default, ON KEY actions are not decorated with a default button in the action frame (the default action view). You can show the default button by configuring a text attribute with the action defaults.

```
ON KEY (CONTROL-Z)
   CALL open_zoom()
```
Related concepts

- Configuring actions on page 1744
  Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with action attributes.

- Default action views on page 1743
  A default action view is created to render an action handler when no explicit action view exists for it.

ON SORT block

Basics

The ON SORT interaction block can be used to detect when rows have to be sorted in a DISPLAY ARRAY or INPUT ARRAY dialog.

ON SORT is used in two different contexts:

1. In a regular full-list DISPLAY ARRAY / INPUT ARRAY dialog, the ON SORT trigger can be used to detect that a list sort was performed.
2. In a DISPLAY ARRAY using paged mode (ON FILL BUFFER), use ON SORT to detect a sort request from the user and re-fetch the rows from the database in the required order.

ON SORT in regular full-list DISPLAY ARRAY or INPUT ARRAY

In a regular DISPLAY ARRAY / INPUT ARRAY dialog not using paged mode, the ON SORT trigger can be used to detect that a list sort was performed.

When the ON SORT block executes in this context, the (visual) sort is already done by the runtime system and the ON SORT block is only used to execute post-sort tasks, such as displaying current row information.

To display the row position information, use the arrayToVisualIndex() dialog method to convert the current program row number to the visual row number:

```dialog
DISPLAY ARRAY arr TO sr.* ...
...
ON SORT
    MESSAGE SFMT( "Row: %1/%2",
        DIALOG.arrayToVisualIndex( "sr", DIALOG.getCurrentRow("sr") ),
        DIALOG.getArrayLength( "sr" )
    )
...
```

If needed, you can get the sort column and sort order with the getSortKey() and isSortReverse() dialog methods:

```dialog
DISPLAY ARRAY arr TO sr.* ...
...
ON SORT
    MESSAGE SFMT( "Sort on %1, %2 order",
        DIALOG.getSortKey("sr"),
        IIF( DIALOG.isSortReverse("sr"), "descending", "ascending" )
    )
...
```

ON SORT in DISPLAY ARRAY using the paged mode

In a DISPLAY ARRAY implementing paged mode with ON FILL BUFFER trigger, built-in row sorting is not available because data is provided by pages.
Use the ON SORT trigger, to detect a sort request and perform a new SQL query to re-order the rows. In this context, the sort column and sort order are available with the getSortKey() and isSortReverse() dialog methods:

```
DEFINE key STRING, rev BOOLEAN
DISPLAY ARRAY arr TO sr.* ...

ON SORT
  -- Re-execute the SQL statement to fill the page of rows in ON FILL BUFFER
  -- Assuming that form field names match table column names
  LET key = DIALOG.getSortKey("sa")
  LET rev = DIALOG.isSortReverse("sa")
  IF key IS NULL THEN
    CALL execute_sql( NULL )
  ELSE
    CALL execute_sql( "ORDER BY " || key || IIF(rev," DESC"," ") )
  END IF
```

See Paged mode of DISPLAY ARRAY on page 1843 for more details about the paged mode in DISPLAY ARRAY and how to implement sort in this type of record list dialog.

**Related concepts**

- **List ordering** on page 1862
  List controllers implement a built-in sort. This feature can be disabled if not required.

**ON TIMER block**

The ON TIMER seconds clause defines a set of instructions that must be executed at regular intervals. This interaction block can be used, for example, to check if a message has arrived in a queue, and needs to be processed.

Do not mix ON TIMER and ON IDLE clauses.

As ON TIMER can fire field input validation, it is therefore not recommended in dialogs allowing input.

The parameter of ON TIMER must be an integer literal or variable. If the value is zero, the dialog timeout is disabled.

It is not recommended to use the ON TIMER trigger with a short timeout period, such as 1 or 2 seconds. The purpose of this trigger is to give the control back to the program after a reasonable period of time, such as 10, 20 or 60 seconds.

```
ON TIMER 30
  CALL check_for_messages()
```

**Important:** The timer value is taken into account when the dialog initializes its internal data structures. If you use a program variable instead of an integer constant, a change of the variable has no effect if the change takes place after the dialog has initialized. If you want to change the value of the timeout variable, it must be done before the dialog block.

**Related concepts**

- **Get program control on a regular (timed) basis** on page 1713
  Execute some code after a given number of seconds, with or without user interaction with the program.

**ON IDLE block** on page 1465

**INPUT ARRAY control instructions**

**ACCEPT INPUT instruction**

The ACCEPT INPUT instruction validates the INPUT instruction and exits the INPUT ARRAY instruction if no error is raised. The AFTER FIELD, ON CHANGE, etc., control blocks will be executed. Statements after the ACCEPT INPUT will not be executed.

Input field validation is a process that does several successive validation tasks, as listed here:
1. The current field value is checked, the check is based on the program variable data type (for example, the user must input a valid date in a DATE field).

2. **NOT NULL** field attributes are checked for all input fields. This attribute forces the field to have a value set by program or entered by the user. If the field contains no value, the constraint is not satisfied. Input values are right-trimmed, so if the user inputs only spaces, this corresponds to a NULL value which does not fulfill the NOT NULL constraint.

3. **INCLUDE** field attributes are checked for all input fields. This attribute forces the field to contain a value that is listed in the include list. If the field contains a value that is not in the list, the constraint is not satisfied.

4. **REQUIRED** field attributes are checked for all input fields. This attribute forces the field to have a default value, or to be "touched" by the user or by program. If the field was not edited during the dialog, the constraint is not satisfied.

If a field does not satisfy one of these constraints, dialog termination is canceled, an error message is displayed, and the focus goes to the first field causing a problem.

**EXIT INPUT instruction**

Use the EXIT INPUT to terminate the INPUT ARRAY instruction and resume the program execution at the instruction following the INPUT ARRAY block.

```c
ON ACTION leave_now
  EXIT INPUT
```

When leaving the INPUT ARRAY instruction, all form items used by the dialog will be disabled until another interactive statement takes control.

**Note:** The INT_FLAG will not be set to TRUE as when the cancel action is fired.

**Related concepts**

**ACCEPT INPUT instruction** on page 1554

**CANCEL DELETE instruction**

In a list controlled by an INPUT ARRAY, row deletion can be canceled by using the CANCEL DELETE instruction in the BEFORE DELETE block. Using this instruction in a different place will generate a compilation error.

When the CANCEL DELETE instruction is executed, the current BEFORE DELETE block is terminated without any other trigger execution (no BEFORE ROW or BEFORE FIELD is executed), and the program execution continues in the user event loop.

You can, for example, prevent row deletion based on some condition:

```c
BEFORE DELETE
  IF user_can_delete() == FALSE THEN
    ERROR "You are not allowed to delete rows"
    CANCEL DELETE
  END IF
```

The instructions that appear after CANCEL DELETE will be skipped.

If the row deletion condition is known before the delete action occurs, disable the delete action to prevent the user from performing a delete row action with the DIALOG.setActionActive() method:

```c
  CALL DIALOG.setActionActive("delete", FALSE)
```

It is also possible to prevent the user from deleting rows with the **DELETE ROW = FALSE** option in the ATTRIBUTE clause.

**Related concepts**

**BEFORE DELETE block** on page 1547
**CANCEL INSERT instruction**

In a list controlled by an **INPUT ARRAY**, row creation can be canceled by the program with the **CANCEL INSERT** instruction. This instruction can only be used in the **BEFORE INSERT** and **AFTER INSERT** control blocks. If it appears at a different place, the compiler will generate an error.

The instructions that appear after **CANCEL INSERT** will be skipped.

If the row creation condition is known before the insert/append action occurs, disable the insert and/or append actions to prevent the user from creating new rows, with **DIALOG.setActionActive()**:

```java
CALL DIALOG.setActionActive("insert", FALSE)
CALL DIALOG.setActionActive("append", FALSE)
```

However, this will not prevent the user from appending a new temporary row at the end of the list, when moving down after the last row. To prevent row creation completely, use the **INSERT ROW = FALSE** and **APPEND ROW = FALSE** options in the **ATTRIBUTE clause of INPUT ARRAY**, or combine with the **AUTO APPEND = FALSE** attribute.

**CANCEL INSERT in BEFORE INSERT**

A **CANCEL INSERT** executed inside a **BEFORE INSERT** block prevents the new row creation. The following tasks are performed:

1. No new row will be created (the new row is not yet shown to the user).
2. The **BEFORE INSERT** block is terminated (further instructions are skipped).
3. The **BEFORE ROW** and **BEFORE FIELD** triggers are executed.
4. Control goes back to the user.

You can, for example, cancel a row creation if the user is not allowed to create rows:

```java
BEFORE INSERT
    IF NOT user_can_insert THEN
        ERROR "You are not allowed to insert rows"
        CANCEL INSERT
    END IF
```

Executing **CANCEL INSERT** in **BEFORE INSERT** will also cancel a temporary row creation, except when there are no more rows in the list. In this case, **CANCEL INSERT** will just be ignored and leave the new row as is (otherwise, the instruction would loop without end). You can prevent automatic temporary row creation with the **AUTO APPEND = FALSE** attribute. If **AUTO APPEND = FALSE** and a **CANCEL INSERT** is executed in **BEFORE INSERT** (user has invoked an append action), the temporary row will be deleted and list will remain empty if it was the last row.

**CANCEL INSERT in AFTER INSERT**

A **CANCEL INSERT** executed inside an **AFTER INSERT** block removes the newly created row. The following tasks are performed:

1. The newly created row is removed from the list (the row exists now and user has entered data).
2. The **AFTER INSERT** block is terminated (further instructions are skipped).
3. The **BEFORE ROW** and **BEFORE FIELD** triggers are executed.
4. The control goes back to the user.

You can, for example, cancel a row insertion if a database error occurs when you try to insert the row into a database table:

```java
AFTER INSERT
    WHENEVER ERROR CONTINUE
    LET r = DIALOG.getCurrentRow("s_items")
    INSERT INTO items VALUES ( p_items[r].* )
```
WHENEVER ERROR STOP
IF SQLCA.SQLCODE<>0 THEN
  ERROR SQLERRMESSAGE
  CANCEL INSERT
END IF

Related concepts
BEFORE DELETE block on page 1547
Appending rows in INPUT ARRAY on page 1848
Rows appended at the end of an editable list are temporary until they are edited.

CONTINUE INPUT instruction

CONTINUE INPUT skips all subsequent statements in the current control block and gives the control back to the
dialog. This instruction is useful when program control is nested within multiple conditional statements, and you want
to return the control to the dialog.

If this instruction is called in a control block that is not AFTER INPUT, further control blocks might be executed
depending on the context.

CONTINUE INPUT instructs the dialog to continue as if the code in the control block was terminated (it's a kind of
GOTO end_of_control_block). However, when executed in AFTER INPUT, the focus returns to the current
row and current field in the list, giving the user another chance to enter data in that field. In this case the BEFORE
ROW and BEFORE FIELD triggers will be invoked.

In this example, an ON ACTION block gives control back to the dialog, skipping all subsequence instructions of the
ON ACTION block:

ON ACTION zoom
  IF p_cust.cust_id IS NULL OR p_cust.cust_name IS NULL THEN
    ERROR "Zoom window cannot be opened."
    CONTINUE INPUT
  END IF
  IF p_cust.cust_address IS NULL THEN
    ...
Use the NEXT FIELD control instruction to give the focus to a specific field and force the dialog to continue.
However, unlike CONTINUE INPUT, the NEXT FIELD instruction will also skip the further control blocks that are
normally executed.

Related concepts
AFTER FIELD block on page 1492
The Dialog class on page 2512
The ui.Dialog class provides a set of methods to configure, query and control the current interactive instruction.

NEXT FIELD instruction

Understanding the NEXT FIELD instruction

The NEXT FIELD field-name instruction gives the focus to the specified field. This instruction can be used
to control field input, in BEFORE FIELD, ON CHANGE or AFTER FIELD blocks, it can also force a DISPLAY
ARRAY or INPUT ARRAY to stay in the current row when NEXT FIELD is used in the AFTER ROW block.

If it exists, the BEFORE FIELD block of the corresponding field is executed.

In editable dialogs, the purpose of the NEXT FIELD instruction is to give the focus to an editable field. Make sure
that the field specified in NEXT FIELD is active and editable, or use NEXT FIELD CURRENT.

In a DISPLAY ARRAY using the FOCUSONFIELD attribute, NEXT FIELD can be used in conjunction with
DIALOG.setCurrentRow(), to set the focus to a specific cell in the list.
Instead of the NEXT FIELD instruction, you can use the DIALOG.nextField("field-name") method to register a field, for example when the name is not known at compile time. However, this method only registers the field. It does not stop code execution, like the NEXT FIELD instruction does. You must execute a CONTINUE DIALOG to get the same behavior as NEXT FIELD.

Form field identification with NEXT FIELD

With the NEXT FIELD instruction, fields are identified by the form field name specification, not the program variable name used by the dialog.

Form fields are bound to program variables with the binding clause of the dialog instruction (INPUT variable-list FROM field-list, INPUT BY NAME variable-list, CONSTRUCT BY NAME sql ON column-list, CONSTRUCT sql ON column-list FROM field-list, INPUT ARRAY array-name FROM screen-array.*).

The field name specification can be any of the following:

- field-name
- table-name.field-name
- screen-record-name.field-name
- FORMONLY.field-name

Here are some examples:

- "cust_name"
- "customer.cust_name"
- "cust_screen_record.cust_name"
- "item_screen_array.item_label"
- "formonly.total"

When no field name prefix is used, the first form field matching that simple field name is used.

When using a prefix in the field name specification, it must match the field prefix assigned by the dialog field binding method used at the beginning of the interactive statement: When no screen-record has been explicitly specified in the field binding clause (for example, when using INPUT BY NAME variable-list), the field prefix must be the database table name or FORMONLY, as defined in the form file, or any valid screen-record using that field. When the FROM clause of the dialog specifies an explicit screen-record (for example, in INPUT variable-list FROM screen-record.* / field-list-with-screen-record-prefix or INPUT ARRAY array-name FROM screen-array.*), the field prefix must be the screen-record name used in the FROM clause.

Abstract field identification is supported with the CURRENT, NEXT and PREVIOUS keywords. These keywords represent the current, next and previous fields respectively. When using FIELD ORDER FORM, the NEXT and PREVIOUS options follow the tabbing order defined by the form. Otherwise, they follow the order defined by the input binding list (with the FROM or BY NAME clause).

In a procedural dialog, if the focus is in the first field of an INPUT or CONSTRUCT sub-dialog, NEXT FIELD PREVIOUS will jump out of the current sub-dialog and set the focus to the previous sub-dialog. If the focus is in the last field of an INPUT or CONSTRUCT sub-dialog, NEXT FIELD NEXT will jump out of the current sub-dialog and set the focus to the next sub-dialog. NEXT FIELD NEXT or NEXT FIELD PREVIOUS also jumps to another sub-dialog when the focus is in a DISPLAY ARRAY sub-dialog. However, when using an INPUT ARRAY sub-dialog, NEXT FIELD NEXT from within the last column will loop to the first column of the current row, and NEXT FIELD PREVIOUS from within the first column will jump to the last column of the current row - the focus stays in the current INPUT ARRAY sub-dialog. When another sub-dialog gets the focus because of a NEXT FIELD NEXT/PREVIOUS, the newly-selected field depends on the sub-dialog type, following the tabbing order as if the end-user had pressed the tab or Shift-Tab key combination.

NEXT FIELD to a non-editable field

Non-editable fields are:

- Fields defined with the NOENTRY attribute in the form.
• Fields defined as PHANTOM fields in the form.
• Fields disabled at runtime with `DIALOG.setFieldActive()`.
• Fields using a widget that does not allow input, such as a LABEL.

In an INPUT, INPUT ARRAY or CONSTRUCT instruction, if a NEXT FIELD instruction specifies a non-editable field, the BEFORE FIELD block of that field is executed. Then the dialog tries to give the focus to that field. Since the field cannot get the focus, the dialog will perform the last pressed navigation key (Tab, Shift-Tab, Left, Right, Up, Down, Accept) and execute the related control blocks, including the AFTER FIELD block of the non-editable field. If no last key is identified, the dialog considers Tab as fallback and moves to the next editable field as defined by the FIELD ORDER mode used by the dialog.

**Note:** In an INPUT ARRAY, doing a NEXT FIELD to a non-editable field which has no editable fields in the last navigation direction, the dialog may move to the next or previous row and give the focus to the first or last editable field of that row.

When selecting a non-editable field with NEXT FIELD NEXT, the runtime system will re-select the current field since it is the next editable field in the dialog. As a result the end user sees no change.

**Important:** Doing a NEXT FIELD to a non-editable field can lead to infinite loops in the dialog; To stay in the current field / row, use NEXT FIELD CURRENT instead.

### NEXT FIELD in procedural DIALOG blocks

In a procedural dialog block, the NEXT FIELD `field-name` instruction gives the focus to the specified field controlled by INPUT, INPUT ARRAY or CONSTRUCT, or to a read-only list when using DISPLAY ARRAY.

When using a DISPLAY ARRAY sub-dialog, it is possible to give the focus to the list, by specifying the name of the first column as argument for NEXT FIELD.

If the target field specified in the NEXT FIELD instruction is inside the current sub-dialog, neither AFTER FIELD nor AFTER ROW will be invoked for the field or list you are leaving. However, the BEFORE FIELD control blocks of the destination field (or the BEFORE ROW in case of read-only list) will be executed.

If the target field specified in the NEXT FIELD instruction is outside the current sub-dialog, the AFTER FIELD, AFTER INSERT, AFTER ROW, and AFTER INPUT, AFTER DISPLAY, AFTER /CONSTRUCT control blocks will be invoked for the field or list you are leaving. Form-level validation rules will also be checked, as if the user had selected the new sub-dialog himself. This guarantees the current sub-dialog is left in a consistent state. The BEFORE INPUT, BEFORE DISPLAY, BEFORE CONSTRUCT, BEFORE ROW and the BEFORE FIELD control blocks of the destination field / list will then be executed.

### NEXT FIELD in record list control blocks

When using NEXT FIELD in AFTER ROW or in ON ROW CHANGE of a DISPLAY ARRAY or INPUT ARRAY, the dialog will stay in the current row and give control back to the user.

This behavior allows you to implement data input rules:

```plaintext
AFTER ROW
  IF NOT int_flag AND arr_count() <= arr_curr() THEN
    IF arr[arr_curr()].it_count * arr[arr_curr()].it_value > maxval THEN
      ERROR "Amount of line exceeds max value."
      NEXT FIELD item_count
    END IF
  END IF
END IF
```

If a condition is not met for a given field, do a NEXT FIELD to that field to point the user to the problem. If a global condition for the current row is not met, use NEXT FIELD CURRENT to stay in the current row and current field.

**Related concepts**

*Giving the focus to a form element* on page 1731
How to force the focus to move or stay in a specific form element using program code.

Understanding multiple dialogs on page 1585
Multiple dialogs are defined with DIALOG blocks inside a FUNCTION.

CLEAR instruction in dialogs
The CLEAR field-list and CLEAR SCREEN ARRAY screen-array.* instructions clear the value buffer of specified form fields. The buffers are directly changed in the current form, and the program variables bound to the dialog are left unchanged. CLEAR can be used outside any dialog instruction, such as the DISPLAY BY NAME / TO instructions.

When a dialog is configured with the UNBUFFERED mode, there is no reason to clear field buffers since any variable assignment will synchronize field buffers. Actually, changing the field buffers with DISPLAY or CLEAR instruction in an UNBUFFERED dialog will have no visual effect, because the variables bound to the dialog will be used to reset the field buffer just before giving control back to the user. To clear fields of an UNBUFFERED dialog, just set to NULL the variables bound to the dialog. However, when using a CONSTRUCT, no program variables are associated with the dialog and no UNBUFFERED concept exits, and the CLEAR or DISPLAY TO / BY NAME instructions are the only way to modify the CONSTRUCT fields.

A screen array with a screen-line specification doesn't make much sense in a GUI application using TABLE containers, you can therefore use the CLEAR SCREEN ARRAY instruction to clear all rows of a list.

Related concepts
Static display (DISPLAY/ERROR/MESSAGE/CLEAR) on page 1452
This section explains the instructions displaying static information to application forms, such as DISPLAY, ERROR, MESSAGE, CLEAR.

Examples
INPUT ARRAY dialog examples.
Example 1: INPUT ARRAY with empty record list
Form definition file "custlist.per":

```
SCHEMA shop
 LAYOUT
  TABLE
   { 
      Id   First name   Last name 
      [f001 |f002        |f003        ]
      [f001 |f002        |f003        ]
      [f001 |f002        |f003        ]
      [f001 |f002        |f003        ]
      [f001 |f002        |f003        ]
      [f001 |f002        |f003        ]
   } 
 END
 END
 TABLES
  customer
 END
 ATTRIBUTES
  f001 = customer.id ;
  f002 = customer.fname ;
  f003 = customer.lname, NOT NULL, REQUIRED ;
 END
 INSTRUCTIONS
  SCREEN RECORD sr_cust( customer.* );
 END
```
Program source code:

```plaintext
SCHEMA shop

MAIN
  DEFINE custarr DYNAMIC ARRAY OF RECORD LIKE customer.*

  OPEN FORM f FROM "custlist"
  DISPLAY FORM f

  INPUT ARRAY custarr WITHOUT DEFAULTS FROM sr_cust.*

END MAIN

Example 2: INPUT ARRAY using a static array

Form definition file "custlist.per":

```plaintext
SCHEMA shop
LAYOUT
  TABLE
    
    
    
    
    
    
    
    Id       First name   Last name
    [f001    |f002        |f003        ]
    [f001    |f002        |f003        ]
    [f001    |f002        |f003        ]
    [f001    |f002        |f003        ]
    [f001    |f002        |f003        ]
    [f001    |f002        |f003        ]
    [f001    |f002        |f003        ]

    END
END

TABLES
  customer
END

ATTRIBUTES
  f001 = customer.id ;
  f002 = customer.fname ;
  f003 = customer.lname, NOT NULL, REQUIRED ;
END

INSTRUCTIONS
  SCREEN RECORD sr_cust( customer.* );
END

Program code:

```plaintext
SCHEMA shop

MAIN

  DEFINE custarr ARRAY[100] OF RECORD LIKE customer.*
  DEFINE allow_insert, size INTEGER

  LET custarr[1].id = 1
  LET custarr[1].fname = "John"
  LET custarr[1].lname = "SMITH"
  LET custarr[2].id = 2
  LET custarr[2].fname = "Mike"
  LET custarr[2].lname = "STONE"
  LET size = 2
  LET allow_insert = TRUE

  OPEN FORM f1 FROM "custlist"
```
DISPLAY FORM f1

INPUT ARRAY custarr WITHOUT DEFAULTS FROM sr_cust.*
   ATTRIBUTES (COUNT=size, MAXCOUNT=50, UNBUFFERED, INSERT
   ROW=allow_insert)
   BEFORE INPUT
      MESSAGE "Editing the customer table"
   BEFORE INSERT
      IF arr_curr()=4 THEN
         CANCEL INSERT
      END IF
   BEFORE FIELD fname
      MESSAGE "Enter First Name"
   BEFORE FIELD lname
      MESSAGE "Enter Last Name"
   AFTER FIELD lname
      IF custarr[arr_curr()].lname IS NULL THEN
         LET custarr[arr_curr()].fname = NULL
      END IF
   END INPUT

END MAIN

Example 3: INPUT ARRAY using a dynamic array

SCHEMA shop
LAYOUT
TABLE
{
   Id       First name   Last name
   [f001    |f002        |f003        |
   [f001    |f002        |f003        |
   [f001    |f002        |f003        |
   [f001    |f002        |f003        |
   [f001    |f002        |f003        |
   [f001    |f002        |f003        |
  }
END
TABLES
   customer
END
ATTRIBUTES
   f001 = customer.id ;
   f002 = customer.fname ;
   f003 = customer.lname, NOT NULL, REQUIRED ;
END
INSTRUCTIONS
   SCREEN RECORD sr_cust( customer.* );
END

SCHEMA shop
MAIN

   DEFINE custarr DYNAMIC ARRAY OF RECORD LIKE customer.*
   DEFINE counter INTEGER
   FOR counter = 1 TO 500
      LET custarr[counter].id = counter
      LET custarr[counter].fname = "ff"||counter
Example 4: INPUT ARRAY updating the database table

Form definition file "custlist.per":

```
SCHEMA shop
LAYOUT
TABLE
{
    Id       First name   Last name
    [f001    |f002        |f003        ]
    [f001    |f002        |f003        ]
    [f001    |f002        |f003        ]
    [f001    |f002        |f003        ]
    [f001    |f002        |f003        ]
    [f001    |f002        |f003        ]
}
END
END TABLES
customer
END
ATTRIBUTES
    f001 = customer.id ;
    f002 = customer.fname ;
    f003 = customer.lname, NOT NULL, REQUIRED ;
END
INSTRUCTIONS
    SCREEN RECORD sr_cust( customer.* );
END
```

Program code:

```
SCHEMA shop

MAIN

DEFINE custarr DYNAMIC ARRAY OF RECORD LIKE customer.*

DEFINE op CHAR(1)
DEFINE i INTEGER

DATABASE shop

OPEN FORM f1 FROM "custlist"
DISPLAY FORM f1

DECLARE c1 CURSOR FOR
    SELECT id, fname, lname FROM customer ORDER BY id
LET i = 1
```
FOREACH c1 INTO custarr[i].*
    LET i = i + 1
END FOREACH
CALL custarr.deleteElement(custarr.getLength())

INPUT ARRAY custarr FROM sr_cust.*
    ATTRIBUTES(WITHOUT DEFAULTS, UNBUFFERED)

BEFORE DELETE
    IF op == "N" THEN -- No real SQL delete for new inserted rows
        IF NOT mbox_yn("List",
            "Are you sure you want to delete this record?",
            "question") THEN
            CANCEL DELETE -- Keeps row in list
        END IF
    WHENEVER ERROR CONTINUE
    DELETE FROM customer
        WHERE ID = custarr[arr_curr()].id
    WHENEVER ERROR STOP
    IF SQLCA.SQLCODE<0 THEN
        ERROR "Could not delete the record from database!"
        CANCEL DELETE -- Keeps row in list
    END IF
END IF

AFTER DELETE
    IF op == "N" THEN
        MESSAGE "Record has been deleted successfully"
    ELSE
        LET op = "N"
    END IF

AFTER FIELD fname
    IF custarr[arr_curr()].fname MATCHES "*@#$%^&()*" THEN
        ERROR "This field contains invalid characters"
    END IF

ON ROW CHANGE
    -- Warning: ON ROW CHANGE can occur if the SQL INSERT fails.
    IF op != "I" THEN LET op = "M" END IF

BEFORE INSERT
    LET op = "I"
    -- (not the best way to get a unique sequence number!)
    SELECT MAX(ID)+1 INTO custarr[arr_curr()].id FROM customer
    IF custarr[arr_curr()].id IS NULL THEN
        LET custarr[arr_curr()].id = 1
    END IF

AFTER INSERT
    LET op = "I"

BEFORE ROW
    LET op = "N"

AFTER ROW
    IF int_flag THEN EXIT INPUT END IF
    IF op == "M" OR op == "I" THEN
        IF custarr[arr_curr()].fname IS NULL
            OR custarr[arr_curr()].lname IS NULL
            OR custarr[arr_curr()].fname ==
            custarr[arr_curr()].lname THEN
            ERROR "First name and last name are equal..."
NEXT FIELD fname
END IF
END IF
IF op == "I" THEN
WHENEVER ERROR CONTINUE
INSERT INTO customer (id, fname, lname)
VALUES ( custarr[arr_curr()].* )
WHENEVER ERROR STOP
IF SQLCA.SQLCODE<0 THEN
ERROR "Could not insert the record into database!"
NEXT FIELD CURRENT
ELSE
MESSAGE "Record has been inserted successfully"
END IF
END IF
END IF
IF op == "M" THEN
WHENEVER ERROR CONTINUE
UPDATE customer SET
  fname = custarr[arr_curr()].fname,
  lname = custarr[arr_curr()].lname
WHERE id = custarr[arr_curr()].id
WHENEVER ERROR STOP
IF SQLCA.SQLCODE<0 THEN
ERROR "Could not update the record in database!"
NEXT FIELD CURRENT
ELSE
MESSAGE "Record has been updated successfully"
END IF
END IF
END INPUT
END MAIN

FUNCTION mbox_yn(title,message,icon)
  DEFINE title, message, icon STRING
  DEFINE r SMALLINT
  MENU title ATTRIBUTES(STYLE='dialog',IMAGE=icon,COMMENT=message)
    COMMAND "Yes" LET r=TRUE
    COMMAND "No"  LET r=FALSE
  END MENU
  RETURN r
END FUNCTION

Query by example (CONSTRUCT)
The CONSTRUCT instruction implements database query criteria input in an application form.

Understanding the CONSTRUCT instruction
The CONSTRUCT instruction provides database query, by entering search filters directly in form fields.

Query by example enables a user to query a database by specifying values (or ranges of values) for screen fields that correspond to the database columns.

The runtime system converts the query values entered by the user into a boolean SQL condition that can be used in the WHERE clause of a prepared SELECT statement.

The CONSTRUCT statement produces an SQL condition corresponding to all search criteria that a user specifies in the fields. The instruction fills a character variable with that SQL condition, and you can use the content of this variable to create the WHERE clause of a SELECT statement. The SELECT statement must be executed with the dynamic SQL management instructions PREPARE or DECLARE ident CURSOR FROM sqltext:
The CONSTRUCT instruction uses the data types of the form field to verify user input and to produce the SQL condition.

**Important:** The SQL condition is generated based on the current database session, which defines the type of database server. Therefore, the program must be connected to a database server before entering the CONSTRUCT block. The generated SQL condition is specific to the database server and may not be used with other types of database servers.

If no criteria were entered, the string '1=1' is assigned to the string variable. This is a boolean SQL expression that always evaluates to true so that all rows are returned.

The CONSTRUCT dialog activates the current form. This is the form most recently displayed or, if you are using more than one window, the form currently displayed in the current window. When the CONSTRUCT statement completes execution, the form is cleared and deactivated.

During a CONSTRUCT instruction, edit field input is left-aligned, independently to the form field data type: During an INPUT, numeric fields are right-aligned, during a CONSTRUCT, they are left-aligned.

By default the screen field tabbing order is defined by the order of the field names in the FROM clause; by default this is the list of column names in the ON clause when no FROM clause is specified. If needed, change the field tabbing order with the FIELD ORDER FORM option and TABINDEX field attributes. For more details, see Defining the tabbing order on page 1729.

When the user moves from field to field or changes values, dialog control blocks such as BEFORE FIELD are executed.

When the user clicks on an action view (button), or when an asynchronous event occurs, dialog interaction blocks like ON ACTION are executed.

The code inside a CONSTRUCT dialog can use control instructions, dialog control functions, and the ui.Dialog class, to implement the dialog behavior.

**Related concepts**

Result set processing on page 715
Shows how to fetch rows from a database query.

Dialog programming basics on page 1706
This section describes basic dialog programming concepts.

**Syntax of CONSTRUCT instruction**

The CONSTRUCT instruction provides database query by example, producing a WHERE condition for SELECT.

**Syntax**

```
CONSTRUCT { BY NAME variable ON column-list
        | variable ON column-list FROM field-list
        | ATTRIBUTES ( { display-attribute
        |   | control-attribute } )
        | HELP help-number
        | dialog-control-block
        | [...]
END CONSTRUCT }
```

where column-list defines a list of database columns as:

```
| column-name
| table-name.*
| table-name. column-name
| [...]
```
where *field-list* defines a list of fields with one or more of:

```
  | field-name
  | table-name.*
  | table-name.field-name
  | screen-array[line]. *
  | screen-array[line].field-name
  | screen-record.*
  | screen-record.field-name
  |
```

where *dialog-control-block* is one of:

```
  | BEFORE CONSTRUCT
  | AFTER CONSTRUCT
  | BEFORE FIELD field-spec [,....]
  | ON CHANGE field-spec [,....]
  | AFTER FIELD field-spec [,....]
  | ON IDLE seconds
  | ON TIMER seconds
  | ON ACTION action-name
    |  |  | INFIELD field-spec
    |  |  | attributes ( action-attributes-construct )
    |  |  | ON KEY ( key-name [,....] )
    |  |  | dialog-statement
    |  |  |
```

where *action-attributes-construct* is:

```
  | TEXT = string
  | COMMENT = string
  | IMAGE = string
  | ACCELERATOR = string
  | DEFAULTVIEW = YES | NO | AUTO
  | CONTEXTMENU = YES | NO | AUTO
  |
```

where *dialog-statement* is one of:

```
  | statement
  | NEXT FIELD | NEXT | PREVIOUS | field-spec |
  | CONTINUE CONSTRUCT
  | EXIT CONSTRUCT
  |
```

where *field-spec* identifies a unique field with one of:

```
  | field-name
  | table-name.field-name
  | screen-array.field-name
  | screen-record.field-name
  |
```

where *display-attribute* is:

```
  | BLACK | BLUE | CYAN | GREEN
  | MAGENTA | RED | WHITE | YELLOW
  | BOLD | DIM | INVISIBLE | NORMAL
  | REVERSE | BLINK | UNDERLINE
  |
```
where control-attribute is:

```plaintext
| ACCEPT [ = boolean ] |
| CANCEL [ = boolean ] |
| FIELD ORDER FORM     |
| HELP = help-number    |
| NAME = "dialog-name" |
```

1. **variable** is the variable that will contain the SQL condition built by the `CONSTRUCT` instruction.
2. **column-name** is the identifier of a database column of the current form.
3. **table-name** is the identifier of a database table of the current form.
4. **field-name** is the identifier of a field of the current form.
5. **screen-array** is the screen array that will be used in the current form.
6. **line** is a screen array line in the form.
7. **screen-record** is the identifier of a screen record of the current form.
8. **help-number** is an integer that allows you to associate a help message number with the instruction.
9. **key-name** is a hot-key identifier (like `F11` or `Control-z`).
10. **dialog-name** is the identifier of the dialog.
11. **action-name** identifies an action that can be executed by the user.
12. **seconds** is an integer literal or variable that defines a number of seconds.
13. **statement** is any instruction supported by the language.
14. **action-attributes** are dialog-specific action attributes.

**CONSTRUCT programming steps**

Follow this procedure to use the `CONSTRUCT` dialog instruction.

To implement a `CONSTRUCT` statement:

1. Declare a variable with the `DEFINE` statement, it can be CHAR, VARCHAR or STRING on page 305. STRING is preferred in order to avoid any size limitation.
2. Open and display the form, using an `OPEN WINDOW WITH FORM` or an `OPEN FORM/DISPLAY FORM` instruction.
3. Set the `INT_FLAG` on page 569 variable to `FALSE`.
4. Define the `CONSTRUCT` block with the list of form fields to be used for the query by example. If needed, define dialog control blocks to implement rules for the query by example.
5. Inside the `CONSTRUCT` statement, control the behavior of the instruction with `BEFORE CONSTRUCT`, `BEFORE FIELD`, `AFTER FIELD`, `AFTER CONSTRUCT` and `ON ACTION` blocks.
6. After the interaction statement block, test the `INT_FLAG` predefined variable to check if the dialog was canceled (INT_FLAG=TRUE) or validated (INT_FLAG=FALSE).
   - If the INT_FLAG variable is TRUE, you should reset it to FALSE to not disturb code that relies on this variable to detect interruption events from the GUI front-end or TUI console.
7. To build the complete SQL statement, concatenate "SELECT ... WHERE" to the string variable that contains the boolean SQL expression produced by `CONSTRUCT`.
8. Define a database cursor with the `DECLARE FROM` instruction, by using the `SELECT` statement.
9. Execute the cursor and fetch the rows found by the database server. You can for example implement a FOREACH loop to fill a program array, to be shown by a `DISPLAY ARRAY` statement.

**Related concepts**

Form specification files on page 1237
Form specification files are the source files defining the layout and content of application forms.

**Using query by example**
Dialog coding concepts, configuration, and code structure.

**Form field specification in CONSTRUCT**
In order to produce an SQL condition, the CONSTRUCT instruction uses a list of database columns that must match form fields for user input.

In **ATTRIBUTES** section of the form file, fields are typically defined from a database schema (with the form `tabname.colname`), to get the corresponding data type; however, it is also possible to use `FORMONLY.field-name` specifications with the **TYPE `data-type`** clause.

Unlike **INPUT**, **DISPLAY ARRAY** and **INPUT ARRAY**, the CONSTRUCT dialog does not use a program variable for each form field.

Only one string variable is required, to hold the SQL condition. Individual field criteria is available in the input buffers (`GET_FLD_BUF()`, `DIALOG.getFieldBuffer()`).

The list of database columns specified in the CONSTRUCT statement will appear in the SQL condition produced.

**Binding columns and fields by name**
The **CONSTRUCT BY NAME** variable `ON column-list` syntax maps the field names to database column names by name. Form fields are typically defined in the form by following a database schema, specifying the column name and data type.

```plaintext
SCHEMA stock
DEFINE where_part STRING
...
CONSTRUCT BY NAME where_part ON cust_name, cust_address
...
END CONSTRUCT
```

**Binding columns and fields by position**
The **CONSTRUCT** variable `ON column-list FROM field-list` clause explicitly maps database columns to form fields by position. The form can include other fields that are not part of the specified column list, but the number of variables or record members must equal the number of form fields listed in the **FROM** clause. Each database column must be of the same (or a compatible) data type as the corresponding form field. When the user enters data, the runtime system checks the entered value against the data type of the form field.

```plaintext
DEFINE where_part STRING
...
CONSTRUCT where_part ON cust_name, cust_address
    FROM field_02, field_04
...
END CONSTRUCT
```

**Related concepts**

**Binding variables to form fields** on page 1715
Some dialogs need program variables to store form field values.

**Query operators in CONSTRUCT**
The **CONSTRUCT** instruction supports a specific query syntax, using wildcard characters and comparison operators.
Table 373: CONSTRUCT query operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Search for this value. The string may contain * ?</td>
</tr>
<tr>
<td>=value</td>
<td>Equals value as is (wildcards are part of the value)</td>
</tr>
<tr>
<td>&lt;&gt; value or != value</td>
<td>Not equal to value (wildcards are part of the value)</td>
</tr>
<tr>
<td>= (without value)</td>
<td>Is NULL</td>
</tr>
<tr>
<td>&lt;&gt; or != (without value)</td>
<td>Is not NULL</td>
</tr>
<tr>
<td>&gt; value</td>
<td>Greater than value</td>
</tr>
<tr>
<td>&gt;= value</td>
<td>Greater than or equal to value</td>
</tr>
<tr>
<td>&lt; value</td>
<td>Less than value</td>
</tr>
<tr>
<td>&lt;= value</td>
<td>Less than or equal to value</td>
</tr>
<tr>
<td>value1:value2 or value1..value2</td>
<td>Range from value1 to value2</td>
</tr>
<tr>
<td>value1</td>
<td>value2 [</td>
</tr>
</tbody>
</table>

Note: When preceding the value with the = equal sign, wildcard characters such as * ? | : will be ignored and be part of the value.

Table 374: CONSTRUCT character wildcards

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>A sequence of zero to n of any characters</td>
</tr>
<tr>
<td>?</td>
<td>Any single-character at this position</td>
</tr>
<tr>
<td>[c1-c2]</td>
<td>A character in the specified range, at this position</td>
</tr>
<tr>
<td>[^c1-c2]</td>
<td>A character NOT in the specified range, at this position</td>
</tr>
<tr>
<td>[c1c2 [ ... ] ]</td>
<td>A character in the specified set, at this position</td>
</tr>
<tr>
<td>[^c1c2 [ ... ] ]</td>
<td>A character NOT in the specified set, at this position</td>
</tr>
</tbody>
</table>

Queries based on character types are case sensitive, because SQL is case sensitive, except if the database server is configured to be case-insensitive.

The * (star) and ? (question mark) wildcards are specific to character string type queries, and will generate a MATCHES expression or a LIKE expression, depending on the type of database used. When entering a * or ?, the pattern can also contain a character range specification with the square brackets notation [a-z] or [xyz]. A caret (^.) as the first character within the square brackets specifies the logical complement of the set, and matches any character that is not listed. For example, the search value [^AB]* specifies all strings beginning with characters other than A or B.

Some syntaxes can produce an "Error in field" dialog error if the feature is supported by the pattern matching operator of the database server. For example, not all db servers support the [a-z] character range specification in the LIKE pattern.

To search for rows with values containing a * star, a ? question mark or a \ backslash, escape the wildcard character with a backslash. Another option is to put an equal sign at the beginning of the QBE string, to find values that match exactly that string.
Table 375: CONSTRUCT input examples with matching and non matching values

<table>
<thead>
<tr>
<th>QBE input example</th>
<th>Matching values</th>
<th>Non matching values</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>100</td>
<td>99, 101, NULL</td>
</tr>
<tr>
<td>&gt;=100</td>
<td>100, 101, 200</td>
<td>10, 99, NULL</td>
</tr>
<tr>
<td>!=100</td>
<td>98, 98, 101, 102</td>
<td>100, NULL</td>
</tr>
<tr>
<td>!=</td>
<td>98, 99, 100, 101</td>
<td>NULL</td>
</tr>
<tr>
<td>1:100</td>
<td>1, 2 ... 99, 100</td>
<td>0, 101, NULL</td>
</tr>
<tr>
<td>aaa:yyy</td>
<td>aaa, aab, ab, yy, yyy</td>
<td>zaa, NULL</td>
</tr>
<tr>
<td>abc</td>
<td>abc</td>
<td>bc, abcd, Abc, NULL</td>
</tr>
<tr>
<td>ABC</td>
<td>ABC</td>
<td>abc, ABC, NULL</td>
</tr>
<tr>
<td>abc*</td>
<td>abc, abcd, abcdef</td>
<td>bc, ABC, NULL</td>
</tr>
<tr>
<td>=abc*</td>
<td>abc*</td>
<td>abc, abcdedef</td>
</tr>
<tr>
<td>*bc</td>
<td>abc, bc</td>
<td>acd, ABC, NULL</td>
</tr>
<tr>
<td>?bc</td>
<td>abc, xbc, zbc</td>
<td>aabc, aBC, NULL</td>
</tr>
<tr>
<td>*bc?</td>
<td>aaaaabcd, abcd, bcd</td>
<td>abcdef, bcdef, NULL</td>
</tr>
<tr>
<td>[a-z]bc</td>
<td>abc, ebc, zbc</td>
<td>2bc, +bc, Abc, NULL</td>
</tr>
<tr>
<td>[^abc]*</td>
<td>deee, feee, zyx, z</td>
<td>azzz, byy, d, NULL</td>
</tr>
<tr>
<td>a[bxy]c</td>
<td>abc, axc, ayc</td>
<td>a2c, azc, aBc, NULL</td>
</tr>
<tr>
<td>*[xyz]c</td>
<td>abcxc, eeeez</td>
<td>abcd, eeee, NULL</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>35</td>
</tr>
<tr>
<td>aa</td>
<td>bb</td>
<td>cc</td>
</tr>
<tr>
<td>=aa</td>
<td>bb</td>
<td>cc</td>
</tr>
<tr>
<td>\abc</td>
<td>\abc, \abcdef</td>
<td>abc, NULL</td>
</tr>
<tr>
<td>*bc</td>
<td>*bc</td>
<td>abc, bc, NULL</td>
</tr>
<tr>
<td>*[?]*</td>
<td>[a], a[b]c, xx[y]zz</td>
<td>a[bb]c, a[]c, NULL</td>
</tr>
</tbody>
</table>

CONSTRUCT instruction configuration

This section describes the options that can be specified in the ATTRIBUTES clause of the CONSTRUCT instruction. The options of the ATTRIBUTES clause override all default attributes and temporarily override any display attributes that the OPTIONS or the OPEN WINDOW statement specified for these fields. With the CONSTRUCT statement, the INVISIBLE attribute is ignored.

NAME option

The NAME attribute can be used to name the CONSTRUCT dialog. This attribute is used to identify the CONSTRUCT dialog.

HELP option

The HELP clause specifies the number of a help message to display if the user invokes the help in the CONSTRUCT dialog. The predefined ‘help’ action is automatically created by the runtime system. You can bind action views to the ‘help’ action.

The HELP clause overrides the HELP attribute.
FIELD ORDER FORM option

By default, the tabbing order is defined by the variable binding list in the instruction description. You can control the tabbing order by using the FIELD ORDER FORM attribute. When this attribute is used, the tabbing order is defined by the TABINDEX attribute of the form fields. If this attribute is used, the Dialog.fieldOrder FGLPROFILE entry is ignored.

The OPTIONS instruction can also change the behavior of the INPUT instruction, with the INPUT WRAP or FIELD ORDER FORM options.

ACCEPT option

The ACCEPT attribute can be set to FALSE to avoid the automatic creation of the accept default action. This option can be used for example when you want to write a specific validation procedure, by using ACCEPT INPUT.

CANCEL option

The CANCEL attribute can be set to FALSE to avoid the automatic creation of the cancel default action. This is useful for example when you only need a validation action (accept), or when you want to write a specific cancellation procedure, by using EXIT INPUT.

If the CANCEL=FALSE option is set, no close action will be created, and you must write an ON ACTION close control block to create an explicit action.

Related concepts

Syntax of CONSTRUCT instruction on page 1566
The CONSTRUCT instruction provides database query by example, producing a WHERE condition for SELECT.

Default actions IN CONSTRUCT

When an CONSTRUCT instruction executes, the runtime system creates a set of default actions. Depending on the invoked default action, field validation occurs and different CONSTRUCT control blocks are executed.

This table lists the default actions created for this dialog:

Table 376: Default actions created for the CONSTRUCT dialog

<table>
<thead>
<tr>
<th>Default action</th>
<th>Description</th>
</tr>
</thead>
</table>
| accept         | Validates the CONSTRUCT dialog (validates field criteria)  
                | *Creation can be avoided with ACCEPT attribute.* |
| cancel         | Cancels the CONSTRUCT dialog (no validation, INT_FLAG is set)  
                | *Creation can be avoided with CANCEL attribute.* |
| close          | By default, cancels the CONSTRUCT dialog (no validation, INT_FLAG is set)  
                | Default action view is hidden. See Implementing the close action on page 1772. |
| help           | Shows the help topic defined by the HELP clause.  
                | *Only created when a HELP clause is defined.* |

The accept and cancel default actions can be avoided with the ACCEPT and CANCEL dialog control attributes:

CONSTRUCT BY NAME cond ON field1 ATTRIBUTES (CANCEL=FALSE)
Related concepts
Dialog programming basics on page 1706
This section describes basic dialog programming concepts.

CONSTRUCT control blocks
CONSTRUCT control blocks execution order
This table shows the order in which the runtime system executes the control blocks in the CONSTRUCT instruction, depending on the user action:

Table 377: Control block execution order for CONSTRUCT

<table>
<thead>
<tr>
<th>Context / User action</th>
<th>Control Block execution order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entering the dialog</td>
<td>1. BEFORE CONSTRUCT</td>
</tr>
<tr>
<td></td>
<td>2. BEFORE FIELD (first field)</td>
</tr>
<tr>
<td>Moving from field A to field B</td>
<td>1. AFTER FIELD (for field A)</td>
</tr>
<tr>
<td></td>
<td>2. BEFORE FIELD (for field B)</td>
</tr>
<tr>
<td>Validating the dialog</td>
<td>1. AFTER FIELD</td>
</tr>
<tr>
<td></td>
<td>2. AFTER CONSTRUCT</td>
</tr>
<tr>
<td>Canceling the dialog</td>
<td>1. AFTER CONSTRUCT</td>
</tr>
</tbody>
</table>

BEFORE CONSTRUCT block

BEFORE CONSTRUCT block in singular and parallel CONSTRUCT dialogs
In a singular CONSTRUCT instruction, or when used as parallel dialog, the BEFORE CONSTRUCT is only executed once when dialog is started.

The BEFORE CONSTRUCT block is executed once at dialog start-up, before the runtime system gives control to the user for criteria input. This block can be used to display messages to the user, initialize form fields with default search criteria values, and setup the dialog instance by deactivating unused fields or actions the user is not allowed to execute.

```
CONSTRUCT BY NAME where_part ON ...
BEFORE CONSTRUCT
    MESSAGE "Enter customer search filter"
    CALL DIALOG.setActionActive("clean", FALSE )
...
```

The fields are cleared before the BEFORE CONSTRUCT block is executed.

You can use the NEXT FIELD control instruction in the BEFORE CONSTRUCT block, to jump to a specific field when the dialog starts.

BEFORE CONSTRUCT block in CONSTRUCT of procedural DIALOG

In a CONSTRUCT sub-dialog of a procedural DIALOG instruction, the BEFORE CONSTRUCT block is executed when the focus goes to a group of fields driven by a CONSTRUCT sub-dialog. This trigger is only invoked if a field of the sub-dialog gets the focus, and none of the other fields had the focus.

BEFORE CONSTRUCT is executed after the BEFORE DIALOG block and before the BEFORE FIELD blocks.

In this example, the BEFORE CONSTRUCT block is used to display a message:

```
CONSTRUCT BY NAME sql ON customer.*
```
BEFORE CONSTRUCT
MESSAGE "Enter customer search filter"

Related concepts
BEFORE INPUT block on page 1489
BEFORE DISPLAY block on page 1509
AFTER CONSTRUCT block on page 1574

AFTER CONSTRUCT block

AFTER CONSTRUCT block in singular and parallel CONSTRUCT dialogs

In a singular CONSTRUCT instruction, or when used as parallel dialog, the AFTER CONSTRUCT is only executed once when dialog is ended.

Use an AFTER CONSTRUCT block to execute instructions after the user has finished search criteria input.
AFTER CONSTRUCT is not executed if an EXIT CONSTRUCT is performed.

The code in AFTER CONSTRUCT can for example check if a criteria combination of different fields is required or denied, and force the end use to enter all

Before checking the content of the fields used in the CONSTRUCT, make sure that the INT_FLAG variable is FALSE. In the case that the user cancels the dialog, the validation rules must be skipped.

Since no program variables are associated with the form fields, you must query the input buffers of the fields to get the values entered by the user.

CONSTRUCT BY NAME where_part ON ...

... 
AFTER CONSTRUCT
    IF NOT INT_FLAG THEN
        IF length(DIALOG.getFieldBuffer(cust_name)) == 0
            OR length(DIALOG.getFieldBuffer(cust_addr)) == 0 THEN
            ERROR "Enter a search criteria for customer name and address fields."
        NEXT FIELD CURRENT
    END IF
END IF
END CONSTRUCT

To limit the validation to fields that have been modified by the end user, you can call the FIELD_TOUCHED() function or the DIALOG.getFieldTouched() method to check if a field has changed during the dialog execution. This makes your validation code execute faster if the user has only modified a couple of fields in a large form.

AFTER CONSTRUCT block in CONSTRUCT of procedural DIALOG

In a CONSTRUCT sub-dialog of a procedural DIALOG instruction, the AFTER CONSTRUCT block is executed when the focus is lost by a group of fields driven by a CONSTRUCT sub-dialog. This trigger is invoked if a field of the sub-dialog loses the focus, and a field of a different sub-dialog gets the focus.

If the focus leaves the current group and goes to an action view, this trigger is not executed, because the focus did not go to another sub-dialog yet.

AFTER CONSTRUCT is executed after the AFTER FIELD and before the AFTER DIALOG block.

Executing a NEXT FIELD in the AFTER CONSTRUCT control block will keep the focus in the group of fields.

In this example, the AFTER CONSTRUCT block is used to build the SELECT statement:

CONSTRUCT BY NAME sql ON customer.*
AFTER CONSTRUCT
LET sql = "SELECT * FROM customers WHERE " || sql

Related concepts
AFTER DISPLAY block on page 1510
AFTER INPUT block on page 1490
BEFORE CONSTRUCT block on page 1573

BEFORE FIELD block
In dialog instructions INPUT, INPUT ARRAY, CONSTRUCT or in a DISPLAY ARRAY using the FOCUSONFIELD attribute, the BEFORE FIELD block is executed every time the specified field gets the focus.

For single record inputs driven by INPUT or query by example (QBEs) driven by CONSTRUCT, the BEFORE FIELD block is executed when moving the focus from field to field.

For editable lists driven by INPUT ARRAY, the BEFORE FIELD block is executed when moving the focus from field to field in the same row, or when moving to another row in the same column.

For record lists driven by DISPLAY ARRAY using the FOCUSONFIELD attribute, the BEFORE FIELD block is executed when moving the focus from field to field. However, the fields will not be editable as in an INPUT ARRAY.

Important: The BEFORE FIELD block is also executed when performing a NEXT FIELD instruction.

The BEFORE FIELD keywords must be followed by a list of form field specification. The screen-record name can be omitted.

BEFORE FIELD is executed after BEFORE INPUT, BEFORE CONSTRUCT, BEFORE ROW and BEFORE INSERT.

Use this block to do some field value initialization, or to display a message to the user:

```
INPUT BY NAME p_cust.* ...
BEFORE FIELD cust_status
   LET p_cust.cust_comment = NULL
   MESSAGE "Enter customer status"
```

When using the default FIELD ORDER CONSTRAINT mode, the dialog executes the BEFORE FIELD block of the field corresponding to the first variable of an INPUT or INPUT ARRAY, even if that field is not editable (NOENTRY, hidden or disabled). The block is executed when you enter the dialog and every time you create a new row in the case of INPUT ARRAY. This behavior is supported for backward compatibility. The block is not executed when using the FIELD ORDER FORM, the mode recommended for DIALOG instructions.

With the FIELD ORDER FORM mode, for each dialog executing for the first time with a specific form, the BEFORE FIELD block will be invoked for the first field of the initial tabbing list defined by the form, even if that field was hidden or moved around in a table. The dialog then behaves as if a NEXT FIELD first-visible-column had been executed in the BEFORE FIELD of that field.

When form-level validation occurs and a field contains an invalid value, the dialog gives the focus to the field, but no BEFORE FIELD trigger will be executed.

Related concepts
Form-level validation rules on page 1727
Form-level validation rules can be defined for each field controlled by a dialog.

AFTER FIELD block on page 1492

ON CHANGE block
The ON CHANGE block can be used to detect when a field has been changed by user input. The ON CHANGE block is executed, if the value has changed since the field got the focus and the modification flag is set.

The ON CHANGE block can be used in INPUT, INPUT ARRAY and CONSTRUCT dialogs.
For editable fields defined as EDIT, TEXTEDIT or BUTTONEDIT, the ON CHANGE block is executed when leaving a field, if the value of the specified field has changed since the field got the focus and the modification flag is set for the field. The field is left when user validates the dialog, when moving to another field, or when moving to another row in an INPUT ARRAY. However, if the text edit field is defined with the COMPLETER attribute to enable autocompletion, the ON CHANGE trigger will be fired after a short period of time, when the user has typed characters in.

For editable fields defined as CHECKBOX, COMBOBOX, DATEEDIT, DATETIMEEDIT, TIMEEDIT, RADIOGROUP, SPINETEDIT, SLIDER or URL-based WEBCOMPONENT (when the COMPONENTTYPE attribute is not used), the ON CHANGE block is invoked immediately when the user changes the value with the widget edition feature. For example, when toggling the state of a CHECKBOX, when selecting an item in a COMBOBOX list, or when choosing a date in the calendar of a DATEEDIT. Note that for such item types, when ON CHANGE is fired, the modification flag is always set.

Note: If both an ON CHANGE block and AFTER FIELD block are defined for a field, the ON CHANGE block is executed before the AFTER FIELD block.

When changing by program the value of the current field in an ON ACTION block, the ON CHANGE block will be executed when leaving the field, if the value is different from the previous value and the modification flag is set (after previous user input or when the touched flag has been changed by program).

In an INPUT or INPUT ARRAY, the field value change is related to the value of the variable bound to the field. In a CONSTRUCT dialog, the field value change is related to the input buffer / displayed value.

Note: With a NEXT FIELD instruction, the state of the field value change is reset, as if the user had left and reentered the field. When using NEXT FIELD in an ON CHANGE block or in an ON ACTION block, the ON CHANGE block will only be re-executed, if the value is changed since NEXT FIELD, and the modification flag is set. Therefore, ON CHANGE should not be used for field validation with a NEXT FIELD, because ON CHANGE will not get triggered again, if the (invalid) value has not changed; Field validation rules must be implemented in AFTER FIELD blocks and/or AFTER INPUT blocks.

Related concepts
Form item types on page 1266
The form item types defines the purpose of form elements.
AFTER FIELD block on page 1492

AFTER FIELD block
In dialog instructions INPUT, INPUT ARRAY, CONSTRUCT or in a DISPLAY ARRAY using the FOCUSONFIELD attribute, the AFTER FIELD block is executed every time the focus leaves the specified field.

For single record inputs driven by INPUT or query by example (QBEs) driven by CONSTRUCT, the AFTER FIELD block is executed when moving the focus from field to field.

For editable lists driven by INPUT ARRAY, the AFTER FIELD block is executed when moving the focus from field to field in the same row, or when moving to another row in the same column.

For record lists driven by DISPLAY ARRAY using the FOCUSONFIELD attribute, the AFTER FIELD block is executed when moving the focus from field to field. However, the fields will not be editable as in an INPUT ARRAY.

The AFTER FIELD keywords must be followed by a list of form field specifications. The screen-record name can be omitted.

AFTER FIELD is executed before AFTER INSERT, ON ROW CHANGE, AFTER ROW, AFTER INPUT or AFTER CONSTRUCT.

When a NEXT FIELD instruction is executed in an AFTER FIELD block, the cursor moves to the specified field, which can be the current field. This can be used to prevent the user from moving to another field / row during data input. Note that the BEFORE FIELD block is also executed when NEXT FIELD is invoked.
The **AFTER FIELD** block of the current field is not executed when performing a **NEXT FIELD**; only **BEFORE INPUT**, **BEFORE CONSTRUCT**, **BEFORE ROW**, and **BEFORE FIELD** of the target item might be executed, depending on the sub-dialog type.

When **ACCEPT DIALOG**, **ACCEPT INPUT**, or **ACCEPT CONSTRUCT** is performed, the **AFTER FIELD** trigger of the current field is executed.

Use the **AFTER FIELD** block to implement field validation rules:

```plaintext
INPUT BY NAME p_item.* ...
AFTER FIELD item_quantity
  IF p_item.item_quantity <= 0 THEN
    ERROR "Item quantity cannot be negative or zero"
    LET p_item.item_quantity = 0
  NEXT FIELD item_quantity
END IF
```

**Related concepts**

- **ACCEPT DIALOG instruction** on page 1640
- **ON CHANGE block** on page 1492

**CONSTRUCT interaction blocks**

**ON ACTION block**

The **ON ACTION** `action-name` blocks execute a sequence of instructions when the user triggers a specific action.

A typical action handler block looks like this:

```plaintext
ON ACTION action-name
  instruction
...
```

Action blocks are bound by name to action views (like buttons) in the current form. Action views can be **BUTTON**, **TOOLBAR** buttons, or **TOPMENU** options, and if no explicit action view is defined, actions are rendered with a default action view, depending on the type of front-end.

This example defines an action block to open a typical zoom window and let the user select a customer record:

```plaintext
ON ACTION zoom
  CALL zoom_customers() RETURNING st, rec.cust_id, rec.cust_name
```

In a dialog handling user input such as **INPUT**, **INPUT ARRAY**, and **CONSTRUCT**, if an action is specific to a field, add the **INFIELD** clause to have the action automatically enabled when the corresponding field gets the focus:

```plaintext
ON ACTION zoom INFIELD cust_city
  CALL zoom_cities() RETURN st, rec.cust_city
```

In most cases actions are decorated with action defaults in form files, but there can be cases where the **ON ACTION** handler needs to define its own attributes at the program level. This can be done by adding the **ATTRIBUTES()** clause of **ON ACTION**:

```plaintext
ON ACTION custinfo ATTRIBUTES(DISCLOSUREINDICATOR, IMAGE="info")
  CALL show_customer_info()
```

For more details about action handlers, and action configuration, see **Dialog actions** on page 1739.

**Related concepts**

- **Configuring actions** on page 1744
- Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with **action attributes**.
- **Action handling basics** on page 1739
This topic describes the basics of action views, action events, and action handlers.

**ON IDLE block**

The **ON IDLE** *seconds* clause defines a set of instructions that must be executed after a given period of user inactivity. This interaction block can be used, for example, to quit the dialog after the user has not interacted with the program for a specified period of time.

Do not mix **ON TIMER** and **ON IDLE** clauses.

As **ON IDLE** can fire field input validation, it is therefore not recommended in dialogs allowing input.

The parameter of **ON IDLE** must be an integer literal or variable. If the value is zero, the dialog timeout is disabled.

It is not recommended to use the **ON IDLE** trigger with a short timeout period such as 1 or 2 seconds; The purpose of this trigger is to give the control back to the program after a relatively long period of inactivity (10, 30 or 60 seconds). This is typically the case when the end user leaves the workstation, or gets a phone call. The program can then execute some code before the user gets the control back.

```plaintext
ON IDLE 30
  IF ask_question("Do you want to reload information from the database?") THEN
    -- Fetch data back from the db server
  END IF
```

**Important:** The timeout value is taken into account when the dialog initializes its internal data structures. If you use a program variable instead of an integer constant, any change of the variable will have no effect if the variable is changed after the dialog has initialized. If you want to change the value of the timeout variable, it must be done before the dialog block.

**Related concepts**

Get program control if user is inactive on page 1712

Execute some code after a given number of seconds, when the user does not interact with the program.

**ON TIMER block** on page 1466

**ON KEY block**

An **ON KEY** *(key-name)* block defines an action with a hidden action view (no default button is visible), that executes a sequence of instructions when the user presses the specified key.

**Note:** The **ON KEY** block is supported for backward compatibility with TUI mode applications. In new developments, consider using **ON ACTION** with accelerators defined in action defaults.

An **ON KEY** block can specify up to four accelerator keys. Each key creates a specific action object that will be identified by the key name in lowercase.

For example, **ON KEY(F5,F6)** creates two actions with the names *F5* and *F6*. Each action object will get an **acceleratorName** attribute assigned, with the corresponding accelerator name. The specified keys must be one of the virtual keys.

**Note:** The **KEY()** clause allows a comma-separated list of keys. Up to four keys can be specified. For new developments, consider using a single key, or prefer **ON ACTION** handlers with a single accelerator definition in action defaults.

In GUI mode, action defaults are applied for **ON KEY** actions by using the name of the action (the key name). You can define secondary accelerator keys, as well as default decoration attributes like button text and image, by using the key name as action identifier. The action name is always in lowercase letters.

Check carefully **ON KEY CONTROL-?** statements to avoid having duplicate accelerators for multiple actions due to the accelerators defined by action defaults. Additionally, **ON KEY** statements used with ESC, TAB, UP, DOWN, LEFT, RIGHT, HELP, NEXT, PREVIOUS, INSERT, CONTROL-M, CONTROL-X, CONTROL-V, CONTROL-C and CONTROL-A should be avoided for use in GUI programs, because it's very likely to clash with default accelerators defined in the factory action defaults file provided by default.
By default, ON KEY actions are not decorated with a default button in the action frame (the default action view). You can show the default button by configuring a text attribute with the action defaults.

```plaintext
ON KEY (CONTROL-Z)
    CALL open_zoom()
```

**Related concepts**

- **Configuring actions** on page 1744
  Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with *action attributes*.

- **Default action views** on page 1743
  A default action view is created to render an action handler when no explicit action view exists for it.

**ON TIMER block**

The ON TIMER *seconds* clause defines a set of instructions that must be executed at regular intervals. This interaction block can be used, for example, to check if a message has arrived in a queue, and needs to be processed.

Do not mix ON TIMER and ON IDLE clauses.

As ON TIMER can fire field input validation, it is therefore not recommended in dialogs allowing input.

The parameter of ON TIMER must be an integer literal or variable. If the value is zero, the dialog timeout is disabled.

It is not recommended to use the ON TIMER trigger with a short timeout period, such as 1 or 2 seconds. The purpose of this trigger is to give the control back to the program after a reasonable period of time, such as 10, 20 or 60 seconds.

```plaintext
ON TIMER 30
    CALL check_for_messages()
```

**Important:** The timer value is taken into account when the dialog initializes its internal data structures. If you use a program variable instead of an integer constant, a change of the variable has no effect if the change takes place after the dialog has initialized. If you want to change the value of the timeout variable, it must be done before the dialog block.

**Related concepts**

- **Get program control on a regular (timed) basis** on page 1713
  Execute some code after a given number of seconds, with or without user interaction with the program.

- **ON IDLE block** on page 1465

**CONSTRUCT control instructions**

- **ACCEPT CONSTRUCT instruction**

The ACCEPT CONSTRUCT instruction validates the CONSTRUCT instruction and exits the dialog block if no error is raised.

The AFTER FIELD and AFTER CONSTRUCT control blocks will be executed.

The statements after the ACCEPT CONSTRUCT will not be executed.

```plaintext
CONSTRUCT BY NAME where_part ON ...
    ...
    ON ACTION default_query
        CALL set_default_filter()
        ACCEPT CONSTRUCT
    ...
END CONSTRUCT
```

The CONSTRUCT instruction creates the default accept action to let the user validate the dialog.

Use the ACCEPT CONSTRUCT instruction only in specific cases when the default accept action is not appropriate.
CONTINUE CONSTRUCT instruction

CONTINUE CONSTRUCT skips all subsequent statements in the current control block and gives the control back to the dialog.

This instruction is useful when program control is nested within multiple conditional statements, and you want to return the control to the dialog. If this instruction is called in a control block that is not AFTER CONSTRUCT, further control blocks might be executed depending on the context.

CONTINUE CONSTRUCT instructs the dialog to continue as if the code in the control block was terminated (therefore it acts as a kind of GOTO end_of_control_block). However, when executed in AFTER CONSTRUCT, the focus returns to the most recently occupied field in the current form, giving the user another chance to enter data in that field. In this case the BEFORE FIELD of the current field will be invoked.

As alternative, use the NEXT FIELD control instruction to give the focus to a specific field and force the dialog to continue. However, unlike CONTINUE CONSTRUCT, the NEXT FIELD instruction will skip the next control blocks that are normally executed.

EXIT CONSTRUCT instruction

The EXIT CONSTRUCT instruction terminates the CONSTRUCT instruction and resumes the program execution at the instruction following the CONSTRUCT block.

Performing an EXIT CONSTRUCT instruction during a dialog is equivalent to canceling the dialog: No field validation will occur, and the AFTER FIELD or AFTER CONSTRUCT blocks will not be executed.

Note: The INT_FLAG will not be set to TRUE as when the cancel action is fired.

NEXT FIELD instruction

Understanding the NEXT FIELD instruction

The NEXT FIELD field-name instruction gives the focus to the specified field. This instruction can be used to control field input, in BEFORE FIELD, ON CHANGE or AFTER FIELD blocks, it can also force a DISPLAY ARRAY or INPUT ARRAY to stay in the current row when NEXT FIELD is used in the AFTER ROW block.

If it exists, the BEFORE FIELD block of the corresponding field is executed.

In editable dialogs, the purpose of the NEXT FIELD instruction is to give the focus to an editable field. Make sure that the field specified in NEXT FIELD is active and editable, or use NEXT FIELD CURRENT.

In a DISPLAY ARRAY using the FOCUSONFIELD attribute, NEXT FIELD can be used in conjunction with DIALOG.setCurrentRow(), to set the focus to a specific cell in the list.

Instead of the NEXT FIELD instruction, you can use the DIALOG.nextField("field-name") method to register a field, for example when the name is not known at compile time. However, this method only registers the field. It does not stop code execution, like the NEXT FIELD instruction does. You must execute a CONTINUE DIALOG to get the same behavior as NEXT FIELD.
Form field identification with NEXT FIELD

With the NEXT FIELD instruction, fields are identified by the form field name specification, not the program variable name used by the dialog.

Form fields are bound to program variables with the binding clause of the dialog instruction (INPUT variable-list FROM field-list, INPUT BY NAME variable-list, CONSTRUCT BY NAME sql ON column-list, CONSTRUCT sql ON column-list FROM field-list, INPUT ARRAY array-name FROM screen-array.*).

The field name specification can be any of the following:

- field-name
- table-name.field-name
- screen-record-name.field-name
- FORMONLY.field-name

Here are some examples:

- "cust_name"
- "customer.cust_name"
- "cust_screen_record.cust_name"
- "item_screen_array.item_label"
- "formonly.total"

When no field name prefix is used, the first form field matching that simple field name is used.

When using a prefix in the field name specification, it must match the field prefix assigned by the dialog field binding method used at the beginning of the interactive statement: When no screen-record has been explicitly specified in the field binding clause (for example, when using INPUT BY NAME variable-list), the field prefix must be the database table name or FORMONLY, as defined in the form file, or any valid screen-record using that field. When the FROM clause of the dialog specifies an explicit screen-record (for example, in INPUT variable-list FROM screen-record.* / field-list-with-screen-record-prefix or INPUT ARRAY array-name FROM screen-array.*), the field prefix must be the screen-record name used in the FROM clause.

Abstract field identification is supported with the CURRENT, NEXT and PREVIOUS keywords. These keywords represent the current, next and previous fields respectively. When using FIELD ORDER FORM, the NEXT and PREVIOUS options follow the tabbing order defined by the form. Otherwise, they follow the order defined by the input binding list (with the FROM or BY NAME clause).

In a procedural dialog, if the focus is in the first field of an INPUT or CONSTRUCT sub-dialog, NEXT FIELD will jump out of the current sub-dialog and set the focus to the previous sub-dialog. If the focus is in the last field of an INPUT or CONSTRUCT sub-dialog, NEXT FIELD NEXT will jump out of the current sub-dialog and set the focus to the next sub-dialog. NEXT FIELD NEXT or NEXT FIELD PREVIOUS also jumps to another sub-dialog when the focus is in a DISPLAY ARRAY sub-dialog. However, when using an INPUT ARRAY sub-dialog, NEXT FIELD NEXT from within the last column will loop to the first column of the current row, and NEXT FIELD PREVIOUS from within the first column will jump to the last column of the current row - the focus stays in the current INPUT ARRAY sub-dialog. When another sub-dialog gets the focus because of a NEXT FIELD NEXT/ PREVIOUS, the newly-selected field depends on the sub-dialog type, following the tabbing order as if the end-user had pressed the tab or Shift-Tab key combination.

NEXT FIELD to a non-editable field

Non-editable fields are:

- Fields defined with the NOENTRY attribute in the form.
- Fields defined as PHANTOM fields in the form.
- Fields disabled at runtime with DIALOG.setFieldActive().
- Fields using a widget that does not allow input, such as a LABEL.
In an INPUT, INPUT ARRAY or CONSTRUCT instruction, if a NEXT FIELD instruction specifies a non-editable field, the BEFORE FIELD block of that field is executed. Then the dialog tries to give the focus to that field. Since the field cannot get the focus, the dialog will perform the last pressed navigation key (Tab, Shift-Tab, Left, Right, Up, Down, Accept) and execute the related control blocks, including the AFTER FIELD block of the non-editable field. If no last key is identified, the dialog considers Tab as fallback and moves to the next editable field as defined by the FIELD ORDER mode used by the dialog.

Note: In an INPUT ARRAY, doing a NEXT FIELD to a non-editable field which has no editable fields in the last navigation direction, the dialog may move to the next or previous row and give the focus to the first or last editable field of that row.

When selecting a non-editable field with NEXT FIELD NEXT, the runtime system will re-select the current field since it is the next editable field in the dialog. As a result the end user sees no change.

Important: Doing a NEXT FIELD to a non-editable field can lead to infinite loops in the dialog; To stay in the current field / row, use NEXT FIELD CURRENT instead.

NEXT FIELD in procedural DIALOG blocks

In a procedural dialog block, the NEXT FIELD field-name instruction gives the focus to the specified field controlled by INPUT, INPUT ARRAY or CONSTRUCT, or to a read-only list when using DISPLAY ARRAY.

When using a DISPLAY ARRAY sub-dialog, it is possible to give the focus to the list, by specifying the name of the first column as argument for NEXT FIELD.

If the target field specified in the NEXT FIELD instruction is inside the current sub-dialog, neither AFTER FIELD nor AFTER ROW will be invoked for the field or list you are leaving. However, the BEFORE FIELD control blocks of the destination field (or the BEFORE ROW in case of read-only list) will be executed.

If the target field specified in the NEXT FIELD instruction is outside the current sub-dialog, the AFTER FIELD, AFTER INSERT, AFTER ROW, and AFTER INPUT, AFTER DISPLAY, AFTER /CONSTRUCT control blocks will be invoked for the field or list you are leaving. Form-level validation rules will also be checked, as if the user had selected the new sub-dialog himself. This guarantees the current sub-dialog is left in a consistent state. The BEFORE INPUT, BEFORE DISPLAY, BEFORE CONSTRUCT, BEFORE ROW and the BEFORE FIELD control blocks of the destination field / list will then be executed.

NEXT FIELD in record list control blocks

When using NEXT FIELD in AFTER ROW or in ON ROW CHANGE of a DISPLAY ARRAY or INPUT ARRAY, the dialog will stay in the current row and give control back to the user.

This behavior allows you to implement data input rules:

```
AFTER ROW
  IF NOT int_flag AND arr_count()<=arr_curr() THEN
    IF arr[arr_curr()].it_count * arr[arr_curr()].it_value > maxval THEN
      ERROR "Amount of line exceeds max value."
    NEXT FIELD item_count
  END IF
END IF
```

If a condition is not met for a given field, do a NEXT FIELD to that field to point the user to the problem. If a global condition for the current row is not met, use NEXT FIELD CURRENT to stay in the current row and current field.

Related concepts

Giving the focus to a form element on page 1731
How to force the focus to move or stay in a specific form element using program code.

Understanding multiple dialogs on page 1585
Multiple dialogs are defined with DIALOG blocks inside a FUNCTION.

CLEAR instruction in dialogs

The CLEAR field-list and CLEAR SCREEN ARRAY screen-array.* instructions clear the value buffer of specified form fields. The buffers are directly changed in the current form, and the program variables bound to the dialog are left unchanged. CLEAR can be used outside any dialog instruction, such as the DISPLAY BY NAME / TO instructions.

When a dialog is configured with the UNBUFFERED mode, there is no reason to clear field buffers since any variable assignment will synchronize field buffers. Actually, changing the field buffers with DISPLAY or CLEAR instruction in an UNBUFFERED dialog will have no visual effect, because the variables bound to the dialog will be used to reset the field buffer just before giving control back to the user. To clear fields of an UNBUFFERED dialog, just set to NULL the variables bound to the dialog. However, when using a CONSTRUCT, no program variables are associated with the dialog and no UNBUFFERED concept exits, and the CLEAR or DISPLAY TO / BY NAME instructions are the only way to modify the CONSTRUCT fields.

A screen array with a screen-line specification doesn't make much sense in a GUI application using TABLE containers, you can therefore use the CLEAR SCREEN ARRAY instruction to clear all rows of a list.

Related concepts

Static display (DISPLAY/ERROR/MESSAGE/CLEAR) on page 1452

This section explains the instructions displaying static information to application forms, such as DISPLAY, ERROR, MESSAGE, CLEAR.

Examples

CONSTRUCT dialog usage examples.

Example 1: CONSTRUCT with binding by field position

Form definition file "form1.per":

```
SCHEMA office

LAYOUT
GRID
{
   Customer id: [f001 ]
   First Name : [f002 ]
   Last Name  : [f003 ]
}
END

TABLES
   customer
END

ATTRIBUTES
   f001 = customer.id;
   f002 = customer.fname;
   f003 = customer.lname, UPSHIFT;
END

INSTRUCTIONS
   SCREEN RECORD sr_cust(customer.*);
END
```

Program source code:

```
MAIN
   DEFINE condition STRING
   DATABASE office
```
OPEN FORM f1 FROM "form1"
DISPLAY FORM f1
CONSTRUCT condition
  ON id, fname, lname
  FROM sr_cust.*
  DISPLAY condition
END MAIN

Example 2: CONSTRUCT with binding by field name

Form definition file "form1.per":

SCHEMA office
LAYOUT
GRID
{
  Customer id: [f001 ]
  First Name : [f002 ]
  Last Name  : [f003 ]
}
END
END

TABLES
customer
END

ATTRIBUTES
  f001 = customer.id;
  f002 = customer.fname;
  f003 = customer.lname, UPSHIFT;
END

INSTRUCTIONS
  SCREEN RECORD sr_cust(customer.*);
END

Program source code:

SCHEMA office
MAIN
  DEFINE condition STRING
  DEFINE statement STRING
  DEFINE cust RECORD LIKE customer.*

  DATABASE office

  OPEN FORM f1 FROM "form1"
  DISPLAY FORM f1

  CONSTRUCT BY NAME condition ON customer.*
    BEFORE CONSTRUCT
      DISPLAY "A*" TO fname
      DISPLAY "B*" TO lname
    END CONSTRUCT

  LET statement = 
    "SELECT fname, lname FROM customer WHERE " || condition
    DISPLAY "SQL: " || statement

  DECLARE c1 CURSOR FROM statement
  FOREACH c1 INTO cust.*
Multiple dialogs (DIALOG - inside functions)

The procedural DIALOG instruction allows for the combination of record list, record input, and query criteria input in the same application form.

Understanding multiple dialogs

Multiple dialogs are defined with DIALOG blocks inside a FUNCTION.

The concept of multiple dialogs refers to the usage of a procedural DIALOG block, to control several elements of a form. During the execution of a procedural dialog, no other window/form can be accessed: multiple dialogs are in the category of modal dialogs.

The DIALOG procedural instruction handles different parts of a form simultaneously, including simple display fields, simple input fields, read-only list of records, editable list of records, query by example fields, and action views. The DIALOG instruction acts as a collection of singular dialogs working in parallel.
Figure 83: Query customers screenshot with multiple dialogs

"Singular interactive instructions" refer to INPUT, CONSTRUCT, DISPLAY ARRAY and INPUT ARRAY independent blocks not surrounded by the DIALOG / END DIALOG keywords. While the DIALOG instruction reuses some of the semantics and behaviors of singular interactive instructions, there are some differences.

Like the singular interactive instructions, DIALOG is an interactive instruction. You can execute a DIALOG instruction from one of the singular dialogs, or execute a singular dialog from a DIALOG block. The parent dialog will be disabled until the child dialog returns.

A DIALOG procedural instruction consist of several sub-dialog blocks declared inside the DIALOG instruction, or external dialog blocks declared in scope outside of the current function. The external dialogs are attached to the current dialog with the SUBDIALOG clause.

The DIALOG instruction binds program variables (such as simple records or arrays of records) with a screen-record or screen-array defined in a form, allowing the user to view and update application data.

When a DIALOG block executes, it activates the current form (the form most recently displayed or the form in the current window). When the statement completes execution, the form is deactivated.
The syntax of the `DIALOG` instruction is very close to singular dialogs, using common triggers such as `BEFORE FIELD`, `ON ACTION`, and so on. Despite the similarities, the behavior and semantics of `DIALOG` are a bit different from singular dialogs.

Understand that the `DIALOG` instruction is not provided to replace singular dialogs. Singular dialogs are still supported. It is recommended that you use singular dialogs if no multiple dialog is required.

Unlike singular dialogs, the `DIALOG` instruction does not use the `INT_FLAG` variable. You must implement `ON ACTION accept` or `ON ACTION cancel` to handle dialog validation or cancellation. These actions do not exist by default in `DIALOG`.

Unlike singular dialogs creating implicit accept and cancel actions, by default there is no way to quit the `DIALOG` instruction. You must implement your own action handler and execute `EXIT DIALOG` or `ACCEPT DIALOG`.

A good practice is to write a setup dialog function to centralize all field and action activations for a specific context. Call that setup function at any place in the `DIALOG` code where the field and action activation rules may change.

While static arrays are supported by the `DIALOG` instruction, it is strongly recommended that you use dynamic arrays instead. With a dynamic array, the actual number of rows is automatically defined by the array variable, while static arrays need an additional step to define the total number of rows.

When needed, use the `UNBUFFERED` mode with multiple dialogs to force model/view synchronization, and use the `FIELD ORDER FORM` option to follow the `TABINDEX` definitions in the form file.

This example is of a `DIALOG` procedural instruction that includes both an `INPUT` and a `DISPLAY ARRAY` sub-dialog, plus a sub-dialog defined externally and included with the `SUBDIALOG` keyword:

```plaintext
SCHEMA stores
DEFINE p_customer RECORD LIKE customer.*
DEFINE p_orders DYNAMIC ARRAY OF RECORD LIKE order.*
FUNCTION customer_dialog()
  DIALOG ATTRIBUTES(UNBUFFERED, FIELD ORDER FORM)
    INPUT BY NAME p_customer.*
    AFTER FIELD cust_name
    CALL setup_dialog(DIALOG)
  END INPUT
  DISPLAY ARRAY p_orders TO s_orders.*
    BEFORE ROW
    CALL setup_dialog(DIALOG)
  END DISPLAY
  SUBDIALOG common_footer
    ON ACTION close
    EXIT DIALOG
  END DIALOG
END FUNCTION
```

All elements of the dialog are active at the same time, so you must handle tabbing order properly. By default - as in singular dialogs - the tabbing order is driven by the binding list (order of program variables). It is strongly recommended that you use the `FIELD ORDER FORM` option and the `TABINDEX` field attributes instead.

Like the singular `INPUT ARRAY` instruction, `DIALOG` creates implicit insert, append, and delete actions. These actions are only active when the focus is in the list.

When the user moves from field to field, changes values, or browses a list, dialog control blocks such as `BEFORE FIELD`, `BEFORE ROW`, `BEFORE INPUT`, `BEFORE DISPLAY` are executed.

When the user clicks on an action view (button), or when an asynchronous event occurs, dialog interaction blocks like `ON ACTION` are executed.

The code inside a `DIALOG / END DIALOG` dialog can use control instructions, dialog control functions, and the `ui.Dialog` class, to implement the dialog behavior.

**Related concepts**

- Dialog programming basics on page 1706
This section describes basic dialog programming concepts.

**Declarative dialogs (DIALOG - at module level)** on page 1648
DIALOG/END DIALOG defined at module level implement declarative dialogs that can be used in procedural dialogs or in parallel dialogs.

**The buffered and unbuffered modes** on page 1720
The buffered and unbuffered mode control the synchronization of program variables and form fields.

**Syntax of the procedural DIALOG instruction**
The DIALOG block is an interactive instruction that executes several sub-dialogs simultaneously.

**Syntax**

```plaintext
DIALOG
  \[ ATTRIBUTES ( dialog-control-attribute \[,\ldots\] ) \]
  \[ record-input-block \]
  \[ construct-block \]
  \[ display-array-block \]
  \[ input-array-block \]
  \[ SUBDIALOG \[module-name,\]dialog-name \]
  \[
  \]
  \[\ldots\]
  \[ dialog-control-block \]
  \[
  \]
END DIALOG
```

where `dialog-control-attribute` is:

```plaintext
\[ FIELD ORDER FORM \]
\[ UNBUFFERED \[ = \] boolean \]
```

where `dialog-name` in the SUBDIALOG clause is the name of a declarative dialog block defined outside the scope of the current function, in another module identified by `module-name`.

where `dialog-control-block` is one of:

```plaintext
\[ BEFORE DIALOG \]
\[ ON ACTION \] action-name
  \[ ATTRIBUTES ( action-attributes-dialog ) \]
\[ ON KEY ( key-name \[,\ldots\] ) \]
\[ ON IDLE \] seconds
\[ ON TIMER \] seconds
\[ COMMAND \] option-name
  \[ option-comment \]
  \[ HELP help-number \]
\[ COMMAND KEY ( key-name \[,\ldots\] ) option-name \]
  \[ option-comment \]
  \[ HELP help-number \]
\[ AFTER DIALOG \]
\[ \]
dialog-statement
  \[
  \]
```

where *action-attributes-dialog* is:

```
| TEXT = string
| COMMENT = string
| IMAGE = string
| ACCELERATOR = string
| DEFAULTVIEW = YES | NO | AUTO |
| CONTEXTMENU = YES | NO | AUTO |
```

where *record-input-block* is:

```
INPUT BY NAME variable record.* [var,...] FROM field-list
  ATTRIBUTES ( input-control-attribute [,...] )
END INPUT
```

where *input-control-attribute* is:

```
HELP = help-number
NAME = "sub-dialog-name"
WITHOUT DEFAULTS [ = boolean ]
```

where *input-control-block* is one of:

```
BEFORE INPUT
BEFORE FIELD field-spec [,...]
ON CHANGE field-spec [,...]
AFTER FIELD field-spec [,...]
AFTER INPUT
ON ACTION action-name
  INFIELD field-spec
  ATTRIBUTES ( action-attributes-input )
ON KEY (key-name [,...])
dIALOG-STATEMENT [,...]
```

where *action-attributes-input* is:

```
| TEXT = string
| COMMENT = string
| IMAGE = string
| ACCELERATOR = string
| DEFAULTVIEW = YES | NO | AUTO |
| VALIDATE = NO
| CONTEXTMENU = YES | NO | AUTO |
```

where *construct-block* is:

```
CONSTRUCT BY NAME variable ON column-list
  variable ON column-list FROM field-list
  ATTRIBUTES ( construct-control-attribute [,...] )
  construct-control-block [,...]
```
where **construct-control-attribute** is:

```
HELP = help-number
NAME = "sub-dialog-name"
```

where **construct-control-block** is one of:

```
BEFORE CONSTRUCT
BEFORE FIELD field-spec [,....]
ON CHANGE field-spec [,....]
AFTER FIELD field-spec [,....]
AFTER CONSTRUCT
ON ACTION action-name
   [INFIELD field-spec]
   ON KEY ( key-name [,....] )
   dialog-statement
   [....]
```

where **action-attributes-construct** is:

```
TEXT = string
COMMENT = string
IMAGE = string
ACCELERATOR = string
DEFAULTVIEW = { YES | NO | AUTO }
CONTEXTMENU = { YES | NO | AUTO }
[....]
```

where **display-array-block** is:

```
DISPLAY ARRAY array TO screen-array.*
   ATTRIBUTES ( display-array-control-attribute [,....] )
   display-array-control-block
[....]
END DISPLAY
```

where **display-array-control-attribute** is:

```
HELP = help-number
COUNT = row-count
KEEP CURRENT ROW = [ = boolean ]
DETAILACTION = action-name
DOUBLECLICK = action-name
ACCESSORYTYPE = { DETAILBUTTON | DISCLOSUREINDICATOR | CHECKMARK }
FOCUSONFIELD
```

where **display-array-control-block** is one of:

```
BEFORE DISPLAY
BEFORE ROW
AFTER ROW
AFTER DISPLAY
ON ACTION action-name
   ATTRIBUTES ( action-attributes-display-array )
```
ON KEY (key-name [, ...])
ON FILL BUFFER
ON SELECTION CHANGE
ON SORT
ON APPEND ATTRIBUTES (action-attributes-listmod-triggers)
ON INSERT ATTRIBUTES (action-attributes-listmod-triggers)
ON UPDATE ATTRIBUTES (action-attributes-listmod-triggers)
ON DELETE ATTRIBUTES (action-attributes-listmod-triggers)
ON EXPAND (row-index)
ON COLLAPSE (row-index)
ON DRAG START (dnd-object)
ON DRAG FINISH (dnd-object)
ON DRAG ENTER (dnd-object)
ON DRAG OVER (dnd-object)
ON DROP (dnd-object)

dialog-statement[
  ...
]

where action-attributes-display-array is:

  TEXT = string
  COMMENT = string
  IMAGE = string
  ACCELERATOR = string
  DEFAULT VIEW = YES | NO | AUTO
  CONTEXT MENU = YES | NO | AUTO
  ROW BOUND[
    ...
  ]

where action-attributes-listmod-triggers is:

  TEXT = string
  COMMENT = string
  IMAGE = string
  ACCELERATOR = string
  DEFAULT VIEW = YES | NO | AUTO
  CONTEXT MENU = YES | NO | AUTO
  ...

where input-array-block is:

INPUT ARRAY array FROM screen-array.*[
  ATTRIBUTES (input-array-control-attribute [, ...])
  input-array-control-block[
    ...
  ]
END INPUT

where input-array-control-attribute is:

  APPEND ROW [= boolean]
  AUTO APPEND [= boolean]
  COUNT = row-count
  DELETE ROW [= boolean]
  HELP = help-number
  INSERT ROW [= boolean]
  KEEP CURRENT ROW [= boolean]
  MAX COUNT = max-row-count
  WITHOUT DEFAULTS [= boolean]
where \textit{input-array-control-block} is one of:

\begin{verbatim}
BEFORE INPUT
BEFORE ROW
BEFORE FIELD \[,...]\]
ON CHANGE field-spec \[,...]\]
AFTER FIELD field-spec \[,...]\]
ON ROW CHANGE
ON SORT
AFTER ROW
BEFORE DELETE
AFTER DELETE
BEFORE INSERT
AFTER INSERT
AFTER INPUT
ON ACTION action-name
\quad [INFIELD field-spec]
\quad \text{ATTRIBUTES ( action-attributes-input-array )}
\text{\quad ON KEY ( key-name \[,...\])}
\quad dialog-statement
\end{verbatim}

where \textit{action-attributes-input-array} is:

\begin{verbatim}
TEXT = string
COMMENT = string
IMAGE = string
ACCELERATOR = string
DEFAULTVIEW = \{ YES \mid NO \mid AUTO \}
VALIDATE = NO
CONTEXTMENU = \{ YES \mid NO \mid AUTO \}
ROWBOUND \[,...\]
\end{verbatim}

where \textit{dialog-statement} is one of:

\begin{verbatim}
statement
ACCEPT DIALOG
CANCEL DIALOG
CONTINUE DIALOG
EXIT DIALOG
NEXT FIELD
\quad CURRENT
\quad NEXT
\quad PREVIOUS
\quad field-spec
\end{verbatim}

where \textit{field-list} defines a list of fields with one or more of:

\begin{verbatim}
field-name
table-name.*
table-name.field-name
screen-array[line].*
screen-array[line].field-name
screen-record.*
screen-record.field-name
\[,...\]
\end{verbatim}
where *field-spec* identifies a unique field with one of:

```
  field-name
  table-name.field-name
  screen-array.field-name
  screen-record.field-name
```

where *column-list* defines a list of database columns as:

```
  column-name
  table-name.*
  table-name.column-name
  , ..., 
```

1. *variable-definition* is a variable declaration with data type as in a regular DEFINE statement.
2. *array* is the array of records used by the DIALOG statement.
3. *help-number* is an integer that allows you to associate a help message number with the command.
4. *field-name* is the identifier of a field of the current form.
5. *option-name* is a string expression defining the label of the action and identifying the action that can be executed by the user.
6. *option-comment* is a string expression containing a description for the menu option, displayed when *option-name* is the current.
7. *column-name* is the identifier of a database column of the current form.
8. *table-name* is the identifier of a database table of the current form.
9. *variable* is a simple program variable (not a record).
10. *record* is a program record (structured variable).
11. *screen-array* is the screen array that will be used in the current form.
12. *line* is a screen array line in the form.
13. *screen-record* is the identifier of a screen record of the current form.
14. *action-name* identifies an action that can be executed by the user.
15. *seconds* is an integer literal or variable that defines a number of seconds.
16. *key-name* is a hot-key identifier (like F11 or Control-z).
17. *row-index* identifies the program variable which holds the row index corresponding to the tree node that has been expanded or collapsed.
18. *dnd-object* references a *ui.DragDrop* variable defined in the scope of the dialog.
19. *statement* is any instruction supported by the language.
20. *action-attributes* are dialog-specific action attributes for the action.

**Procedural dialog programming steps**

Follow this procedure to use the DIALOG instruction.

To implement a procedural DIALOG block:

1. Create a form specification file containing screen record(s) and/or screen array(s). The screen records and screen arrays identify the presentation elements to be used by the runtime system to display the data models (the content of program variables bound to the DIALOG blocks).
2. With the DEFINE instruction, declare program variables (records and arrays) that will be used as data models. For record lists (DISPLAY ARRAY or INPUT ARRAY), the members of the program array must correspond to the elements of the screen array, by number and data types. To handle record lists, use dynamic arrays instead of static arrays.
3. Open and display the form, using OPEN WINDOW WITH FORM or the OPEN FORM/DISPLAY FORM instructions.
4. Fill the program variables (the model) with data. For lists, you typically use a result set cursor.
5. Implement the **DIALOG instruction block** to handle interaction. Define each sub-dialog with program variables to be used as data models. The sub-dialogs will define how variables will be used (display or input).
   a) Inside each sub-dialog instruction, define the behavior with control blocks such as **BEFORE DIALOG, AFTER ROW, BEFORE FIELD**, and interaction blocks such as **ON ACTION**.
   b) To end the **DIALOG** instruction, implement an **ON ACTION close or ON ACTION accept / ON ACTION cancel** to handle dialog validation and cancellation, with the **ACCEPT DIALOG** and **EXIT DIALOG** control instructions. The **INT_FLAG** variable will **not** be set as in singular dialogs.

### Related concepts

- **Form specification files** on page 1237
  Form specification files are the source files defining the layout and content of application forms.

- **Using multiple dialogs** on page 1594
  Dialog coding concepts, configuration and code structure.

### Using multiple dialogs

Dialog coding concepts, configuration and code structure.

#### Identifying sub-dialogs in procedural DIALOG

Sub-dialogs need to be identified by a name to distinguish the different contexts.

A procedural **DIALOG** block is a collection of sub-dialogs that act as controllers for different parts of a form. In order to program a procedural **DIALOG** block, there must be a unique identifier for each sub-dialog.

For example, to set the current row of a screen array with the **DIALOG.setCurrentRow()** method, you pass the name of the screen array to specify the sub-dialog to be affected. Sub-dialog identifiers are also used as a prefix to specify actions for the sub-dialog.

The following topics describe how to specify the names of the different types of **DIALOG** sub-dialogs:

- **Identifying an INPUT sub-dialog** on page 1596
- **Identifying a DISPLAY ARRAY sub-dialog** on page 1598
- **Identifying an INPUT ARRAY sub-dialog** on page 1599
- **Identifying a CONSTRUCT sub-dialog** on page 1597
- **The SUBDIALOG clause** on page 1600.

### Related concepts

- **Structure of a procedural DIALOG block** on page 1594
- **The Dialog class** on page 2512
  The **ui.Dialog** class provides a set of methods to configure, query and control the current interactive instruction.

- **Binding action views to action handlers** on page 1762
  How are action views of the forms bound to action handlers in the program code?

### Structure of a procedural DIALOG block

A procedural **DIALOG** instruction is made up of several **sub-dialogs**, plus global control blocks such as **BEFORE DIALOG** and action handlers such as **ON ACTION or COMMAND**.

Sub-dialogs can be defined inside the **DIALOG** instruction, or can be declared externally in another module and attached to the current **DIALOG** block with the **SUBDIALOG** clause. A dialog defined in the scope of a function is know as a **procedural dialog block**, while a dialog declared in the scope of a module is named a **declarative dialog block**.

The sub-dialogs bind program variables to form fields and define the type of interaction that will take place for the data model (simple input, list input or query). The sub-dialogs implement individual **control blocks** which let you control the behavior of the interactive instruction. Sub-dialogs can also hold action handlers, which will define local sub-dialog actions.

The **DIALOG** procedural instruction can hold the following type of sub-dialogs:

1. Simple record input with the **INPUT** sub-dialog block.
2. Query by example input with the **CONSTRUCT** sub-dialog block.
3. Read-only record list navigation with the DISPLAY ARRAY sub-dialog block.
4. Editable record list handling with the INPUT ARRAY sub-dialog block.
5. A SUBDIALOG clause referencing a declarative sub-dialog by name.

Related concepts
Declarative dialogs (DIALOG - at module level) on page 1648
DIALOG/END DIALOG defined at module level implement declarative dialogs that can be used in procedural dialogs or in parallel dialogs.

The INPUT sub-dialog
The INPUT sub-dialog implements single record input in fields of the current form.

Program variable to form field binding
Each record member variable is bound to the corresponding field of a screen record, in order to manipulate the values that the user enters in the form fields.

The INPUT clause can be used in two forms:
1. INPUT BY NAME variable-list
2. INPUT variable-list FROM field-list

The BY NAME clause implicitly binds the fields to the variables that have the same identifiers as the field names. The variables must be declared with the same names as the fields from which they accept input. The runtime system ignores any record name prefix when making the match. The unqualified names of the variables and of the fields must be unique and unambiguous within their respective domains. If they are not, the runtime system generates an exceptions, and sets the STATUS variable to a negative value.

```plaintext
DEFINE p_cust RECORD
    cust_num INTEGER,
    cust_name VARCHAR(50),
    cust_address VARCHAR(100)
END RECORD
...
DIALOG
    INPUT BY NAME p_cust.*
    BEFORE FIELD cust_name
    ...
END INPUT
...
END DIALOG

The FROM clause explicitly binds the fields in the screen record to a list of program variables by position. The number of variables or record members must equal the number of fields listed in the FROM clause. Each variable must be of the same (or a compatible) data type as the corresponding screen field. When the user enters data, the runtime system checks the entered value against the data type of the variable, not the data type of the screen field.

```plaintext
DEFINE c_name VARCHAR(50)
    c_addr VARCHAR(100)
...
DIALOG
    INPUT c_name,
    c_addr
    FROM FORMONLY.field01,
    FORMONLY.field02
    BEFORE FIELD cust_name
    ...
END INPUT
...
END DIALOG
```
Identifying an INPUT sub-dialog

The name of an INPUT sub-dialog can be used to qualify sub-dialog actions with a prefix.

In order to identify the INPUT sub-dialog with a specific name, you can use the ATTRIBUTES clause to set the NAME attribute:

```
INPUT BY NAME p_cust.*
  ATTRIBUTES (NAME = "cust")
...
```

Control blocks in INPUT

Simple record input declared with the INPUT sub-dialog can raise the following triggers:

- BEFORE INPUT
- BEFORE FIELD
- ON CHANGE
- AFTER FIELD
- AFTER INPUT

In the singular INPUT instruction, BEFORE INPUT and AFTER INPUT blocks are typically used as initialization and finalization blocks. In an INPUT sub-dialog of a DIALOG block, BEFORE INPUT and AFTER INPUT blocks will be executed each time the focus goes to (BEFORE) or leaves (AFTER) the group of fields defined by this sub-dialog.

Related concepts

INPUT ATTRIBUTES clause on page 1602

INPUT specific attributes can be defined in the ATTRIBUTE clause of the sub-dialog header.

The CONSTRUCT sub-dialog

The CONSTRUCT sub-dialog provides database query by example feature, converting search criteria entered by the user into an SQL WHERE condition that can be used to execute a SELECT statement.

Defining query by example fields

The CONSTRUCT sub-dialog requires a character string variable to hold the WHERE clause, and a list of screen fields where the user can enter search criteria.

```
DEFINE sql_condition STRING
  ...
DIALOG
  CONSTRUCT BY NAME sql_condition
    ON customer.cust_name, customer.cust_address
    BEFORE FIELD cust_name
     ...
  END CONSTRUCT
  ...
END DIALOG
```

Make sure the character string variable is large enough to store all possible SQL conditions. It is better to use a STRING data type to avoid any size problems.

CONSTRUCT uses the field data types defined in the current form file to produce the SQL conditions. This is different from other interactive instructions, where the data types of the program variables define the way to handle input/display. It is strongly recommended (but not mandatory) that the form field data types correspond to the data types of the program variables used for input. This is implicit if both form fields and program variables are based on the database schema file.

The CONSTRUCT clause can be used in two forms:
1. **CONSTRUCT** BY NAME *string-variable* ON *column-list*
2. **CONSTRUCT** *string-variable* ON *column-list* FROM *field-list*

The `BY NAME` clause implicitly binds the form fields to the columns, where the form field identifiers match the column names specified in the column-list after the `ON` keyword. You can specify the individual column names (separated by commas) or use the `tablename.*` shortcut to include all columns defined for a table in the database schema file.

The `FROM` clause explicitly binds the form fields listed after the `FROM` keyword with the column definitions listed after the `ON` keyword.

In both cases, the name of the columns in *column-list* will be used to produce the SQL condition in *string-variable*.

**Identifying a CONSTRUCT sub-dialog**

The name of a CONSTRUCT sub-dialog can be used to qualify sub-dialog actions with a prefix. In order to identify the CONSTRUCT sub-dialog with a specific name, use the `ATTRIBUTES` clause to set the `NAME` attribute:

```plaintext
CONSTRUCT BY NAME sql_condition ON customer.*
   ATTRIBUTES (NAME = "q_cust")
   ...
```

**Control blocks in CONSTRUCT**

A Query By Example declared with the CONSTRUCT clause can raise the following triggers:

- BEFORE CONSTRUCT
- BEFORE FIELD
- AFTER FIELD
- AFTER CONSTRUCT

In the singular CONSTRUCT instruction, BEFORE CONSTRUCT and AFTER CONSTRUCT blocks are typically used as initialization and finalization blocks. In DIALOG block, BEFORE CONSTRUCT and AFTER CONSTRUCT blocks will be executed each time the focus goes to (BEFORE) or leaves (AFTER) the group of fields defined by this sub-dialog.

**Related concepts**

- Query operators in CONSTRUCT on page 1569
- CONSTRUCT ATTRIBUTES clause on page 1605

CONSTRUCT specific attributes can be defined in the ATTRIBUTE clause of the sub-dialog header.

**The DISPLAY ARRAY sub-dialog**

The DISPLAY ARRAY sub-dialog is the controller to implement the navigation in a list of records, with option data modification actions.

**Program array to screen array binding**

The DISPLAY ARRAY sub-dialog binds the members of the flat record (or the primitive member) of an array to the screen-array or screen-record fields specified with the `TO` keyword.

The number of variables in each record of the program array must be the same as the number of fields in each screen record (that is, in a single row of the screen array).

You typically bind a program array to a screen-array in order to display a page of records. However, the DIALOG instruction can also bind the program array to a simple flat screen-record. In this case, only one record will be visible at a time.

The next code example defines an array with a flat record and binds it to a screen array:

```plaintext
DEFINE p_items DYNAMIC ARRAY OF RECORD
    item_num INTEGER,
```
If the screen array is defined with one field only, you can bind an array defined with a primitive type:

```plaintext
DEFINE p_names DYNAMIC ARRAY OF VARCHAR(50)
...
DIALOG
  DISPLAY ARRAY p_names TO sa.*
  BEFORE DELETE
  ...
END DISPLAY
... END DIALOG
```

### Identifying a DISPLAY ARRAY sub-dialog

The name of the screen array specified with the **TO** clause identifies the list. The dialog class method takes the name of the screen array as the parameter, identifying the list. For example, you would use `DIALOG.getCurrentRow("screen-array")` to query for the current row in the list identified by 'screen-array'. The name of the screen-array is also used to qualify **sub-dialog actions** with a prefix.

### Control blocks in DISPLAY ARRAY

Read-only record lists declared with the **DISPLAY ARRAY** sub-dialog can raise the following triggers:

- **BEFORE DISPLAY**
- **BEFORE ROW**
- **AFTER ROW**
- **AFTER DISPLAY**

In the singular **DISPLAY ARRAY** instruction, **BEFORE DISPLAY** and **AFTER DISPLAY** blocks are typically used as initialization and finalization blocks. In a **DISPLAY ARRAY** sub-dialog of a **DIALOG** block, **BEFORE DISPLAY** and **AFTER DISPLAY** blocks will be executed each time the focus goes to (**BEFORE**) or leaves (**AFTER**) the group of fields defined by this sub-dialog.

### Related concepts

**DISPLAY ARRAY ATTRIBUTES clause** on page 1603

**DISPLAY ARRAY** specific attributes can be defined in the **ATTRIBUTE** clause of the sub-dialog header.

### The INPUT ARRAY sub-dialog

The **INPUT ARRAY** sub-dialog is the controller to implement the navigation and edition in a list of records.

**Important:** This feature is not supported on mobile platforms.

### Program array to screen array binding

The **INPUT ARRAY** sub-dialog binds the members of the flat record (or the primitive member) of an array to the **screen-array or screen-record** fields specified with the **FROM** keyword. The number of variables in each record of the program array must be the same as the number of fields in each screen record (that is, in a single row of the screen array).
You typically bind a program array to a screen-array in order to display a page of records. However, the DIALOG instruction can also bind the program array to a simple flat screen-record. In this case, only one record will be visible at a time.

The next code example defines an array with a flat record and binds it to a screen array:

```dialog
DEFINE p_items DYNAMIC ARRAY OF RECORD
  item_num INTEGER,
  item_name VARCHAR(50),
  item_price DECIMAL(6,2)
END RECORD
...
DIALOG
  INPUT ARRAY p_items FROM sa.*
    BEFORE INSERT
    ...
  END INPUT
  ...
END DIALOG
```

If the screen array is defined with one field only, you can bind an array defined with a primitive type:

```dialog
DEFINE p_names DYNAMIC ARRAY OF VARCHAR(50)
...
DIALOG
  INPUT ARRAY p_names FROM sa.*
    BEFORE DELETE
    ...
  END INPUT
  ...
END DIALOG
```

**Identifying an INPUT ARRAY sub-dialog**

The name of the screen array specified with the FROM clause will be used to identify the list. For example, the dialog class method such as `DIALOG.getCurrentRow("screen-array")` takes the name of the screen array as the parameter, to identify the list you want to query for the current row. The name of the screen-array is also used to qualify sub-dialog actions with a prefix.

**Control blocks in INPUT ARRAY**

Editable record lists declared with the INPUT ARRAY sub-dialog can raise the following triggers:

- BEFORE INPUT
- BEFORE ROW
- BEFORE FIELD
- ON CHANGE
- AFTER FIELD
- ON ROW CHANGE
- AFTER ROW
- BEFORE DELETE
- AFTER DELETE
- BEFORE INSERT
- AFTER INSERT
- AFTER INPUT

In the singular INPUT ARRAY instruction, BEFORE INPUT and AFTER INPUT blocks are typically used as initialization and finalization blocks. In the INPUT ARRAY sub-dialog of a DIALOG block, BEFORE INPUT and
AFTER INPUT blocks are executed each time the focus goes to (BEFORE) or leaves (AFTER) the group of fields defined by this sub-dialog.

Related concepts

INPUT ARRAY ATTRIBUTES clause on page 1603

INPUT ARRAY specific attributes can be defined in the ATTRIBUTE clause of the sub-dialog header.

The SUBDIALOG clause

Purpose of SUBDIALOG

The SUBDIALOG clause defines a declarative dialog to be attached to the current procedural DIALOG block.

By using form inclusion (with the FORM clause in LAYOUT sections) and declarative dialogs + SUBDIALOG, you enforce code reusability in your application sources.

Note: Declarative dialog blocks can also be used to implement parallel dialogs.

Defining the declarative dialog

The declarative dialog is implemented outside the scope of the using DIALOG block, at the same level as a function.

The declarative dialog can be defined in a different module, to be reused in other DIALOG instructions. The sub-dialog module must be imported with the IMPORT FGL instruction.

Like other module elements such as functions and reports, the name specification is mandatory when defining a declarative dialog. The name of the declarative dialog will be referenced in a SUBDIALOG clause of a procedural dialog instruction.

In the "comment.4gl" module:

```
DIALOG comment_input()
  ...  
END DIALOG
```

In the using module (note that we use the module prefix here):

```
IMPORT FGL comment
  ...
FUNCTION mydialog()
  DIALOG ...
    ...
    SUBDIALOG comment.comment_input
    ...
  END DIALOG
END FUNCTION
```

See also Identifying sub-dialogs in procedural DIALOG on page 1594.

Sub-dialogs in form definitions

Implementing a sub-dialog as a declarative dialog in a separate module is typically used in conjunction with the FORM clause, in the LAYOUT section of form specification files:

```
LAYOUT
  ...
FORM "comment"
  ...
END
```
Semantics with SUBDIALOG

In terms of semantics, behavior and control block execution, a declarative dialog attached to a procedural dialog with SUBDIALOG, behaves like a sub-dialog that is defined inside the procedural DIALOG block.

For example, BEFORE INPUT inside an INPUT block of a declarative dialog will be executed when the focus goes to one of the fields of that sub-dialog.

Scope of dialog instructions

Other sub-dialogs can reference the attached declarative dialog in the current scope.

For example, to execute a NEXT FIELD instruction referencing a field in another sub-dialog:

```
DIALOG ... -- Parent dialog block
...  
    NEXT FIELD the_comment  -- Field of the declarative dialog.
...  
END DIALOG
```

Scope of DIALOG keyword

When using the DIALOG keyword inside a declarative dialog block to use ui.Dialog class methods, it references the current procedural dialog object:

```
DIALOG comment_input()  
...  
    CALL DIALOG.setFieldActive("the_comment",TRUE)  
...  
END DIALOG
```

Writing generic code

To be reused by different procedural DIALOG instructions, the code of sub-dialog modules must be generic. However, if the sub-dialog code needs to interact with the parent DIALOG, it must be possible to call a function from the parent DIALOG.

You achieve this by using function references. Parent modules can then configure the sub-dialog module at runtime, with callback functions:

1. Create a user-defined TYPE with the FUNCTION type matching the callback function of the using module.
2. Define a module variable, with the declared function type. If you want to keep it private to the module, define a setter function to assign the variable with the callback function reference.
3. When the parent DIALOG needs to be notified by some change in the sub-dialog, check that the callback variable is not NULL, and call the function with appropriate values.

Tip: You do not have to implement a lot of complex callback functions. The main purpose is to indicate to the parent DIALOG, that something happened in the sub-dialog. The parent DIALOG can then query the sub-dialog module for more information, as long as the sub-dialog module provides functions to query its status.

```
PUBLIC TYPE cb_comment_event FUNCTION (event STRING)  
PRIVATE DEFINE cb_ce cb_comment_event  
...  
PUBLIC FUNCTION set_event_callback(f cb_comment_event)  
    LET cb_ce = f  
END FUNCTION  

DIALOG comment_input()  
...  
    IF cb_ce IS NOT NULL THEN  
    CALL cb_ce("comment_changed")
```
For a complete example, see Example 3: DIALOG with SUBDIALOG on page 1646.

Procedural DIALOG block configuration

This section describes the ATTRIBUTES clause attributes that can be used to configure a procedural DIALOG instruction and its sub-dialogs.

The ATTRIBUTES clause of dialogs overrides all default attributes and temporarily overrides any display attributes that the OPTIONS or the OPEN WINDOW statement specified for these fields.

DIALOG ATTRIBUTES clause

FIELD ORDER FORM option

By default, the form tabbing order is defined by the variable list in the binding specification. You can control the tabbing order by using the FIELD ORDER FORM attribute; when this attribute is used, the tabbing order is defined by the TABINDEX attribute of the form items.

The field order mode can also be specified globally with the OPTIONS FIELD ORDER instruction.

With FIELD ORDER FORM, if the user changes the focus from field A to a distant field B with the mouse, the dialog does not execute the BEFORE FIELD / AFTER FIELD triggers of intermediate fields which appear in the binding specification between field A and field B. Unlike singular dialogs, if the default FIELD ORDER CONSTRAINT mode is used in a multiple dialog instruction, intermediate triggers are never executed (i.e. the Dialog.fieldOrder FQLPROFILE entry is ignored by DIALOG.)

See also Defining the tabbing order on page 1729.

UNBUFFERED option

The UNBUFFERED attribute indicates that the dialog is sensitive to program variable changes. When using this option, you bypass the compatible "buffered" mode.

The unbuffered mode can be set globally for all DIALOG instructions with the ui.Dialog.setDefaultUnbuffered() class method:

```
CALL ui.Dialog.setDefaultUnbuffered(TRUE)
DIALOG -- Will work in UNBUFFERED mode ... END DIALOG
```

INPUT ATTRIBUTES clause

INPUT specific attributes can be defined in the ATTRIBUTE clause of the sub-dialog header.

HELP option

The HELP attribute defines the number of the help message to be displayed when invoked and focus is in the list controlled by the INPUT sub-dialog. The predefined 'help' action is automatically created by the runtime system. You can bind action views to the 'help' action. The HELP clause overrides the HELP attribute.

NAME option

The NAME attribute can be used to identify the INPUT sub-dialog, especially useful to qualify sub-dialog actions

WITHOUT DEFAULTS option

By default, sub-dialogs use the default values defined in the form files. If you want to use the values stored in the program variables bound to the dialog, you must use the WITHOUT DEFAULTS attribute. For more details see WITHOUT DEFAULTS option.
**DISPLAY ARRAY ATTRIBUTES clause**
DISPLAY ARRAY specific attributes can be defined in the ATTRIBUTE clause of the sub-dialog header.

**HELP option**
The HELP attribute defines the number of the help message to be displayed when invoked and focus is in the list controlled by the DISPLAY ARRAY sub-dialog. The predefined 'help' action is automatically created by the runtime system. You can bind action views to the 'help' action.

The HELP clause overrides the HELP attribute.

**COUNT option**
The COUNT attribute defines the number of valid rows in the static array to be displayed as default rows. If you do not use the COUNT attribute, the runtime system cannot determine how much data to display, so the screen array remains empty. The COUNT option is ignored when using a dynamic array, unless page mode is used. In this case, the COUNT attribute must be used to define the total number of rows, because the dynamic array will only hold a page of the entire row set. If the value of COUNT is negative or zero, it defines an empty list.

See also Controlling the number of rows on page 1834.

**DOUBLECLICK option**
The DOUBLECLICK option can be used to define the action that will be fired when the user chooses a row from the list. Different configuration options are available to control the row selection action of desktop and mobile devices. For more details, see Defining the action for a row choice on page 1867.

**ACCESSORYTYPE option**
**Important:** This feature is only for mobile platforms.

The accessortype attribute can be used to define the decoration of rows, typically used on an iOS device. Values can be DETAILBUTTON, DISCLOSUREINDICATOR, CHECKMARK to respectively get an (i), > or check mark icon. For more details, see Row configuration on iOS devices on page 1876.

**DETAILACTION option**
**Important:** This feature is only for mobile platforms.

The DETAILACTION attribute can be used to define the action that will be fired when the user selects the detail button of a row. The detail button is typically shown with an (i) icon on iOS devices. Note that the DOUBLECLICK attribute can be used to distinguish the action when the user selects the row instead of the detail button in the row. For more details, see Row configuration on iOS devices on page 1876.

**FOCUSONFIELD option**
**Important:** This feature is not supported on mobile platforms.

When the FOCUSONFIELD option is used, the DISPLAY ARRAY allows focus at the field (or cell) level. It it then possible to implement BEFORE FIELD and AFTER FIELD blocks, as well as using NEXT FIELD instructions. However, the dialog still manages a read-only list. For more details, see Field-level focus in DISPLAY ARRAY on page 1840.

**INPUT ARRAY ATTRIBUTES clause**
INPUT ARRAY specific attributes can be defined in the ATTRIBUTE clause of the sub-dialog header.
HELP option
The HELP clause specifies the number of a help message to display if the user invokes the help the INPUT ARRAY dialog. The predefined 'help' action is automatically created by the runtime system. You can bind action views to the 'help' action. The HELP clause overrides the HELP attribute.

COUNT option
The COUNT attribute defines the number of valid rows in the static array to be displayed as default rows. If you do not use the COUNT attribute, the runtime system cannot determine how much data to display, so the screen array remains empty. The COUNT option is ignored when using a dynamic array. If you specify the COUNT attribute, the WITHOUT DEFAULTS option is not required because it is implicit. If the COUNT attribute is greater than MAXCOUNT, the runtime system will take MAXCOUNT as the actual number of rows. If the value of COUNT is negative or zero, it defines an empty list.

MAXCOUNT option
The MAXCOUNT attribute defines the maximum number of rows that can be inserted in the program array. This attribute allows you to give an upper limit of the total number of rows the user can enter. It can be used with static or dynamic arrays.

When binding a static array, MAXCOUNT is used as upper limit if it is lower or equal to the actual declared static array size. If MAXCOUNT is greater than the array size, the size of the static array is used as the upper limit. If MAXCOUNT is lower than the COUNT attribute (or to the SET_COUNT() parameter when using a singular INPUT ARRAY), the actual number of rows in the array will be reduced to MAXCOUNT.

When binding a dynamic array, the user can enter an infinite number of rows unless the MAXCOUNT attribute is used. If MAXCOUNT is lower than the actual size of the dynamic array, the number of rows in the array will be reduced to MAXCOUNT.

If MAXCOUNT is negative or equal to zero, the user cannot insert rows.

APPEND ROW option
The APPEND ROW attribute can be set to FALSE to avoid the append default action, and deny the user to add rows at the end of the list. If APPEND ROW = FALSE, it is still possible to insert rows in the middle of the list. Use the INSERT ROW attribute to disallow the user from inserting rows. Additionally, even with APPEND ROW=FALSE and INSERT ROW=FALSE, you can still get automatic temporary row creation if AUTO APPEND is not set to FALSE.

INSERT ROW option
The INSERT ROW attribute can be set to FALSE to avoid the insert default action, and deny the user to insert new rows in the middle of the list. However, even if INSERT ROW is FALSE, it is still possible to append rows at the end of the list. Use the APPEND ROW attribute to disallow the user from appending rows. Additionally, even with APPEND ROW=FALSE and INSERT ROW=FALSE, you can still get automatic temporary row creation if AUTO APPEND is not set to FALSE.

DELETE ROW option
The DELETE ROW attribute can be set to FALSE to avoid the delete default action, and deny the user to remove rows from the list.

AUTO APPEND option
By default, an INPUT ARRAY controller creates a temporary row when needed (for example, when the user deletes the last row of the list, an new row will be automatically created). You can prevent this default behavior by setting the AUTO APPEND attribute to FALSE. When this attribute is set to FALSE, the only way to create a new temporary row is to execute the append action.
If both the APPEND ROW and INSERT ROW attributes are set to FALSE, the dialog automatically behaves as if AUTO APPEND equals FALSE.

**KEEP CURRENT ROW option**

Depending on the list container used in the form, the current row may be highlighted during the execution of the dialog, and cleared when the instruction ends. You can change this default behavior by using the KEEP CURRENT ROW attribute, to force the runtime system to keep the current row highlighted.

**WITHOUT DEFAULTS option**

You typically use the INPUT ARRAY sub-dialog with the WITHOUT DEFAULTS attribute. If this attribute is not set when using an INPUT ARRAY sub-dialog, the list is empty even if the array holds data. For more details see WITHOUT DEFAULTS option.

**CONSTRUCT ATTRIBUTES clause**

CONSTRUCT specific attributes can be defined in the ATTRIBUTE clause of the sub-dialog header.

**HELP option**

The HELP attribute defines the number of the help message to be displayed when invoked and focus is in the list controlled by the CONSTRUCT sub-dialog. The predefined 'help' action is automatically created by the runtime system. You can bind action views to the 'help' action.

The HELP clause overrides the HELP attribute.

**NAME option**

The NAME attribute can be used to identify the CONSTRUCT sub-dialog; this is especially useful to qualify sub-dialog actions.

**Default actions created by a DIALOG block**

Default actions ease the implementation of the controller by providing expected actions.

The runtime system creates a set of default actions based on the sub-dialogs defined in a (declarative or procedural) DIALOG block. These actions are provided to ease the implementation of the controller. For example, when using an INPUT ARRAY sub-dialog, the dialog instruction will automatically create the insert, append and delete default actions.

**Table 378: Default actions created for the DIALOG block on page 1605** lists the default actions created for the DIALOG interactive instruction, for each type of sub-dialogs:

<table>
<thead>
<tr>
<th>Default action</th>
<th>Control Block execution order</th>
</tr>
</thead>
<tbody>
<tr>
<td>help</td>
<td>Shows the help topic defined by the HELP clause. Only created when a HELP clause or option is defined for the sub-dialog.</td>
</tr>
<tr>
<td>insert</td>
<td>Inserts a new row before current row. Only for INPUT ARRAY dialogs. Action creation can be avoided with INSERT ROW = FALSE attribute.</td>
</tr>
<tr>
<td>append</td>
<td>Appends a new row at the end of the list. Only for INPUT ARRAY dialogs. Action creation can be avoided with APPEND ROW = FALSE attribute.</td>
</tr>
</tbody>
</table>
### Default action

<table>
<thead>
<tr>
<th>Default action</th>
<th>Control Block execution order</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete</td>
<td>Deletes the current row. Only for INPUT ARRAY dialogs. Action creation can be avoided with DELETE ROW = FALSE attribute.</td>
</tr>
<tr>
<td>nextrow</td>
<td>Moves to the next row in a list displayed in one row of fields. See note (1).</td>
</tr>
<tr>
<td>prevrow</td>
<td>Moves to the previous row in a list displayed in one row of fields. See note (1).</td>
</tr>
<tr>
<td>firstrow</td>
<td>Moves to the first row in a list displayed in one row of fields. See note (1).</td>
</tr>
<tr>
<td>lastrow</td>
<td>Moves to the last row in a list displayed in one row of fields. See note (1).</td>
</tr>
<tr>
<td>find</td>
<td>Opens the fglfind dialog window to let the user enter a search value, and seeks to the row matching the value. See note (2).</td>
</tr>
<tr>
<td>findnext</td>
<td>Seeks to the next row matching the value entered during the fglfind dialog. See note (2).</td>
</tr>
</tbody>
</table>

### Notes:

1. The action is only created with a DISPLAY ARRAY or INPUT ARRAY using a screen record bound to a set of form fields in a GRID container, and this set of fields show only a single row of the list. The action is not created when using a screen array bound to a list container such as TABLE, TREE and SCROLLGRID, or to a set of fields in a GRID container, that repeat on several lines to show more than one single row.

2. The action is only created if the context allows built-in find.

The insert, append and delete default actions can be avoided with dialog control attributes:

```
INPUT ARRAY arr TO sr.* ATTRIBUTES( INSERT ROW=FALSE, APPEND ROW=FALSE, ... )
...```

### Related concepts

- **DISPLAY ARRAY ATTRIBUTES clause** on page 1603
- **INPUT ARRAY ATTRIBUTES clause** on page 1603

### Dialog data blocks

*Dialog data blocks* are dialog triggers invoked when the dialog controller needs data to feed the view with values.

Such blocks are typically used when record list data is provided dynamically, with the paged mode or when implementing dynamic tree-views.
**ON FILL BUFFER block**

The **ON FILL BUFFER** block is used to fill a page of rows into the dynamic array, based on an offset and a number of rows.

This data block is only used in **DISPLAY ARRAY** dialog blocks.

The offset can be retrieved with the `FGL_DIALOG_GETBUFFERSTART()` built-in function and the number of rows to provide is defined by the `FGL_DIALOG_GETBUFFERLENGTH()` built-in function.

The **ON FILL BUFFER** block is executed when the runtime system needs data rows to fill the current page of the list dialog. This can happen before a **BEFORE DISPLAY** of a singular **DISPLAY ARRAY**, or before the **BEFORE DIALOG** block of a **DIALOG / END DIALOG** instruction containing **DISPLAY ARRAY** sub-dialogs.

For more details about **ON FILL BUFFER** usage, see Paged mode of **DISPLAY ARRAY** on page 1843.

**ON EXPAND block**

The **ON EXPAND** block is executed when a tree view node is expanded (opened).

This data block is used to implement dynamic trees in a **DISPLAY ARRAY**, where nodes are added, depending on the nodes opened by the end user.

For more details, see Dynamic filling of very large trees on page 1896.

**Related concepts**

**ON COLLAPSE block** on page 1508

**ON COLLAPSE block**

The **ON COLLAPSE** block is executed when a tree view node is collapsed (i.e. closed).

This data block is used to implement dynamic trees in a **DISPLAY ARRAY**, where nodes are removed from view by the user closing or collapsing them.

For more details, see Dynamic filling of very large trees on page 1896.

**Related concepts**

**ON EXPAND block** on page 1508

**DIALOG control blocks**

**Dialog control blocks** are predefined dialog triggers where you can implement specific code to control the interactive instruction.

The code may involve using `ui.Dialog` class methods or dialog specific instructions such as **NEXT FIELD** or **CONTINUE DIALOG**.

**Control block execution order in multiple dialogs**

This table shows the order in which control blocks are executed in a procedural **DIALOG** instruction, depending on the context and user action:

**Table 379: Control block execution order for a procedural dialog**

<table>
<thead>
<tr>
<th>Context / User action</th>
<th>Control Block execution order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entering the dialog</td>
<td>1. BEFORE DIALOG</td>
</tr>
<tr>
<td></td>
<td>2. BEFORE INPUT, BEFORE CONSTRUCT or BEFORE DISPLAY (first sub-dialog getting focus)</td>
</tr>
<tr>
<td></td>
<td>3. BEFORE ROW (if focus goes to a list)</td>
</tr>
<tr>
<td></td>
<td>4. BEFORE FIELD (if focus goes to a field)</td>
</tr>
<tr>
<td>Context / User action</td>
<td>Control Block execution order</td>
</tr>
<tr>
<td>-------------------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>When the focus goes to an INPUT or to a CONSTRUCT from a different sub-dialog</td>
<td>1. <em>Triggers raised by the context of the sub-dialog you leave</em> \</td>
</tr>
<tr>
<td></td>
<td>2. BEFORE INPUT or BEFORE CONSTRUCT (new sub-dialog getting focus) \</td>
</tr>
<tr>
<td></td>
<td>3. BEFORE FIELD \</td>
</tr>
<tr>
<td>When the focus leaves an INPUT or a CONSTRUCT to a different sub-dialog</td>
<td>1. <em>ON CHANGE</em> (if INPUT and value of current field has changed) \</td>
</tr>
<tr>
<td></td>
<td>2. AFTER FIELD (for the current field) \</td>
</tr>
<tr>
<td></td>
<td>3. AFTER INPUT or AFTER CONSTRUCT (current sub-dialog losing focus) \</td>
</tr>
<tr>
<td></td>
<td>4. <em>Triggers raised by the context of the sub-dialog you enter</em> \</td>
</tr>
<tr>
<td>When the focus goes to a DISPLAY ARRAY list or to an INPUT ARRAY list from a</td>
<td>1. <em>Triggers raised by the context of the sub-dialog you leave</em> \</td>
</tr>
<tr>
<td>different sub-dialog</td>
<td>2. BEFORE INPUT or BEFORE DISPLAY (new sub-dialog getting focus) \</td>
</tr>
<tr>
<td></td>
<td>3. BEFORE ROW (the row that was selected in the list) \</td>
</tr>
<tr>
<td></td>
<td>4. BEFORE FIELD (if it’s an INPUT ARRAY) \</td>
</tr>
<tr>
<td>When the focus leaves a DISPLAY ARRAY or INPUT ARRAY list to a different sub-dialog</td>
<td>1. <em>ON CHANGE</em> (if INPUT ARRAY and value of current field has changed) \</td>
</tr>
<tr>
<td></td>
<td>2. AFTER FIELD (if it’s an INPUT ARRAY) \</td>
</tr>
<tr>
<td></td>
<td>3. AFTER INSERT (if INPUT ARRAY and current row was just created) \</td>
</tr>
<tr>
<td></td>
<td>or \</td>
</tr>
<tr>
<td></td>
<td><em>ON ROW CHANGE</em> (if INPUT ARRAY and a value in the row has changed) \</td>
</tr>
<tr>
<td></td>
<td>4. AFTER ROW (the current row in the list you leave) \</td>
</tr>
<tr>
<td></td>
<td>5. AFTER INPUT or AFTER DISPLAY (current sub-dialog losing focus) \</td>
</tr>
<tr>
<td></td>
<td>6. <em>Triggers raised by the context of the sub-dialog you enter</em> \</td>
</tr>
<tr>
<td>Moving from field A to field B in an INPUT or CONSTRUCT sub-dialog or in the same</td>
<td>1. <em>ON CHANGE</em> (if value of current field has changed) \</td>
</tr>
<tr>
<td>row of an INPUT ARRAY list</td>
<td>2. AFTER FIELD A \</td>
</tr>
<tr>
<td></td>
<td>3. BEFORE FIELD B \</td>
</tr>
<tr>
<td>Moving from field A of an INPUT or CONSTRUCT sub-dialog to field B in another</td>
<td>1. <em>ON CHANGE</em> (if value of current field has changed) \</td>
</tr>
<tr>
<td>INPUT or CONSTRUCT sub-dialog</td>
<td>2. AFTER FIELD A \</td>
</tr>
<tr>
<td></td>
<td>3. AFTER INPUT or AFTER CONSTRUCT (for sub-dialog of field A) \</td>
</tr>
<tr>
<td></td>
<td>4. BEFORE INPUT or BEFORE CONSTRUCT (for sub-dialog of field B) \</td>
</tr>
<tr>
<td></td>
<td>5. BEFORE FIELD B \</td>
</tr>
<tr>
<td>Moving to a different row in a DISPLAY ARRAY list</td>
<td>1. AFTER ROW (the row you leave) \</td>
</tr>
<tr>
<td></td>
<td>2. BEFORE ROW (the new current row) \</td>
</tr>
<tr>
<td>Context / User action</td>
<td>Control Block execution order</td>
</tr>
<tr>
<td>-------------------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Moving to a different row in an <strong>INPUT ARRAY</strong> list when current row was newly created</td>
<td>1. <strong>ON CHANGE</strong> (if value of current field has changed)</td>
</tr>
<tr>
<td></td>
<td>2. <strong>AFTER FIELD</strong> (for field A in the row you leave)</td>
</tr>
<tr>
<td></td>
<td>3. <strong>AFTER INSERT</strong> (the newly created row)</td>
</tr>
<tr>
<td></td>
<td>4. <strong>AFTER ROW</strong> (the newly created row)</td>
</tr>
<tr>
<td></td>
<td>5. <strong>BEFORE ROW</strong> (the new current row)</td>
</tr>
<tr>
<td></td>
<td>6. <strong>BEFORE FIELD</strong> (field in the new current row)</td>
</tr>
<tr>
<td>Moving to a different row in an <strong>INPUT ARRAY</strong> list when current row was modified</td>
<td>1. <strong>ON CHANGE</strong> (if value of current field has changed)</td>
</tr>
<tr>
<td></td>
<td>2. <strong>AFTER FIELD</strong> (for field A in the row you leave)</td>
</tr>
<tr>
<td></td>
<td>3. <strong>ON ROW CHANGE</strong> (the values in current row have changed)</td>
</tr>
<tr>
<td></td>
<td>4. <strong>AFTER ROW</strong> (for the row that was modified)</td>
</tr>
<tr>
<td></td>
<td>5. <strong>BEFORE ROW</strong> (the new current row)</td>
</tr>
<tr>
<td></td>
<td>6. <strong>BEFORE FIELD</strong> (field in the new current row)</td>
</tr>
<tr>
<td>Inserting or appending a new row in an <strong>INPUT ARRAY</strong> list</td>
<td>1. <em>Triggers raised by the context of the sub-dialog you leave</em></td>
</tr>
<tr>
<td></td>
<td>2. <strong>BEFORE INSERT</strong> (for the new current row)</td>
</tr>
<tr>
<td></td>
<td>3. <strong>BEFORE ROW</strong> (the new current row)</td>
</tr>
<tr>
<td></td>
<td>4. <strong>BEFORE FIELD</strong> (field in the new current row)</td>
</tr>
<tr>
<td>Deleting a row in an <strong>INPUT ARRAY</strong> list</td>
<td>1. <strong>BEFORE DELETE</strong> (for the current row to be deleted)</td>
</tr>
<tr>
<td></td>
<td>2. <strong>AFTER DELETE</strong> (now the deleted row is removed)</td>
</tr>
<tr>
<td></td>
<td>3. <strong>AFTER ROW</strong> (for the deleted row)</td>
</tr>
<tr>
<td></td>
<td>4. <strong>BEFORE ROW</strong> (the new current row)</td>
</tr>
<tr>
<td>Firing the <em>insert or append</em> action for the <strong>ON INSERT</strong> block in a <strong>DISPLAY ARRAY</strong> list</td>
<td>1. <strong>AFTER ROW</strong></td>
</tr>
<tr>
<td></td>
<td>2. <strong>ON INSERT</strong></td>
</tr>
<tr>
<td></td>
<td>3. <strong>BEFORE ROW</strong></td>
</tr>
<tr>
<td>Firing the <em>delete</em> action for the <strong>ON DELETE</strong> block in a <strong>DISPLAY ARRAY</strong> list</td>
<td>1. <strong>AFTER ROW</strong></td>
</tr>
<tr>
<td></td>
<td>2. <strong>ON DELETE</strong></td>
</tr>
<tr>
<td></td>
<td>3. <strong>BEFORE ROW</strong></td>
</tr>
<tr>
<td>Validating the dialog with <strong>ACCEPT DIALOG</strong></td>
<td>1. <strong>ON CHANGE</strong> (if focus is in input field and value has changed)</td>
</tr>
<tr>
<td></td>
<td>2. <strong>AFTER FIELD</strong> (if focus is in input field)</td>
</tr>
<tr>
<td></td>
<td>3. <strong>AFTER INSERT</strong> (if <strong>INPUT ARRAY</strong> and current row was just created)</td>
</tr>
<tr>
<td></td>
<td>or</td>
</tr>
<tr>
<td></td>
<td><strong>ON ROW CHANGE</strong> (if <strong>INPUT ARRAY</strong> and a value in the row has changed)</td>
</tr>
<tr>
<td></td>
<td>4. <strong>AFTER ROW</strong> (if focus is in a list)</td>
</tr>
<tr>
<td></td>
<td>5. <strong>AFTER INPUT, AFTER CONSTRUCT</strong> or <strong>AFTER CONSTRUCT</strong> (current sub-dialog)</td>
</tr>
<tr>
<td></td>
<td>6. <strong>AFTER DIALOG</strong></td>
</tr>
<tr>
<td>Canceling the dialog with <strong>EXIT DIALOG</strong></td>
<td>None of the control blocks will be executed; we just leave the dialog instruction.</td>
</tr>
</tbody>
</table>
**BEFORE DIALOG block**

The **BEFORE DIALOG** block is executed one time as the first trigger when the **DIALOG** instruction starts, before the runtime system gives control to the user. You can implement variable initialization and dialog configuration in this block.

**Note:** Like all control blocks of the procedural **DIALOG / END DIALOG** instruction, the **BEFORE DIALOG** block must appear after the sub-dialog definitions. For a detailed description, see the **procedural DIALOG instruction syntax**.

In the next code example, the **BEFORE DIALOG** block performs some dialog setup and gives the focus to a specific field:

```plaintext
DIALOG
  DISPLAY ARRAY ...
  ...
  END DISPLAY
  INPUT BY NAME ...
  ...
  END INPUT
  BEFORE DIALOG
    CALL DIALOG.setActionActive("save",FALSE)
    CALL DIALOG.setFieldActive("cust_status", is_admin())
    IF cust_is_new() THEN
      NEXT FIELD cust_name
    END IF
  ...
END DIALOG
```

A **DIALOG** instruction can include no more than one **BEFORE DIALOG** control block.

**Related concepts**

- The **Dialog class** on page 2512
- The **ui.Dialog class** provides a set of methods to configure, query and control the current interactive instruction.

**AFTER DIALOG block**

The **AFTER DIALOG** block is executed one time as the last trigger when the **DIALOG** instruction terminates, when performing an **ACCEPT DIALOG** instruction. Dialog finalization code can be implemented in this block.

The dialog terminates when an **ACCEPT DIALOG** or **EXIT DIALOG** control instruction is executed. However, the **AFTER DIALOG** block is **not** executed if an **EXIT DIALOG** is performed.

If you execute one of the following control instructions in an **AFTER DIALOG** block, the dialog will not terminate and it will give control back to the user:

1. **NEXT FIELD**
2. **NEXT OPTION**
3. **CONTINUE DIALOG**

In the next example, the **AFTER DIALOG** block checks whether a field value is correct and gives control back to the dialog if the value is wrong:

```plaintext
ON ACTION accept
  ACCEPT DIALOG
...
AFTER DIALOG
  IF NOT cust_is_valid_status(p_cust.cust_status) THEN
    ERROR "Customer state is not valid"
    NEXT FIELD cust_status
```
Related concepts

ACCEPT DIALOG instruction on page 1640
EXIT DIALOG instruction on page 1640
BEFORE DIALOG block on page 1610

BEFORE FIELD block

In dialog instructions INPUT, INPUT ARRAY, CONSTRUCT or in a DISPLAY ARRAY using the FOCUSONFIELD attribute, the BEFORE FIELD block is executed every time the specified field gets the focus.

For single record inputs driven by INPUT or query by example (QBEs) driven by CONSTRUCT, the BEFORE FIELD block is executed when moving the focus from field to field.

For editable lists driven by INPUT ARRAY, the BEFORE FIELD block is executed when moving the focus from field to field in the same row, or when moving to another row in the same column.

For record lists driven by DISPLAY ARRAY using the FOCUSONFIELD attribute, the BEFORE FIELD block is executed when moving the focus from field to field. However, the fields will not be editable as in an INPUT ARRAY.

Important: The BEFORE FIELD block is also executed when performing a NEXT FIELD instruction.

The BEFORE FIELD keywords must be followed by a list of form field specification. The screen-record name can be omitted.

BEFORE FIELD is executed after BEFORE INPUT, BEFORE CONSTRUCT, BEFORE ROW and BEFORE INSERT.

Use this block to do some field value initialization, or to display a message to the user:

```plaintext
INPUT BY NAME p_cust.* ...
BEFORE FIELD cust_status
  LET p_cust.cust_comment = NULL
  MESSAGE "Enter customer status"
```

When using the default FIELD ORDER CONSTRAINT mode, the dialog executes the BEFORE FIELD block of the field corresponding to the first variable of an INPUT or INPUT ARRAY, even if that field is not editable (NOENTRY, hidden or disabled). The block is executed when you enter the dialog and every time you create a new row in the case of INPUT ARRAY. This behavior is supported for backward compatibility. The block is not executed when using the FIELD ORDER FORM, the mode recommended for DIALOG instructions.

With the FIELD ORDER FORM mode, for each dialog executing for the first time with a specific form, the BEFORE FIELD block will be invoked for the first field of the initial tabbing list defined by the form, even if that field was hidden or moved around in a table. The dialog then behaves as if a NEXT FIELD first-visible-column had been executed in the BEFORE FIELD of that field.

When form-level validation occurs and a field contains an invalid value, the dialog gives the focus to the field, but no BEFORE FIELD trigger will be executed.

Related concepts

Form-level validation rules on page 1727
Form-level validation rules can be defined for each field controlled by a dialog.

AFTER FIELD block on page 1492

AFTER FIELD block

In dialog instructions INPUT, INPUT ARRAY, CONSTRUCT or in a DISPLAY ARRAY using the FOCUSONFIELD attribute, the AFTER FIELD block is executed every time the focus leaves the specified field.

For single record inputs driven by INPUT or query by example (QBEs) driven by CONSTRUCT, the AFTER FIELD block is executed when moving the focus from field to field.
For editable lists driven by `INPUT` array, the `AFTER FIELD` block is executed when moving the focus from field to field in the same row, or when moving to another row in the same column.

For record lists driven by `DISPLAY` array using the `FOCUSONFIELD` attribute, the `AFTER FIELD` block is executed when moving the focus from field to field. However, the fields will not be editable as in an `INPUT` array.

The `AFTER FIELD` keywords must be followed by a list of form field specifications. The screen-record name can be omitted.

`AFTER FIELD` is executed before `AFTER INSERT`, `ON ROW CHANGE`, `AFTER ROW`, `AFTER INPUT` or `AFTER CONSTRUCT`.

When a `NEXT FIELD` instruction is executed in an `AFTER FIELD` block, the cursor moves to the specified field, which can be the current field. This can be used to prevent the user from moving to another field/row during data input. Note that the `BEFORE FIELD` block is also executed when `NEXT FIELD` is invoked.

The `AFTER FIELD` block of the current field is not executed when performing a `NEXT FIELD`; only `BEFORE INPUT`, `BEFORE CONSTRUCT`, `BEFORE ROW`, and `BEFORE FIELD` of the target item might be executed, depending on the sub-dialog type.

When `ACCEPT DIALOG`, `ACCEPT INPUT`, or `ACCEPT CONTRACT` is performed, the `AFTER FIELD` trigger of the current field is executed.

Use the `AFTER FIELD` block to implement field validation rules:

```plaintext
INPUT BY NAME p_item.* ...
AFTER FIELD item_quantity
  IF p_item.item_quantity <= 0 THEN
    ERROR "Item quantity cannot be negative or zero"
  END IF
  LET p_item.item_quantity = 0
  NEXT FIELD item_quantity
END IF
```

**Related concepts**

- [ACCEPT DIALOG instruction](#) on page 1640
- [ON CHANGE block](#) on page 1492

**ON CHANGE block**

The `ON CHANGE` block can be used to detect when a field has been changed by user input. The `ON CHANGE` block is executed, if the value has changed since the field got the focus and the modification flag is set.

The `ON CHANGE` block can be used in `INPUT`, `INPUT ARRAY` and `CONSTRUCT` dialogs.

For editable fields defined as `EDIT`, `TEXTEDIT` or `BUTTONEDIT`, the `ON CHANGE` block is executed when leaving a field, if the value of the specified field has changed since the field got the focus and the modification flag is set for the field. The field is left when user validates the dialog, when moving to another field, or when moving to another row in an `INPUT` array. However, if the text edit field is defined with the `COMPLETER` attribute to enable autocompletion, the `ON CHANGE` trigger will be fired after a short period of time, when the user has typed characters in.

For editable fields defined as `CHECKBOX`, `COMBOBOX`, `DATEEDIT`, `DATETIMEEDIT`, `TIMEEDIT`, `RADIOGROUP`, `SPINEDIT`, `SLIDER` or `URL-based` `WEBCOMPONENT` (when the `COMPONENTTYPE` attribute is not used), the `ON CHANGE` block is invoked immediately when the user changes the value with the widget edition feature. For example, when toggling the state of a `CHECKBOX`, when selecting an item in a `COMBOBOX` list, or when choosing a date in the calendar of a `DATEEDIT`. Note that for such item types, when `ON CHANGE` is fired, the modification flag is always set.

```plaintext
ON CHANGE order_checked -- Defined as CHECKBOX
  CALL setup_dialog(DIALOG)
```
**Note:** If both an **ON CHANGE** block and **AFTER FIELD** block are defined for a field, the **ON CHANGE** block is executed before the **AFTER FIELD** block.

When changing by program the value of the current field in an **ON ACTION** block, the **ON CHANGE** block will be executed when leaving the field, if the value is different from the previous value and the modification flag is set (after previous user input or when the touched flag has been changed by program).

In an **INPUT** or **INPUT ARRAY**, the field value change is related to the value of the variable bound to the field. In a **CONSTRUCT** dialog, the field value change is related to the input buffer / displayed value.

**Note:** With a **NEXT FIELD** instruction, the state of the field value change is reset, as if the user had left and reentered the field. When using **NEXT FIELD** in an **ON CHANGE** block or in an **ON ACTION** block, the **ON CHANGE** block will only be re-executed, if the value is changed since **NEXT FIELD**, and the modification flag is set.

Therefore, **ON CHANGE** should not be used for field validation with a **NEXT FIELD**, because on **CHANGE** will not get triggered again, if the (invalid) value has not changed; Field validation rules must be implemented in **AFTER FIELD** blocks and/or **AFTER INPUT** blocks.

**Related concepts**
Form item types on page 1266
The form item types defines the purpose of form elements.
**AFTER FIELD** block on page 1492

**BEFORE INPUT block**

**BEFORE INPUT** block in singular and parallel **INPUT, INPUT ARRAY** dialogs

In a singular **INPUT, INPUT ARRAY** instruction, or when used as parallel dialog, the **BEFORE INPUT** is only executed once when the dialog is started.

The **BEFORE INPUT** block is executed once at dialog startup, before the runtime system gives control to the user. This block can be used to display messages to the user, initialize program variables and set up the dialog instance by deactivating unused fields or actions the user is not allowed to execute.

```plaintext
INPUT BY NAME cust_rec.* ... 
BEFORE INPUT
    MESSAGE "Input customer information"
    CALL DIALOG.setActionActive("check_info", is_super_user() )
    CALL DIALOG.setFieldActive("cust_comment", is_super_user() )
...
```

The fields are initialized with the defaults values before the **BEFORE INPUT** block is executed. When the **INPUT** instruction uses the **WITHOUT DEFAULTS** option, the default values are taken from the program variables bound to the fields, otherwise (with defaults), the **DEFAULT** attributes of the form fields are used.

Use the **NEXT FIELD** control instruction in the **BEFORE INPUT** block, to jump to a specific field when the dialog starts.

**BEFORE INPUT** block in **INPUT and INPUT ARRAY** of procedural **DIALOG**

In an **INPUT or INPUT ARRAY** sub-dialog of a procedural **DIALOG** instruction, the **BEFORE INPUT** block is executed when the focus goes to a group of fields driven by the sub-dialog. This trigger is only invoked if a field of the sub-dialog gets the focus, and none of the other fields had the focus.

When the focus is in a list driven by an **INPUT ARRAY** sub-dialog, moving to a different row will not invoke the **BEFORE INPUT** block.

**BEFORE INPUT** is executed **after** the **BEFORE DIALOG** block and **before** the **BEFORE ROW, BEFORE FIELD** blocks.
In this example, the BEFORE INPUT block is used to set up a specific action and display a message:

```plaintext
INPUT BY NAME p_order.*
BEFORE INPUT
    CALL DIALOG.setActionActive("validate_order", TRUE)
END BEFORE INPUT
```

**Related concepts**
- BEFORE CONSTRUCT block on page 1573
- BEFORE DISPLAY block on page 1509
- AFTER INPUT block on page 1490

### AFTER INPUT block

#### AFTER INPUT block in singular and parallel INPUT, INPUT ARRAY dialogs

In a singular INPUT, INPUT ARRAY instruction, or when used as parallel dialog, the AFTER INPUT is only executed once when dialog ends.

The AFTER INPUT block is executed after the user has validated or canceled the INPUT or INPUT ARRAY dialog with the accept or cancel default actions, or when the ACCEPT INPUT instruction is executed.

The AFTER INPUT block is not executed when the EXIT INPUT instruction is performed.

In singular and parallel dialogs, this block is typically used to implement global dialog validation rules for several fields. If the values entered by the user do not satisfy the constraints, use the NEXT FIELD instruction to force the dialog to continue. The CONTINUE INPUT instruction can be used instead of NEXT FIELD, when no particular field has to be selected.

Before checking the validation rules, make sure that the INT_FLAG variable is FALSE: because if the user cancels the dialog, the validation rules must be skipped.

```plaintext
INPUT BY NAME cust_rec.*
    WITHOUT DEFAULTS ATTRIBUTES ( UNBUFFERED )
...
AFTER INPUT
    IF NOT INT_FLAG THEN
        IF cust_rec.cust_address IS NOT NULL
            AND cust_rec.cust_zipcode IS NULL THEN
                ERROR "Address is incomplete, enter a zipcode."
                NEXT FIELD zipcode
            END IF
        END IF
    END IF
END INPUT
```

To limit the validation to fields that have been modified by the end user, you can call the FIELD_TOUCHED() function or the DIALOG.getFieldTouched() method to check if a field has changed during the dialog execution. This will make your validation code faster if the user has only modified a couple of fields in a large form.

#### AFTER INPUT block in INPUT and INPUT ARRAY of procedural DIALOG

In an INPUT or INPUT ARRAY sub-dialog of a procedural DIALOG instruction, the AFTER INPUT block is executed when the focus is lost by a group of fields driven by an INPUT or INPUT ARRAY sub-dialog. This trigger is invoked if a field of the sub-dialog loses the focus, and a field of a different sub-dialog gets the focus. When the focus is in a list driven by an INPUT ARRAY sub-dialog, moving to a different row will not invoke the AFTER INPUT block.

If the focus leaves the current group and goes to an action view, this trigger is not executed, because the focus has not gone to another sub-dialog yet.

AFTER INPUT is executed after the AFTER FIELD, AFTER ROW blocks and before the AFTER DIALOG block.
Executing a `NEXT FIELD` in the `AFTER INPUT` control block will keep the focus in the group of fields. Within an `INPUT ARRAY` sub-dialog, `NEXT FIELD` will keep the focus in the list and stay in the current row. You typically use this behavior to control user input.

In this example, the `AFTER INPUT` block is used to validate data and disable an action that can only be used in the current group:

```
INPUT BY NAME p_order.*
  AFTER INPUT
    IF NOT check_order_data(DIALOG) THEN
      NEXT FIELD CURRENT
    END IF
    CALL DIALOG.setFieldActive("validate_order", FALSE)
```

**Related concepts**
- `AFTER DISPLAY` block on page 1510
- `AFTER CONSTRUCT` block on page 1574
- `BEFORE INPUT` block on page 1489

**BEFORE CONSTRUCT block**

**BEFORE CONSTRUCT block in singular and parallel `CONSTRUCT` dialogs**

In a singular `CONSTRUCT` instruction, or when used as parallel dialog, the `BEFORE CONSTRUCT` is only executed once when dialog is started.

The `BEFORE CONSTRUCT` block is executed once at dialog start-up, before the runtime system gives control to the user for criteria input. This block can be used to display messages to the user, initialize form fields with default search criteria values, and setup the dialog instance by deactivating unused fields or actions the user is not allowed to execute.

```
CONSTRUCT BY NAME where_part ON ...
  BEFORE CONSTRUCT
    MESSAGE "Enter customer search filter"
    CALL DIALOG.setActionActive("clean", FALSE )
  ...
```

The fields are cleared before the `BEFORE CONSTRUCT` block is executed.

You can use the `NEXT FIELD` control instruction in the `BEFORE CONSTRUCT` block, to jump to a specific field when the dialog starts.

**BEFORE CONSTRUCT block in `CONSTRUCT` of procedural `DIALOG`**

In a `CONSTRUCT` sub-dialog of a procedural `DIALOG` instruction, the `BEFORE CONSTRUCT` block is executed when the focus goes to a group of fields driven by a `CONSTRUCT` sub-dialog. This trigger is only invoked if a field of the sub-dialog gets the focus, and none of the other fields had the focus.

`BEFORE CONSTRUCT` is executed **after** the `BEFORE DIALOG` block and **before** the `BEFORE FIELD` blocks.

In this example, the `BEFORE CONSTRUCT` block is used to display a message:

```
CONSTRUCT BY NAME sql ON customer.*
  BEFORE CONSTRUCT
    MESSAGE "Enter customer search filter"
```

**Related concepts**
- `BEFORE INPUT` block on page 1489
- `BEFORE DISPLAY` block on page 1509
- `AFTER CONSTRUCT` block on page 1574
**AFTER CONSTRUCT block**

**AFTER CONSTRUCT block in singular and parallel CONSTRUCT dialogs**

In a singular CONSTRUCT instruction, or when used as parallel dialog, the AFTER CONSTRUCT is only executed once when dialog is ended.

Use an AFTER CONSTRUCT block to execute instructions after the user has finished search criteria input.

AFTER CONSTRUCT is not executed if an EXIT CONSTRUCT is performed.

The code in AFTER CONSTRUCT can for example check if a criteria combination of different fields is required or denied, and force the end use to enter all.

Before checking the content of the fields used in the CONSTRUCT, make sure that the INT_FLAG variable is FALSE.

In the case that the user cancels the dialog, the validation rules must be skipped.

Since no program variables are associated with the form fields, you must query the input buffers of the fields to get the values entered by the user.

```plaintext
CONSTRUCT BY NAME where_part ON ...

... 
AFTER CONSTRUCT
  IF NOT INT_FLAG THEN
    IF length(DIALOG.getFieldBuffer(cust_name))==0 
    OR length(DIALOG.getFieldBuffer(cust_addr))==0 THEN
      ERROR "Enter a search criteria for customer name and address fields." 
    NEXT FIELD CURRENT
  END IF
END IF
END CONSTRUCT
```

To limit the validation to fields that have been modified by the end user, you can call the FIELD_TOUCHED() function or the DIALOG.getFieldTouched() method to check if a field has changed during the dialog execution. This makes your validation code execute faster if the user has only modified a couple of fields in a large form.

**AFTER CONSTRUCT block in CONSTRUCT of procedural DIALOG**

In a CONSTRUCT sub-dialog of a procedural DIALOG instruction, the AFTER CONSTRUCT block is executed when the focus is lost by a group of fields driven by a CONSTRUCT sub-dialog. This trigger is invoked if a field of the sub-dialog loses the focus, and a field of a different sub-dialog gets the focus.

If the focus leaves the current group and goes to an action view, this trigger is not executed, because the focus did not go to another sub-dialog yet.

AFTER CONSTRUCT is executed after the AFTER FIELD and before the AFTER DIALOG block.

Executing a NEXT FIELD in the AFTER CONSTRUCT control block will keep the focus in the group of fields.

In this example, the AFTER CONSTRUCT block is used to build the SELECT statement:

```plaintext
CONSTRUCT BY NAME sql ON customer.*
AFTER CONSTRUCT
  LET sql = "SELECT * FROM customers WHERE " || sql
```

**Related concepts**

AFTER DISPLAY block on page 1510
AFTER INPUT block on page 1490
BEFORE CONSTRUCT block on page 1573
**BEFORE DISPLAY block**

**BEFORE DISPLAY block in singular and parallel DISPLAY ARRAY dialogs**

In a singular DISPLAY ARRAY instruction, or when used as parallel dialog, the BEFORE DISPLAY is only executed once when the dialog is started.

The BEFORE DISPLAY block is executed once at dialog startup, before the runtime system gives control to the user. This block can be used to display messages to the user, initialize program variables, and set up the dialog instance by deactivating actions the user is not allowed to execute.

```plaintext
DISPLAY ARRAY p_items TO s_items.*
   BEFORE DISPLAY
       CALL DIALOG.setActive("clear_item_list", is_super_user())
```

**BEFORE DISPLAY block DISPLAY ARRAY of procedural DIALOG**

In a DISPLAY ARRAY sub-dialog of a procedural DIALOG instruction, the BEFORE DISPLAY block is executed when a DISPLAY ARRAY list gets the focus.

BEFORE DISPLAY is executed before the BEFORE ROW block.

In this example the BEFORE DISPLAY block enables an action and displays a message:

```plaintext
DISPLAY ARRAY p_items TO s_items.*
   BEFORE DISPLAY
       CALL DIALOG.setActive("print_list", TRUE)
       MESSAGE "You are now in the list of items"
```

**Related concepts**

BEFORE INPUT block on page 1489
BEFORE CONSTRUCT block on page 1573
AFTER DISPLAY block on page 1510

**AFTER DISPLAY block**

**AFTER DISPLAY block in singular and parallel DISPLAY ARRAY dialogs**

In a singular DISPLAY ARRAY instruction, or when used as parallel dialog, the AFTER DISPLAY is only executed once when the dialog is ended.

You typically implement dialog finalization in this block.

```plaintext
DISPLAY ARRAY p_items TO s_items.*
   AFTER DISPLAY
       DISPLAY "Current row is: ", arr_curr()
```

**AFTER DISPLAY block in DISPLAY ARRAY of procedural DIALOG**

In a DISPLAY ARRAY sub-dialog of a procedural DIALOG instruction, the AFTER DISPLAY block is executed when a DISPLAY ARRAY list loses the focus and the focus goes to another sub-dialog.

If the focus leaves the current group and goes to an action view, this trigger is not executed, because the focus has not gone to another sub-dialog yet.

AFTER DISPLAY is executed after the AFTER ROW block.

In this example, the AFTER DISPLAY block disables an action that is specific to the current list:

```plaintext
DISPLAY ARRAY p_items TO s_items.*
   AFTER DISPLAY
```
CALL DIALOG.setActionActive("clear_item_list", FALSE)

**Related concepts**

- **AFTER INPUT block** on page 1490
- **AFTER CONSTRUCT block** on page 1574
- **BEFORE DISPLAY block** on page 1509

**BEFORE ROW block**

**BEFORE ROW block in singular and parallel DISPLAY ARRAY, INPUT ARRAY dialogs**

In a singular DISPLAY ARRAY, INPUT ARRAY instruction, or when used as parallel dialog, the BEFORE ROW block is executed each time the user moves to another row. This trigger can also be executed in other situations, such as when you delete a row, or when the user tries to insert a row but the maximum number of rows in the list is reached.

You typically do some dialog setup / message display in the BEFORE ROW block, because it indicates that the user selected a new row or entered in the list.

When the dialog starts, BEFORE ROW will be executed for the current row, but only if there are data rows in the array.

When called in this block, `DIALOG.getCurrentRow() / arr_curr()` return the index of the current row.

In this example, the BEFORE ROW block gets the new row number and displays it in a message:

```plaintext
DISPLAY ARRAY ...
... 
BEFORE ROW
   MESSAGE "We are on row # ", arr_curr()
...
```

**BEFORE ROW block in DISPLAY ARRAY and INPUT ARRAY of procedural DIALOG**

In an INPUT or INPUT ARRAY sub-dialog of a procedural DIALOG instruction, the BEFORE ROW block is executed when a DISPLAY ARRAY or INPUT ARRAY list gets the focus, or when the user moves to another row inside a list. This trigger can also be executed in other situations, for example when you delete a row, or when the user tries to insert a row but the maximum number of rows in the list is reached.

You typically do some dialog setup / message display in the BEFORE ROW block, because it indicates that the user selected a new row. Do not use this trigger to detect focus changes; Use the BEFORE DISPLAY or BEFORE INPUT blocks instead.

In DISPLAY ARRAY, BEFORE ROW is executed after the BEFORE DISPLAY block. In INPUT ARRAY, BEFORE ROW is executed before the BEFORE INSERT and BEFORE FIELD blocks and after the BEFORE INPUT blocks.

When the procedural dialog starts, BEFORE ROW will only be executed if the list has received the focus and there is a current row (the array is not empty). If you have other elements in the form which can get the focus before the list, BEFORE ROW will not be triggered when the dialog starts. You must pay attention to this, because this behavior is different to the behavior of singular DISPLAY ARRAY or INPUT ARRAY. In singular dialogs, the BEFORE ROW block is always executed when the dialog starts (and when there are rows in the array).

When called in this block, `DIALOG.getCurrentRow() / arr_curr()` return the index of the current row.

In this example the BEFORE ROW block displays a message with the current row number:

```plaintext
DISPLAY ARRAY p_items TO s_items.*
BEFORE ROW
   MESSAGE "We are in items, on row ", DIALOG.getCurrentRow("s_items")
```
**Related concepts**
- BEFORE INPUT block on page 1489
- BEFORE DISPLAY block on page 1509

**ON ROW CHANGE block**

The **ON ROW CHANGE** block is executed in a list controlled by an **INPUT ARRAY**, when leaving the current row and when the row has been modified since it got the focus. This is typically used to detect row modification.

The code in **ON ROW CHANGE** will not be executed when leaving new rows created by the user with the default append or insert action. To detect row creation, you must use the BEFORE INSERT or AFTER INSERT control blocks.

The **ON ROW CHANGE** block is only executed if at least one field value in the current row has changed since the row was entered, and the modification flag of the field is set. The modified field(s) may not be the current field, and several field values can be changed. Values may have been changed by the user or by the program. The modification flag is reset for all fields when entering another row, when going to another sub-dialog, or when leaving the dialog instruction.

**ON ROW CHANGE** is executed after the **AFTER FIELD** block and before the **AFTER ROW** block.

When called in this block, `DIALOG.getCurrentRow()` / `arr_curr()` return the index of the current row that has been changed.

You can, for example, code database modifications (UPDATE) in the **ON ROW CHANGE** block:

```plaintext
INPUT ARRAY p_items FROM s_items.*

... 
ON ROW CHANGE
   LET r = DIALOG.getCurrentRow("s_items")
   UPDATE items SET
      items.item_code        = p_items[r].item_code,
      items.item_description = p_items[r].item_description,
      items.item_price       = p_items[r].item_price,
      items.item_updatedate  = TODAY
   WHERE items.item_num = p_items[r].item_num
```

**Related concepts**
- Input field modification flag on page 1723
  Each input field controlled by a dialog instruction has a modification flag.
- AFTER ROW block on page 1511

**AFTER ROW block**

**AFTER ROW block in singular and parallel DISPLAY ARRAY, INPUT ARRAY dialogs**

In a singular **DISPLAY ARRAY**, **INPUT ARRAY** instruction, or when used as parallel dialog, the **AFTER ROW** block is executed each time the user moves to another row, before the current row is left. This trigger can also be executed in other situations, such as when you delete a row, or when the user inserts a new row.

A **NEXT FIELD** instruction executed in the **AFTER ROW** control block will keep the user entry in the current row. Use this behavior to implement row validation and prevent the user from leaving the list or moving to another row.

When called in this block, `DIALOG.getCurrentRow()` / `arr_curr()` returns the index of the row that you are leaving.

**AFTER ROW block in DISPLAY ARRAY and INPUT ARRAY of procedural DIALOG**

In an **INPUT** or **INPUT ARRAY** sub-dialog of a procedural **DIALOG** instruction, the **AFTER ROW** block is executed when a **DISPLAY ARRAY** or **INPUT ARRAY** list loses the focus, or when the user moves to another row in a list.
This trigger can also be executed in other situations, for example when you delete a row, or when the user inserts a new row.

**AFTER ROW** is executed after the **AFTER FIELD**, **AFTER INSERT** and before **AFTER DISPLAY** or **AFTER INPUT** blocks.

When called in this block, `DIALOG.getCurrentRow() / arr_curr()` returns the index of the row that you are leaving.

For both **INPUT ARRAY** and **DISPLAY ARRAY** sub-dialogs, a **NEXT FIELD** executed in the **AFTER ROW** control block will keep the focus in the list and stay in the current row. Use this feature to implement row validation and prevent the user from leaving the list or moving to another row.

**AFTER ROW and temporary rows in INPUT ARRAY**

**Important:** After creating a temporary row at the end of a list driven by INPUT ARRAY, if you leave that row to go to a previous row without data input (setting the touched flag), or when the cancel action is invoked, the temporary row will be automatically removed. The **AFTER ROW** block will be executed for the temporary row, but `DIALOG.getCurrentRow() / arr_curr()` will be one row greater than `DIALOG.getArrayLength() / ARR_COUNT()`. In this case, it is recommended that you ignore the **AFTER ROW** event. For example, it is recommended that you avoid executing a **NEXT FIELD** or **CONTINUE INPUT** instruction, and trying to access the dynamic array with a row index that is greater than the total number of rows, otherwise the runtime system will adapt the total number of rows to the actual number of rows in the program array.

In this example, the **AFTER ROW** block checks the current row index and verifies a variable value to force the focus to stay in the current row if the value is wrong:

```
INPUT ARRAY p_items FROM s_items.*
...
AFTER ROW
  LET r = DIALOG.getCurrentRow("s_items")
  IF r <= DIALOG.getArrayLength("s_items") THEN
    IF NOT item_is_valid_quantity(p_item[r].item_quantity) THEN
      ERROR "Item quantity is not valid"
      NEXT FIELD item_quantity
    END IF
  END IF
END IF
```

Another way to handle the case of temporary rows in **AFTER ROW** is to use a flag to know if the **AFTER INSERT** block was executed. The **AFTER INSERT** block is not executed if the temporary row is automatically removed. By setting a first value in **BEFORE INSERT** and changing the flag in **AFTER INSERT**, you can detect if the row was permanently added to the list:

```
INPUT ARRAY p_items FROM s_items.*
...
BEFORE INSERT
  LET op = "T"
...
AFTER INSERT
  LET op = "I"
...
AFTER ROW
  IF op == "I" THEN
    IF NOT item_is_valid_quantity(p_item[arr_curr()].item_quantity) THEN
      ERROR "Item quantity is not valid"
      NEXT FIELD item_quantity
    END IF
  END IF
WHENEVER ERROR CONTINUE
  INSERT INTO items (item_num, item_name, item_quantity)
    VALUES ( p_item[arr_curr()].* )
WHENEVER ERROR STOP
```
IF SQLCA.SQLCODE<0 THEN
    ERROR "Could not insert the record into database!"
    NEXT FIELD CURRENT
ELSE
    MESSAGE "Record has been inserted successfully"
END IF
END IF

Related concepts
NEXT FIELD instruction on page 1497
BEFORE ROW block on page 1510
ON ROW CHANGE block on page 1544

BEFORE INSERT block

The BEFORE INSERT block is executed when a new row is created in an INPUT ARRAY. You typically use this trigger to set some default values in the newly-created row. A new row can be created by moving down after the last row, by executing an insert action, or by executing an append action.

The BEFORE INSERT block is executed after the BEFORE ROW block and before the BEFORE FIELD block.

When called in this block, DIALOG.getCurrentRow() / arr_curr() returns the index of the newly-created row.

To distinguish row insertion from an appended row, compare the current row (DIALOG.getCurrentRow("screen-array")) with the total number of rows (DIALOG.getArrayLength("screen-array")). If the current row index and the total number of rows correspond, the BEFORE INSERT concerns a temporary row, otherwise it concerns an inserted row.

Row creation can be stopped by using the CANCEL INSERT instruction inside BEFORE INSERT. If possible, it is however better to disable the insert and append actions to prevent the user executing the actions with DIALOG.setActionActive().

In this example, the BEFORE INSERT block checks if the user can create rows and prevents new row creation if needed; otherwise, it sets some default values:

```
INPUT ARRAY p_items FROM s_items.*
...
BEFORE INSERT
    IF NOT user_can_append THEN
        ERROR "You are not allowed to append rows"
        CANCEL INSERT
    END IF
    LET r = DIALOG.getCurrentRow("s_items")
    LET p_items[r].item_num = get_new_serial("items")
    LET p_items[r].item_name = "undefined"
```

Related concepts
Appending rows in INPUT ARRAY on page 1848
Rows appended at the end of an editable list are temporary until they are edited.

BEFORE ROW block on page 1510

AFTER INSERT block

The AFTER INSERT block of INPUT ARRAY is executed when the creation of a new row is validated. In this block, you can for example implement SQL to insert a new row in the database table.

The AFTER INSERT block is executed after the AFTER FIELD block and before the AFTER ROW block.

When called in this block, DIALOG.getCurrentRow() / arr_curr() returns the index of the newly-created row.
When the user appends a new row at the end of the list, then moves UP to another row or validates the dialog, the 
AFTER INSERT block is only executed if at least one field was edited. If no data entry is detected, the dialog 
automatically removes the new appended row and thus does not trigger the AFTER INSERT block.

When executing a NEXT FIELD in the AFTER INSERT block, the dialog will keep the focus in the list and stay 
in the current row. Use this behavior to implement row input validation and prevent the user from leaving the list or 
moving to another row. However, this will not cancel the row insertion and will not invoke the BEFORE INSERT 
/ AFTER INSERT triggers again. The only way to keep the focus in the current row after the row was inserted is to 
execute a NEXT FIELD in the AFTER ROW block.

In this example, the AFTER INSERT block inserts a new row in the database and cancels the operation if the SQL 
command fails:

```sql
INPUT ARRAY p_items FROM s_items.*
...
AFTER INSERT
  WHENEVER ERROR CONTINUE
  INSERT INTO items VALUES
    ( p_items[ DIALOG.getCurrentRow("s_items") ].* )
  WHENEVER ERROR STOP
  IF SQLCA.SQLCODE<>0 THEN
    ERROR SQLERRMESSAGE
    CANCEL INSERT
  END IF
```

Related concepts
NEXT FIELD instruction on page 1497
AFTER ROW block on page 1511

BEFORE DELETE block

The BEFORE DELETE block is executed each time the user deletes a row of an INPUT ARRAY list, before the row 
is removed from the list.

You typically code the database table synchronization in the BEFORE DELETE block, by executing a DELETE SQL 
statement using the primary key of the current row. In the BEFORE DELETE block, the row to be deleted still exists 
in the program array, so you can access its data to identify what record needs to be removed.

The BEFORE DELETE block is executed before the AFTER DELETE block.

If needed, the deletion can be canceled with the CANCEL DELETE instruction.

When called in this block, DIALOG.getCurrentRow() / arr_curr() returns the index of the row that will be 
deleted.

This example uses the BEFORE DELETE block to remove the row from the database table and cancels the deletion 
operation if an SQL error occurs:

```sql
INPUT ARRAY p_items FROM s_items.*
BEFORE DELETE
  LET r = DIALOG.getCurrentRow("s_items")
  WHENEVER ERROR CONTINUE
  DELETE FROM items
    WHERE item_num = p_items[r].item_num
  WHENEVER ERROR STOP
  IF SQLCA.SQLCODE<>0 THEN
    ERROR SQLERRMESSAGE
    CANCEL DELETE
  END IF
...
```
Related concepts

AFTER DELETE block on page 1548

AFTER DELETE block

The AFTER DELETE block is executed each time the user deletes a row of an INPUT ARRAY list, after the row has been deleted from the list.

The AFTER DELETE block is executed after the BEFORE DELETE block and before the AFTER ROW block for the deleted row and the BEFORE ROW block of the new current row.

When an AFTER DELETE block executes, the program array has already been modified; the deleted row no longer exists in the array (except in the special case when deleting the last row). The arr_curr() function or the ui.Dialog.getCurrentRow() method returns the same index as in BEFORE ROW, but it is the index of the new current row. The AFTER ROW block is also executed just after the AFTER DELETE block.

Important: When deleting the last row of the list, AFTER DELETE is executed for the delete row, and DIALOG.getCurrentRow() / arr_curr() will be one greater than DIALOG.getArrayLength() / ARR_COUNT(). Ensure you avoid accessing a dynamic array with a row index that is greater than the total number of rows, otherwise the runtime system will adapt the total number of rows to the actual number of rows in the program array. When using a static array, you must ignore the values in the rows after ARR_COUNT().

Here the AFTER DELETE block is used to re-number the rows with a new item line number (note that DIALOG.getArrayLength() / ARR_COUNT() may return zero):

```
INPUT ARRAY p_items FROM s_items.*
AFTER DELETE
    LET r = DIALOG.getCurrentRow("s_items")
    FOR i=r TO DIALOG.getArrayLength("s_items")
        LET p_items[i].item_lineno = i
    END FOR
...
```

It is not possible to use the CANCEL DELETE instruction in an AFTER DELETE block. At this time it is too late to cancel row deletion, as the data row no longer exists in the program array.

Related concepts

BEFORE INSERT block on page 1546
AFTER ROW block on page 1511

DIALOG interaction blocks

Dialog interaction blocks are dialog triggers that can be used to execute specific code when the user executes an action in the dialog. For example, when pressing a button in the form, the corresponding ON ACTION interaction block will be executed.

Interaction blocks also include special handlers such as timeout event handler, drag & drop handlers, and modification triggers for DISPLAY ARRAY sub-dialogs.

ON ACTION block

The ON ACTION action-name blocks execute a sequence of instructions when the user triggers a specific action. A typical action handler block looks like this:

```
ON ACTION action-name
    instruction
...
```

Action blocks are bound by name to action views (like buttons) in the current form. Action views can be BUTTON, TOOLBAR buttons, or TOPMENU options, and if no explicit action view is defined, actions are rendered with a default action view, depending on the type of front-end.
This example defines an action block to open a typical zoom window and let the user select a customer record:

```
ON ACTION zoom
    CALL zoom_customers() RETURNING st, rec.cust_id, rec.cust_name
```

In a dialog handling user input such as `INPUT`, `INPUT ARRAY` and `CONSTRUCT`, if an action is specific to a field, add the `INFIELD` clause to have the action automatically enabled when the corresponding field gets the focus:

```
ON ACTION zoom INFIELD cust_city
    CALL zoom_cities() RETURN st, rec.cust_city
```

In most cases actions are decorated with action defaults in form files, but there can be cases where the `ON ACTION` handler needs to define its own attributes at the program level. This can be done by adding the `ATTRIBUTES()` clause of `ON ACTION`:

```
ON ACTION custinfo ATTRIBUTES(DISCLOSUREINDICATOR, IMAGE="info")
    CALL show_customer_info()
```

For more details about action handlers, and action configuration, see `Dialog actions` on page 1739.

**Related concepts**

- Configuring actions on page 1744
- Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with `action attributes`.
- Action handling basics on page 1739
- This topic describes the basics of action views, action events, and action handlers.

**ON IDLE block**

The `ON IDLE` `seconds` clause defines a set of instructions that must be executed after a given period of user inactivity. This interaction block can be used, for example, to quit the dialog after the user has not interacted with the program for a specified period of time.

Do not mix `ON TIMER` and `ON IDLE` clauses.

As `ON IDLE` can fire field input validation, it is therefore not recommended in dialogs allowing input.

The parameter of `ON IDLE` must be an integer literal or variable. If the value is zero, the dialog timeout is disabled.

It is not recommended to use the `ON IDLE` trigger with a short timeout period such as 1 or 2 seconds; The purpose of this trigger is to give the control back to the program after a relatively long period of inactivity (10, 30 or 60 seconds). This is typically the case when the end user leaves the workstation, or gets a phone call. The program can then execute some code before the user gets the control back.

```
ON IDLE 30
    IF ask_question("Do you want to reload information from the database?") THEN
        -- Fetch data back from the db server
    END IF
```

**Important:** The timeout value is taken into account when the dialog initializes its internal data structures. If you use a program variable instead of an integer constant, any change of the variable will have no effect if the variable is changed after the dialog has initialized. If you want to change the value of the timeout variable, it must be done before the dialog block.

**Related concepts**

- Get program control if user is inactive on page 1712
- Execute some code after a given number of seconds, when the user does not interact with the program.

**ON TIMER block** on page 1466
**ON KEY block**

An **ON KEY** (*key-name*) block defines an action with a hidden action view (no default button is visible), that executes a sequence of instructions when the user presses the specified key.

**Note:** The **ON KEY** block is supported for backward compatibility with TUI mode applications. In new developments, consider using **ON ACTION** with accelerators defined in action defaults.

An **ON KEY** block can specify up to four accelerator keys. Each key creates a specific action object that will be identified by the key name in lowercase.

For example, **ON KEY (F5, F6)** creates two actions with the names *f5* and *f6*. Each action object will get an *acceleratorName* attribute assigned, with the corresponding accelerator name. The specified keys must be one of the virtual keys.

**Note:** The **KEY()** clause allows a comma-separated list of keys. Up to four keys can be specified. For new developments, consider using a single key, or prefer **ON ACTION** handlers with a single accelerator definition in action defaults.

In GUI mode, action defaults are applied for **ON KEY** actions by using the name of the action (the key name). You can define secondary accelerator keys, as well as default decoration attributes like button text and image, by using the key name as action identifier. The action name is always in lowercase letters.

Check carefully **ON KEY CONTROL-?** statements to avoid having duplicate accelerators for multiple actions due to the accelerators defined by action defaults. Additionally, **ON KEY** statements used with ESC, TAB, UP, DOWN, LEFT, RIGHT, HELP, NEXT, PREVIOUS, INSERT, CONTROL-M, CONTROL-X, CONTROL-V, CONTROL-C and CONTROL-A should be avoided for use in GUI programs, because it's very likely to clash with default accelerators defined in the factory action defaults file provided by default.

By default, **ON KEY** actions are not decorated with a default button in the action frame (the default action view). You can show the default button by configuring a *text* attribute with the action defaults.

```
ON KEY (CONTROL-Z)
    CALL open_zoom()
```

**Related concepts**

**Configuring actions** on page 1744
Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with *action attributes*.

**Default action views** on page 1743
A default action view is created to render an action handler when no explicit action view exists for it.

**ON TIMER block**

The **ON TIMER** *seconds* clause defines a set of instructions that must be executed at regular intervals. This interaction block can be used, for example, to check if a message has arrived in a queue, and needs to be processed.

Do not mix **ON TIMER** and **ON IDLE** clauses.

As **ON TIMER** can fire field input validation, it is therefore not recommended in dialogs allowing input.

The parameter of **ON TIMER** must be an integer literal or variable. If the value is zero, the dialog timeout is disabled.

It is not recommended to use the **ON TIMER** trigger with a short timeout period, such as 1 or 2 seconds. The purpose of this trigger is to give the control back to the program after a reasonable period of time, such as 10, 20 or 60 seconds.

```
ON TIMER 30
    CALL check_for_messages()
```

**Important:** The timer value is taken into account when the dialog initializes its internal data structures. If you use a program variable instead of an integer constant, a change of the variable has no effect if the change takes place after the dialog has initialized. If you want to change the value of the timeout variable, it must be done before the dialog block.
Related concepts

Get program control on a regular (timed) basis on page 1713
Execute some code after a given number of seconds, with or without user interaction with the program.

ON IDLE block on page 1465

COMMAND [KEY] block

Use COMMAND [KEY] blocks as global procedural DIALOG action handler to execute a sequence of instructions when the user clicks on a button or presses a specific key. COMMAND defines the text and comment decoration attributes as well as accelerator keys for a specific action.

COMMAND is especially useful when writing TUI programs; however, it's legal to use such handler when programming new GUI dialogs, to allow focusable action views (BUTTON in form can take the focus if bound to a COMMAND handler).

Declaring a COMMAND block in DIALOG is similar to an ON ACTION block, except that COMMAND defines an implicit text and comment decoration attribute. The name of the action will be the command text converted to lowercase letters. For example, with the following code:

```
COMMAND "Open" "Opens a new file"
```

The name of the action will be "open", and the default decoration text will be "Open" with a capital letter.

Note that if you use an ampersand (&) in the command name, some front-ends consider the letter following & as an Alt-key accelerator, and the letter will be underscored; however, the ampersand forms part of the action name. For example, COMMAND "&Save" will create an action with the name "&save". It is not recommended to use & ampersand characters in action names.

Unlike ON KEY actions, if no explicit action view is defined in the form, the default action view will be visible for a COMMAND handler (i.e. the automatic button will appear for this action on the front-end).

Action defaults will be applied by using the action name. For explicit action views such as a BUTTON in the form layout, the text/comment defined in the corresponding action default entry will overwrite the values used in the COMMAND handler. When no explicit action view is defined in the form, the text/comment defined in the program COMMAND clause take precedence over action defaults, to display the default action view (button on action frame).

Inside DIALOG instruction, COMMAND blocks can only be defined as global dialog actions; Sub-dialog specific COMMAND handlers cannot be defined. When binding a form BUTTON to a COMMAND handler, the button can get the focus and will be managed in the tabbing list, using preferably the FIELD ORDER FORM option.

When using the optional KEY clause, COMMAND defines also an implicit accelerator key. The key name must be specified between parentheses with COMMAND KEY:

```
COMMAND KEY (F5) "Open" "Opens a new file"
```

Note: The KEY() clause allows a comma-separated list of keys. Up to four keys can be specified. For new developments, consider using a single key, or prefer ON ACTION handlers with a single accelerator definition in action defaults.

When using multiple keys in an COMMAND KEY clause, the DIALOG instruction will assign the specified keys as accelerators:

```
COMMAND KEY (F5, CONTROL-P, CONTROL-Z) "Open" "Opens a new file"
```

With the above code example, the action name will be "open" and accelerators will be F5, CONTROL-P and CONTROL-Z.

Note: The keys defined with the KEY() clause will take precedence over accelerators defined with action defaults corresponding to the action name.
The \texttt{COMMAND [KEY]} block specification can also define a help number with the \texttt{HELP} clause, to display the corresponding text of the current help file.

\begin{verbatim}
COMMAND "Open" "Opens a new file" HELP 34
\end{verbatim}

\textbf{Related concepts}
\begin{itemize}
  \item \texttt{Binding action views to action handlers} on page 1762
  \item \texttt{Configuring actions} on page 1744
\end{itemize}
How are action views of the forms bound to action handlers in the program code?

\textbf{ON APPEND block}
Similar to the \texttt{ON INSERT} control block, the \texttt{ON APPEND} trigger can be used to enable row creation during a \texttt{DISPLAY ARRAY} dialog. If this block is defined, the dialog will automatically create the append action. This action can be decorated, enabled and disabled as a regular action.

If the dialog defines an \texttt{ON ACTION append} interaction block and the \texttt{ON APPEND} block is used, the compiler will stop with error \texttt{-8408}.

When the user fires the append action, the dialog first executes the user code of the \texttt{AFTER ROW} block if defined. Then the dialog moves to the end of the list, and creates a new row after the last existing row. After creating the row, the dialog executes the user code of the \texttt{ON APPEND} block.

The dialog handles only row creation actions and navigation, you must program the record input with a regular \texttt{INPUT} statement, to let the end user enter data for the newly-created row. This is typically done with an \texttt{INPUT} binding explicitly array fields to the screen record fields. The new current row in the program array is identified with \texttt{arr_curr()}, and the current screen line in the form is defined by \texttt{SCR_LINE()}:

\begin{verbatim}
DISPLAY ARRAY arr TO sr.*
  ...
  ON APPEND
    INPUT arr[arr_curr()].* FROM sr[scr_line()].* ;
  ...
\end{verbatim}

Pay attention to the semicolon ending the \texttt{INPUT} instruction, which is usually needed here to solve a language grammar conflict when nested dialog instructions are implemented.

After the user code is executed, the dialog gets the control back and processes the new row as follows:
\begin{itemize}
  \item If the \texttt{INT\_FLAG} global variable is \texttt{FALSE} and \texttt{STATUS} is zero, the new row is kept in the program array, and the \texttt{BEFORE ROW} block is executed for the newly-created row.
  \item If the \texttt{INT\_FLAG} global variable is \texttt{TRUE} or \texttt{STATUS} is different from zero, the new row is removed from the program array, and the \texttt{BEFORE ROW} block is executed for the row that existed at the current position, before the new row was created.
\end{itemize}

The \texttt{DISPLAY ARRAY} dialog always resets \texttt{INT\_FLAG} to \texttt{FALSE} and \texttt{STATUS} to zero before executing the user code of the \texttt{ON APPEND} block.

The append action is disabled if the maximum number of rows is reached.

If needed, the \texttt{ON APPEND} handler can be configured with action attributes by added an \texttt{ATTRIBUTES()} clause, as with user-defined action handlers:

\begin{verbatim}
ON APPEND ATTRIBUTES(TEXT=%"custlist.delete", IMAGE="listdel")
\end{verbatim}

\textbf{Related concepts}
\begin{itemize}
  \item \texttt{Record input (INPUT)} on page 1482
\end{itemize}
The **INPUT** instruction provides single record input control in an application form.

**ON UPDATE block** on page 1518  
**ON DELETE block** on page 1519

**ON INSERT block**

Similar to the **ON APPEND** control block, the **ON INSERT** trigger can be used to enable row creation during a **DISPLAY ARRAY** dialog. If this block is defined, the dialog will automatically create the insert action. This action can be decorated, enabled and disabled as a regular action.

If the dialog defines an **ON ACTION** insert interaction block and the **ON INSERT** block is used, the compiler will stop with error **-8408**.

When the user fires the insert action, the dialog first executes the user code of the **AFTER ROW** block if defined. Then the new row is created: The insert action creates a new row before current row in the list. After creating the row, the dialog executes the user code of the **ON INSERT** block.

The dialog handles only row creation actions and navigation, you must program the record input with a regular **INPUT** statement, to let the end user enter data for the newly-created row. This is typically done with an **INPUT** binding explicitly array fields to the screen record fields. The new current row in the program array is identified with **arr_curr()**, and the current screen line in the form is defined by **scr_line()**:

```plaintext
DISPLAY ARRAY arr TO sr.*
    ...
    ON INSERT
        INPUT arr[arr_curr()].* FROM sr[scr_line()].* ;
    ...
```

Pay attention to the semicolon ending the **INPUT** instruction, which is usually needed here to solve a language grammar conflict when nested dialog instructions are implemented.

After the user code is executed, the dialog gets the control back and processes the new row as follows:

- If the **INT_FLAG** global variable is **FALSE** and **STATUS** is zero, the new row is kept in the program array, and the **BEFORE ROW** block is executed for the new created row.
- If the **INT_FLAG** global variable is **TRUE** or **STATUS** is different from zero, the new row is removed from the program array, and the **BEFORE ROW** block is executed for the row that was existing at the current position, before the new row was created.

The **DISPLAY ARRAY** dialog always resets **INT_FLAG** to **FALSE** and **STATUS** to zero before executing the user code of the **ON INSERT** block.

The insert action is disabled if the maximum number of rows is reached.

If needed, the **ON INSERT** handler can be configured with action attributes by added an **ATTRIBUTES()** clause, as with user-defined action handlers:

```plaintext
ON INSERT ATTRIBUTES(TEXT="custlist.delete", IMAGE="listdel")
```

**Related concepts**

**Record input (INPUT)** on page 1482  
The **INPUT** instruction provides single record input control in an application form.

**ON UPDATE block** on page 1518  
**ON DELETE block** on page 1519

**ON UPDATE block**

The **ON UPDATE** trigger can be used to enable row modification during a **DISPLAY ARRAY** dialog. If this block is defined, the dialog will automatically create the update action. This action can be decorated, enabled and disabled as regular actions.
You typically configure the TABLE container in the form by defining the DOUBLECLICK attribute to "update", in order to trigger the update action when the user double-clicks on a row.

If the dialog defines an ON ACTION update interaction block and the ON UPDATE block is used, the compiler will stop with error -8408.

When the user fires the update action, the dialog executes the user code of the ON UPDATE block.

The dialog handles only the row modification action and navigation, you must program the record input with a regular INPUT statement, to let the end user modify the data of the current row. This is typically done with an INPUT binding explicitly array fields to the screen record fields, with the WITHOUT DEFAULTS clause. The current row in the program array is identified with arr_curr(), and the current screen line in the form is defined by scr_line():

```sql
DISPLAY ARRAY arr TO sr.*
...
ON UPDATE
  INPUT arr[arr_curr()].* WITHOUT DEFAULTS FROM sr[scr_line()].* ;
...
```

Pay attention to the semicolon ending the INPUT instruction, which is usually needed here to solve a language grammar conflict when nested dialog instructions are implemented.

After the user code is executed, the dialog gets the control back and processes the current row as follows:

- If the INT_FLAG global variable is FALSE and STATUS is zero, the modified values of the current row are kept in the program array.
- If the INT_FLAG global variable is TRUE or STATUS is different from zero, the old values of the current row are restored in the program array.

The DISPLAY ARRAY dialog always resets INT_FLAG to FALSE and STATUS to zero before executing the user code of the ON UPDATE block.

If needed, the ON UPDATE handler can be configured with action attributes by added an ATTRIBUTES() clause, as with user-defined action handlers:

```sql
ON UPDATE ATTRIBUTES(TEXT="%custlist.delete", IMAGE="listdel")
```

**Related concepts**

- **Record input (INPUT)** on page 1482
  The INPUT instruction provides single record input control in an application form.

- **ON INSERT block** on page 1517
- **ON APPEND block** on page 1516
- **ON DELETE block** on page 1519

**ON DELETE block**

The ON DELETE trigger can be used to enable row deletion during a DISPLAY ARRAY dialog. If this block is defined, the dialog will automatically create the delete action. This action can be decorated, enabled and disabled as regular actions.

If the dialog defines an ON ACTION delete interaction block and the ON DELETE block is used, the compiler will stop with error -8408.

When the user fires the delete action, the dialog executes the user code of the ON DELETE block.

The dialog handles only the row deletion action and navigation, you can typically program a validation dialog box to let the user confirm the deletion. The current row in the program array is identified with arr_curr():

```sql
DISPLAY ARRAY arr TO sr.*
...
ON DELETE
```
IF fgl_winQuestion("Delete",
   "Do you want to delete this record?",
   "yes", "no|yes", "help", 0) == "no"
THEN
   LET int_flag = TRUE
END IF

After the user code is executed, the dialog gets the control back and processes the current row as follows:

- If the INT_FLAG global variable is FALSE and STATUS is zero, the current row is deleted from the program array, and the BEFORE ROW block is executed for the next row in the list.
- If the INT_FLAG global variable is TRUE or STATUS is different from zero, the current row is kept in the program array, and the BEFORE ROW block is executed again for the current row.

The DISPLAY ARRAY dialog always resets INT_FLAG to FALSE and STATUS to zero before executing the user code of the ON DELETE block.

If needed, the ON DELETE handler can be configured with action attributes by adding an ATTRIBUTES() clause, as with user-defined action handlers:

ON DELETE ATTRIBUTES(TEXT="%"custlist.delete", IMAGE="listdel")

Related concepts
ON APPEND block on page 1516
ON INSERT block on page 1517
ON UPDATE block on page 1518

ON SELECTION CHANGE block

The ON SELECTION CHANGE trigger can be used to enable multi-row selection and detect when rows are selected or de-selected by the end user during a DISPLAY ARRAY dialog. If this block is defined, multi-row selection is automatically enabled. However, the feature can be enabled/disabled with the setSelectionMode() dialog method.

Related concepts
Multiple row selection on page 1851
Multiple row selection allows the end user to select several rows within a list of records.

ON SORT block

Basics

The ON SORT interaction block can be used to detect when rows have to be sorted in a DISPLAY ARRAY or INPUT ARRAY dialog.

ON SORT is used in two different contexts:

1. In a regular full-list DISPLAY ARRAY / INPUT ARRAY dialog, the ON SORT trigger can be used to detect that a list sort was performed.
2. In a DISPLAY ARRAY using paged mode (ON FILL BUFFER), use ON SORT to detect a sort request from the user and re-fetch the rows from the database in the required order.

ON SORT in regular full-list DISPLAY ARRAY or INPUT ARRAY

In a regular DISPLAY ARRAY / INPUT ARRAY dialog not using paged mode, the ON SORT trigger can be used to detect that a list sort was performed.

When the ON SORT block executes in this context, the (visual) sort is already done by the runtime system and the ON SORT block is only used to execute post-sort tasks, such as displaying current row information.
To display the row position information, use the `arrayToVisualIndex()` dialog method to convert the current program row number to the visual row number:

```
DISPLAY ARRAY arr TO sr.* ...  
...  
ON SORT  
  MESSAGE SFMT( "Row: %1/%2",  
    DIALOG.arrayToVisualIndex( "sr", DIALOG.getCurrentRow("sr") ),  
    DIALOG.getArrayLength( "sr" )  
  )  
...  
```

If needed, you can get the sort column and sort order with the `getSortKey()` and `isSortReverse()` dialog methods:

```
DISPLAY ARRAY arr TO sr.* ...  
...  
ON SORT  
  MESSAGE SFMT( "Sort on %1, %2 order",  
    DIALOG.getSortKey("sr"),  
    IIF( DIALOG.isSortReverse("sr"), "descending", "ascending" )  
  )  
...  
```

**ON SORT in DISPLAY ARRAY using the paged mode**

In a `DISPLAY ARRAY` implementing paged mode with `ON FILL BUFFER` trigger, built-in row sorting is not available because data is provided by pages.

Use the `ON SORT` trigger, to detect a sort request and perform a new SQL query to re-order the rows. In this context, the sort column and sort order are available with the `getSortKey()` and `isSortReverse()` dialog methods:

```
DEFINE key STRING, rev BOOLEAN  
DISPLAY ARRAY arr TO sr.* ...  
...  
ON SORT  
  -- Re-execute the SQL statement to fill the page of rows in ON FILL BUFFER  
  -- Assuming that form field names match table column names  
  LET key = DIALOG.getSortKey("sa")  
  LET rev = DIALOG.isSortReverse("sa")  
  IF key IS NULL THEN  
    CALL execute_sql( NULL )  
  ELSE  
    CALL execute_sql( "ORDER BY " || key || IIF(rev," DESC"," " ) )  
  END IF  

See Paged mode of DISPLAY ARRAY on page 1843 for more details about the paged mode in DISPLAY ARRAY and how to implement sort in this type of record list dialog.

**Related concepts**

- List ordering on page 1862
- List controllers implement a built-in sort. This feature can be disabled if not required.

**ON DRAG_START block**

The `ON DRAG_START` block is executed when the end user begins the drag operation. If this dialog trigger has not been defined, default dragging is enabled for this dialog.
In the ON DRAG_START block, the program typically specifies the type of drag & drop operation by calling `ui.DragDrop.setOperation()` with "move" or "copy". This call will define the default and unique drag operation. If needed, the program can allow another type of drag operation with `ui.DragDrop.addPossibleOperation()`. The end user can then choose to move or copy the dragged object, if the drag & drop target allows it.

If the dragged object can be dropped outside the program, the MIME type and drag/drop data must be defined with `ui.DragDrop.setMimeType()` and `ui.DragDrop.setBuffer()` methods.

Example:

```define dnd ui.DragDrop
define dnd ui.DragDrop
... display array arr to sr.* ...

... on drag_start (dnd)
call dnd.setOperation("move") -- Move is the default operation
call dnd.addPossibleOperation("copy") -- User can toggle to copy if needed
call dnd.setMimeType("text/plain")
call dnd.setBuffer(arr[arr_curr()].cust_name)
...
end display```

Related concepts

- Handle drag & drop data with MIME types on page 1919
- How to handle MIME types with drag & drop?
- The DragDrop class on page 2570
  The `ui.DragDrop` class is used to control the events related to drag & drop events.
- Drag & drop on page 1917
  Explains programming techniques for the drag & drop feature.

**ON DRAG_FINISHED block**

Execution of the ON DRAG_FINISHED block notifies the dialog where the drag started and that the drop operation has been completed or terminated.

Call `ui.DragDrop.getOperation()` to get the final type of operation of the drop. On successful completion, the method returns "move" or "copy"; otherwise the function returns NULL. If NULL is returned, the ON DRAG_FINISHED trigger can be ignored.

In cases of successful moves to a target out of the current DISPLAY ARRAY, the application must remove the transferred data from the source model. For example, if a row was moved from dialog A to B, dialog A will get an ON DRAG_FINISHED execution after the row was dropped into B, which removes the row from the list A.

The ON DRAG_FINISHED interaction block is optional.

```define dnd ui.DragDrop
define dnd ui.DragDrop
... display array arr to sr.* ...

... on drag_start (dnd)
  let last_dragged_row = arr_curr()
...
... on drag_finished (dnd)
  if dnd.getOperation() == "move" then
    call dialog.deleteRow(last_dragged_row)
  end if
... end display```
**Related concepts**

The DragDrop class on page 2570
The `ui.DragDrop` class is used to control the events related to drag & drop events.

Drag & drop on page 1917
Explains programming techniques for the drag & drop feature.

**ON DRAG_ENTER block**

When the `ON DROP` control block is defined, the `ON DRAG_ENTER` block will be executed when the mouse cursor enters the visual boundaries of the drop target dialog. Entering the target dialog is accepted by default if no `ON DRAG_ENTER` block is defined. However, when `ON DROP` is defined, it is recommended that you also define `ON DRAG_ENTER` to prevent the drop of objects with an unsupported MIME type coming from other applications.

The program can decide to disallow or allow a specific drop operation with a call to `ui.DragDrop.setOperation();` passing a NULL to the method will prevent the drop.

To check what MIME type is available in the drag & drop buffer, the program uses the `ui.DragDrop.selectMimeType()` method. This method takes the MIME type as a parameter and returns TRUE if the passed MIME type is used. You can call this method several times to check the availability of different MIME types.

You may also define the visual effect when hovering over the target list with `ui.DragDrop.setFeedback()`.

```define
define dnd ui.dragdrop
... display array arr to sr.* ...
... on drag_enter (dnd)
  if dnd.selectmimetype("text/plain") then
    call dnd.setoperation("copy")
    call dnd.setfeedback("all")
  else
    call dnd.setoperation(null)
  end if
on drop (dnd)
... end display
```

Once the mouse has entered the target area, subsequent mouse cursor moves can be detected with the `ON DRAG_OVER` trigger.

When using a table or tree-view as drop target, you can control the visual effect when the mouse moves over the rows, depending on the type of drag & drop you want to achieve.

Basically, a dragged object can be:

1. Inserted in between two rows (visual effect must show where the object will be inserted)
2. Copied/merged to the current row (visual effect must show the row under the mouse)
3. Dropped somewhere on the target widget (the exact location inside the widget does not matter)

The visual effect can be defined with the `ui.DragDrop.setFeedback()` method, typically called in the `ON DRAG_ENTER` block.

The values to pass to the `setFeedback()` method to get the desired visual effects described are respectively:

1. insert (default)
2. select
3. all

```define
define dnd ui.dragdrop
... display array arr to sr.* ...
```
ON DRAG_ENTER (dnd)
  IF canDrop() THEN
    CALL dnd.setOperation(NULL)
  ELSE
    CALL dnd.setFeedback("select")
  END IF
END IF
...
END DISPLAY

Related concepts
The DragDrop class on page 2570
The ui.DragDrop class is used to control the events related to drag & drop events.

Drag & drop on page 1917
Explains programming techniques for the drag & drop feature.

ON DRAG_OVER block
When the ON DROP control block is defined, the ON DRAG_OVER block will be executed after ON DRAG_ENTER, when the mouse cursor is moving over the drop target, or when the drag & drop operation has changed (toggling copy/move).

ON DRAG_OVER will be called only once per row, even if the mouse cursor moves over the row.

In the ON DRAG_OVER block, the method ui.DragDrop.getLocationRow() returns the index of the row in the target array, and can be used to allow or disallow the drop. When using a tree-view, you must also check the index returned by the ui.DragDrop.getLocationParent() method to detect if the object was dropped as a sibling or as a child node, and allow/disallow the drop operation accordingly.

The program can change the drop operation at any execution of the ON DRAG_OVER block. You can disallow or allow a specific drop operation with a call to ui.DragDrop.setOperation(); passing a NULL to the method will disallow the drop.

The current operation (returned by ui.DragDrop.getOperation()) is the value set in previous ON DRAG_ENTER or ON DRAG_OVER events, or the operation selected by the end user, if it can toggle between copy and move. Thus, ON DRAG_OVER can occur even if the mouse position has not changed.

If dropping has been prevented with ui.DragDrop.setOperation(NULL) in the previous ON DRAG_OVER event, the program can reset the operation to allow a drop with a call to ui.DragDrop.setOperation() with the operation parameter "move" or "copy".

ON DRAG_OVER will not be called if drop has been disabled in ON DRAG_ENTER with ui.DragDrop.setOperation(NULL)

ON DRAG_OVER is optional, and must only be defined if the operation or the acceptance of the drag object depends on the target row of the drop target.

DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr TO sr.* ...
...
ON DRAG_ENTER (dnd)
...
ON DRAG_OVER (dnd)
  IF arr[dnd.getLocationRow()].acceptsCopy THEN
    CALL dnd.setOperation("copy")
  ELSE
    CALL dnd.setOperation(NULL)
  END IF
ON DROP (dnd)
...
END DISPLAY
During a drag & drop process, the end user (or the target application) can decide to modify the type of the operation, to indicate whether the dragged object has to be copied or moved from the source to the target. For example, in a typical file explorer, by default files are moved when doing a drag & drop on the same disk. To make a copy of a file, you must press the Ctrl key while doing the drag & drop with the mouse.

In the drop target dialog, you can detect such operation changes in the ON_DRAG_OVER trigger and query the ui.DragDrop object for the current operation with ui.DragDrop.getOperation(). In the drag source dialog, you typically check ui.DragDrop.getOperation() in the ON_DRAG_FINISHED trigger to know what type of operation occurred, to keep (“copy” operation) or delete (“move” operation) the original dragged object.

This example tests the current operation in the drop target list and displays a message accordingly:

```plaintext
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr TO sr.* ...
...
ON DRAG_ENTER (dnd)
...
ON DRAG_OVER (dnd)
  CASE dnd.getOperation()
    WHEN "move"
      MESSAGE "The object will be moved to row ", dnd.getLocationRow()
    WHEN "copy"
      MESSAGE "The object will be copied to row ", dnd.getLocationRow()
  END CASE
...
ON DROP (dnd)
...
END DISPLAY
```

Related concepts

The DragDrop class on page 2570
The ui.DragDrop class is used to control the events related to drag & drop events.

Tree views on page 1889
Describes how to implement tree views.

Drag & drop on page 1917
Explains programming techniques for the drag & drop feature.

ON DROP block

To enable drop actions on a list, you must define the ON DROP block; otherwise the list will not accept drop actions.

The ON DROP block is executed after the end user has released the mouse button to drop the dragged object. ON DROP will not occur if drop has been disallowed in the previous ON_DRAG_OVER event or in ON_DRAG_ENTER with a call to ui.DragDrop.setOperation(NULL).

When ON DROP executes, the MIME type of the dragged object can be checked with ui.DragDrop.getSelectedMimeType(). Then call the ui.DragDrop.getBuffer() method to retrieve drag & drop data from external applications.

Ideally, the drop operation is accepted (there is no need for additional calls to ui.DragDrop.setOperation()).

In this block, the ui.DragDrop.getLocationRow() method returns the index of the row in the target array, and can be used to execute the code to get the drop data / object into the row that has been chosen by the user.

```plaintext
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr TO sr.* ...
...
ON DROP (dnd)
```
If the drag & drop operations are local to the same list or tree-view controller, you can use the `ui.DragDrop.dropInternal()` method to simplify the code. This method implements the typical move of the dragged rows or tree-view node. This is especially useful in case of a tree-view, but is also the preferred way to move rows around in simple tables.

This ON DROP code example uses the `dropInternal()` method:

```plaintext
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr_tree TO sr_tree.* ...
...
ON DROP (dnd)
  CALL dnd.dropInternal()
...
END DISPLAY
```

If you want to implement by hand the code to drop a node in a tree-view, you must check the index returned by the `ui.DragDrop.getLocationParent()` method to detect if the object was dropped as a sibling or as a child node, and execute the code corresponding to the drop operation. If the drop target row index returned by `getLocationRow()` is a child of the parent row index returned by `getLocationParent()`, the new row must be inserted before `getLocationRow()`; otherwise the new row must be added as a child of the parent node identified by `getLocationParent()`.

**Related concepts**

- **The DragDrop class** on page 2570

  The `ui.DragDrop` class is used to control the events related to drag & drop events.

- **Drag & drop** on page 1917

  Explains programming techniques for the drag & drop feature.

**DIALOG control instructions**

*Dialog control instructions* are language instructions dedicated to dialog control, to programmatically force the dialog to behave in a given way.

For example the `NEXT FIELD` instruction forces the focus to a specific form field.

**NEXT FIELD instruction**

**Understanding the NEXT FIELD instruction**

The `NEXT FIELD` `field-name` instruction gives the focus to the specified field. This instruction can be used to control field input, in `BEFORE FIELD`, `ON CHANGE` or `AFTER FIELD` blocks, it can also force a `DISPLAY ARRAY` or `INPUT ARRAY` to stay in the current row when `NEXT FIELD` is used in the `AFTER ROW` block.

If it exists, the `BEFORE FIELD` block of the corresponding field is executed.

In editable dialogs, the purpose of the `NEXT FIELD` instruction is to give the focus to an editable field. Make sure that the field specified in `NEXT FIELD` is active and editable, or use `NEXT FIELD CURRENT`.

In a `DISPLAY ARRAY` using the `FOCUSONFIELD` attribute, `NEXT FIELD` can be used in conjunction with `DIALOG.setCurrentRow()`, to set the focus to a specific cell in the list.

Instead of the `NEXT FIELD` instruction, you can use the `DIALOG.nextField("field-name")` method to register a field, for example when the name is not known at compile time. However, this method only registers the field. It does not stop code execution, like the `NEXT FIELD` instruction does. You must execute a `CONTINUE DIALOG` to get the same behavior as `NEXT FIELD`. 
Form field identification with NEXT FIELD

With the NEXT FIELD instruction, fields are identified by the form field name specification, not the program variable name used by the dialog.

Form fields are bound to program variables with the binding clause of the dialog instruction (INPUT variable-list FROM field-list, INPUT BY NAME variable-list, CONSTRUCT BY NAME sql ON column-list, CONSTRUCT sql ON column-list FROM field-list, INPUT ARRAY array-name FROM screen-array.*).

The field name specification can be any of the following:

- field-name
- table-name.field-name
- screen-record-name.field-name
- FORMONLY.field-name

Here are some examples:

- "cust_name"
- "customer.cust_name"
- "cust_screen_record.cust_name"
- "item_screen_array.item_label"
- "formonly.total"

When no field name prefix is used, the first form field matching that simple field name is used.

When using a prefix in the field name specification, it must match the field prefix assigned by the dialog field binding method used at the beginning of the interactive statement: When no screen-record has been explicitly specified in the field binding clause (for example, when using INPUT BY NAME variable-list), the field prefix must be the database table name or FORMONLY, as defined in the form file, or any valid screen-record using that field. When the FROM clause of the dialog specifies an explicit screen-record (for example, in INPUT variable-list FROM screen-record.* / field-list-with-screen-record-prefix or INPUT ARRAY array-name FROM screen-array.*), the field prefix must be the screen-record name used in the FROM clause.

Abstract field identification is supported with the CURRENT, NEXT and PREVIOUS keywords. These keywords represent the current, next and previous fields respectively. When using FIELD ORDER FORM, the NEXT and PREVIOUS options follow the tabbing order defined by the form. Otherwise, they follow the order defined by the input binding list (with the FROM or BY NAME clause).

In a procedural dialog, if the focus is in the first field of an INPUT or CONSTRUCT sub-dialog, NEXT FIELD PREVIOUS will jump out of the current sub-dialog and set the focus to the previous sub-dialog. If the focus is in the last field of an INPUT or CONSTRUCT sub-dialog, NEXT FIELD NEXT will jump out of the current sub-dialog and set the focus to the next sub-dialog. NEXT FIELD NEXT or NEXT FIELD PREVIOUS also jumps to another sub-dialog when the focus is in a DISPLAY ARRAY sub-dialog. However, when using an INPUT ARRAY sub-dialog, NEXT FIELD NEXT from within the last column will loop to the first column of the current row, and NEXT FIELD PREVIOUS from within the first column will jump to the last column of the current row - the focus stays in the current INPUT ARRAY sub-dialog. When another sub-dialog gets the focus because of a NEXT FIELD NEXT/PREVIOUS, the newly-selected field depends on the sub-dialog type, following the tabbing order as if the end-user had pressed the tab or Shift-Tab key combination.

NEXT FIELD to a non-editable field

Non-editable fields are:

- Fields defined with the NOENTRY attribute in the form.
- Fields defined as PHANTOM fields in the form.
- Fields disabled at runtime with DIALOG.setFieldActive().
- Fields using a widget that does not allow input, such as a LABEL.
In an INPUT, INPUT ARRAY or CONSTRUCT instruction, if a NEXT FIELD instruction specifies a non-editable field, the BEFORE FIELD block of that field is executed. Then the dialog tries to give the focus to that field. Since the field cannot get the focus, the dialog will perform the last pressed navigation key (Tab, Shift-Tab, Left, Right, Up, Down, Accept) and execute the related control blocks, including the AFTER FIELD block of the non-editable field. If no last key is identified, the dialog considers Tab as fallback and moves to the next editable field as defined by the FIELD ORDER mode used by the dialog.

**Note:** In an INPUT ARRAY, doing a NEXT FIELD to a non-editable field which has no editable fields in the last navigation direction, the dialog may move to the next or previous row and give the focus to the first or last editable field of that row.

When selecting a non-editable field with NEXT FIELD NEXT, the runtime system will re-select the current field since it is the next editable field in the dialog. As a result the end user sees no change.

**Important:** Doing a NEXT FIELD to a non-editable field can lead to infinite loops in the dialog; To stay in the current field/row, use NEXT FIELD CURRENT instead.

### NEXT FIELD in procedural DIALOG blocks

In a procedural dialog block, the NEXT FIELD field-name instruction gives the focus to the specified field controlled by INPUT, INPUT ARRAY or CONSTRUCT, or to a read-only list when using DISPLAY ARRAY.

When using a DISPLAY ARRAY sub-dialog, it is possible to give the focus to the list, by specifying the name of the first column as argument for NEXT FIELD.

If the target field specified in the NEXT FIELD instruction is inside the current sub-dialog, neither AFTER FIELD nor AFTER ROW will be invoked for the field or list you are leaving. However, the BEFORE FIELD control blocks of the destination field (or the BEFORE ROW in case of read-only list) will be executed.

If the target field specified in the NEXT FIELD instruction is outside the current sub-dialog, the AFTER FIELD, AFTER INSERT, AFTER ROW, and AFTER INPUT, AFTER DISPLAY, AFTER /CONSTRUCT control blocks will be invoked for the field or list you are leaving. Form-level validation rules will also be checked, as if the user had selected the new sub-dialog himself. This guarantees the current sub-dialog is left in a consistent state. The BEFORE INPUT, BEFORE DISPLAY, BEFORE CONSTRUCT, BEFORE ROW and the BEFORE FIELD control blocks of the destination field/list will then be executed.

### NEXT FIELD in record list control blocks

When using NEXT FIELD in AFTER ROW or in ON ROW CHANGE of a DISPLAY ARRAY or INPUT ARRAY, the dialog will stay in the current row and give control back to the user.

This behavior allows you to implement data input rules:

```
AFTER ROW
    IF NOT int_flag AND arr_count()<=arr_cur() THEN
        IF arr[arr_cur()].it_count * arr[arr_cur()].it_value > maxval THEN
            ERROR "Amount of line exceeds max value."
        NEXT FIELD item_count
    END IF
END IF
```

If a condition is not met for a given field, do a NEXT FIELD to that field to point the user to the problem. If a global condition for the current row is not met, use NEXT FIELD CURRENT to stay in the current row and current field.

### Related concepts

- **Giving the focus to a form element** on page 1731
  - How to force the focus to move or stay in a specific form element using program code.

- **Understanding multiple dialogs** on page 1585
Multiple dialogs are defined with DIALOG blocks inside a FUNCTION.

CLEAR instruction in dialogs

The CLEAR field-list and CLEAR SCREEN ARRAY screen-array.* instructions clear the value buffer of specified form fields. The buffers are directly changed in the current form, and the program variables bound to the dialog are left unchanged. CLEAR can be used outside any dialog instruction, such as the DISPLAY BY NAME / TO instructions.

When a dialog is configured with the UNBUFFERED mode, there is no reason to clear field buffers since any variable assignment will synchronize field buffers. Actually, changing the field buffers with DISPLAY or CLEAR instruction in an UNBUFFERED dialog will have no visual effect, because the variables bound to the dialog will be used to reset the field buffer just before giving control back to the user. To clear fields of an UNBUFFERED dialog, just set to NULL the variables bound to the dialog. However, when using a CONSTRUCT, no program variables are associated with the dialog and no UNBUFFERED concept exits, and the CLEAR or DISPLAY TO / BY NAME instructions are the only way to modify the CONSTRUCT fields.

A screen array with a screen-line specification doesn't make much sense in a GUI application using TABLE containers, you can therefore use the CLEAR SCREEN ARRAY instruction to clear all rows of a list.

Related concepts

Static display (DISPLAY/ERROR/MESSAGE/CLEAR) on page 1452
This section explains the instructions displaying static information to application forms, such as DISPLAY, ERROR, MESSAGE, CLEAR.

DISPLAY TO / BY NAME instruction

The DISPLAY variable-list TO field-list or DISPLAY BY NAME variable-list instruction fills the value buffers of specified form fields with the values contained in the specified program variables. The DISPLAY TO / BY NAME instruction changes the buffers directly in the current form, not the program variables bound to the dialog. DISPLAY can be used outside any dialog instruction, in the same way as the CLEAR instruction. DISPLAY also sets the modification flag of fields.

As DIALOG is typically used with the UNBUFFERED mode, there is no reason to set field buffers in a DIALOG block since any variable assignment will synchronize field buffers. Actually, changing the field buffers with the DISPLAY or CLEAR instruction will have no visual effect if the fields are used by a dialog working in UNBUFFERED mode, because the variables bound to the dialog will be used to reset the field buffer just before giving control back to the user. So if you want to set field values, just assign the variables and the fields will be synchronized. However, when using a CONSTRUCT binding, you may want to set field buffers with this DISPLAY instruction, as there are no program variables bound to fields (with CONSTRUCT, only one string variable is bound to hold the SQL condition).

Instead of using a DISPLAY instruction to set the modification flag of fields to simulate user input, use the DIALOG.setFieldTouched() method instead.

Related concepts

Input field modification flag on page 1723
Each input field controlled by a dialog instruction has a modification flag.

CONTINUE DIALOG instruction

The CONTINUE DIALOG statement continues the execution of a DIALOG instruction, skipping all statements appearing after this instruction.

Control returns to the dialog instruction, which executes remaining control blocks as if the program reached the end of the current control block. Then the control goes back to the user and the dialog waits for a new event.

The CONTINUE DIALOG statement is useful when program control is nested within multiple conditional statements, and you want to return control to the user by skipping the rest of the statements.
In the following code example, an ON ACTION block gives control back to the dialog, skipping all instructions below line 04:

```
ON ACTION zoom
    IF p_cust.cust_id IS NULL OR p_cust.cust_name IS NULL THEN
        ERROR "Zoom window cannot be opened if no info to identify customer"
        CONTINUE DIALOG
    END IF
    IF p_cust.cust_address IS NULL THEN
        ...
```

If `CONTINUE DIALOG` is called in a control block that is not AFTER DIALOG, further control blocks might be executed depending on the context. Actually, `CONTINUE DIALOG` just instructs the dialog to continue as if the code in the control block was terminated (it is a kind of GOTO end_of_control_block). However, when executed in AFTER DIALOG, the focus returns to the current field or read-only list. In this case the BEFORE ROW and BEFORE FIELD triggers will be invoked.

A `CONTINUE DIALOG` in AFTER FIELD, AFTER INPUT, AFTER DISPLAY or AFTER CONSTRUCT will only stop the program flow of the current block of statements; instructions after `CONTINUE DIALOG` will not be executed. If the user has selected a field in a different sub-dialog, this new field will get the focus and all necessary AFTER / BEFORE control blocks will be executed.

In case of input error in a field, the best practice is to use a NEXT FIELD instruction to stay in the dialog and set the focus to the field that the user has to correct.

**Related concepts**

NEXT FIELD instruction on page 1497

**EXIT DIALOG instruction**

The EXIT DIALOG statement terminates a procedural DIALOG block without any further control block execution.

**Note:** When used in a declarative DIALOG block, the EXIT DIALOG instruction does only make sense when the declarative dialog block is included in a procedural dialog block with the SUBDIALOG clause.

Program flow resumes at the Instruction following the END DIALOG keywords. Blocks such as AFTER DIALOG will not be executed.

```
ON ACTION quit
    EXIT DIALOG
```

When leaving the DIALOG instruction, all form items used by the dialog will be disabled until another interactive statement takes control.

The EXIT DIALOG instruction leaves the DIALOG block immediately, while CANCEL DIALOG makes some additional tasks.

**Related concepts**

CANCEL DIALOG instruction on page 1641

ACCEPT DIALOG instruction on page 1640

**ACCEPT DIALOG instruction**

The ACCEPT DIALOG statement validates all input fields bound to the DIALOG instruction and leaves the block if no error is raised.

**Note:** When used in a declarative DIALOG block, the ACCEPT DIALOG instruction does only make sense when the declarative dialog block is included in a procedural dialog block with the SUBDIALOG clause.

When defined in the dialog block, ON CHANGE, AFTER FIELD, AFTER ROW, AFTER INPUT, AFTER DISPLAY, AFTER CONSTRUCT control blocks will be executed when ACCEPT DIALOG is performed.

The statements appearing after the ACCEPT DIALOG instruction will be skipped.
You typically code an **ACCEPT DIALOG** in an **ON ACTION accept** block:

```
ON ACTION accept ACCEPT DIALOG
```

**Important:** Any usage of **ACCEPT DIALOG** outside an **ON ACTION accept** block is not intended and its behavior is undefined.

Input field validation is a process that does several successive validation tasks:

1. The current field value is checked, depending on the variable data type (for example, the user must input a valid date in a **DATE** field).
2. **NOT NULL** field attributes are checked for all input fields. This attribute forces the field to have a value set by program or entered by the user. If the field contains no value, the constraint is not satisfied. Input values are right-trimmed, so if the user inputs only spaces, this corresponds to a **NULL** value which does not fulfill the **NOT NULL** constraint.
3. **REQUIRED** field attributes are checked for all input fields. This attribute forces the field to have a default value, or to be modified by the user or by program with a **DISPLAY TO** / **BY NAME** or **DIALOG.setFieldTouched()** call. If the field was not modified during the dialog, the **REQUIRED** constraint is not satisfied.
4. **INCLUDE** field attributes are checked for all input fields. This attribute forces the field to contain a value that is listed in the include list. If the field contains a value that is not in the list, the constraint is not satisfied.

If a field does not satisfy one of these constraints, dialog termination is canceled, an error message is displayed, and the focus goes to the first field causing a problem.

After input field validation has succeeded, different types of control blocks will be executed, such as **AFTER FIELD**, **AFTER ROW**, **AFTER INPUT** and **AFTER DIALOG**.

In order to validate some parts of the dialog without leaving the block, use the **DIALOG.validate()** method.

**Related concepts**

**Input field modification flag** on page 1723
Each input field controlled by a dialog instruction has a modification flag.

**CANCEL DIALOG instruction** on page 1641

**EXIT DIALOG instruction** on page 1640

**ui.Dialog.accept** on page 2520
Validates and terminates the dialog.

**CANCEL DIALOG instruction**

The **CANCEL DIALOG** statement terminates a procedural **DIALOG** block, after executing the **AFTER INPUT**, **AFTER DISPLAY**, **AFTER CONSTRUCT** control block of the current sub-dialog, and the **AFTER DIALOG** control block.

**Note:** When used in a declarative **DIALOG** block, the **CANCEL DIALOG** instruction does only make sense when the declarative dialog block is included in a procedural dialog block with the **SUBDIALOG** clause.

The **CANCEL DIALOG** instruction can be used from multiple dialogs to mimic the cancel default action of single dialogs.

The **CANCEL DIALOG** instruction makes the following:

1. Set the **INT_FLAG** register to **TRUE**.
2. If defined, execute the code in the **AFTER INPUT**, **AFTER DISPLAY** or **AFTER CONSTRUCT** block of the current sub-dialog.
3. If defined, execute the code in the **AFTER DIALOG** block.

The statements appearing after the **CANCEL DIALOG** instruction will be skipped.
You typically code an CANCEL DIALOG in an ON ACTION cancel block:

```plaintext
ON ACTION cancel CANCEL DIALOG
```

**Note:** The default settings regarding action attributes for the cancel action define the validate attribute to "no", in order to avoid current field validation for this action. This is important when using the UNBUFFERED mode. For more details, see Actions configuration for field validation on page 1722.

**Related concepts**
- EXIT DIALOG instruction on page 1640
- ACCEPT DIALOG instruction on page 1640
- ui.Dialog.cancel on page 2527
  Cancels a parent dialog from a sub-dialog.

**CANCEL DELETE instruction**

In a list controlled by an INPUT ARRAY, row deletion can be canceled by using the CANCEL DELETE instruction in the BEFORE DELETE block. Using this instruction in a different place will generate a compilation error.

When the CANCEL DELETE instruction is executed, the current BEFORE DELETE block is terminated without any other trigger execution (no BEFORE ROW or BEFORE FIELD is executed), and the program execution continues in the user event loop.

You can, for example, prevent row deletion based on some condition:

```plaintext
BEFORE DELETE
  IF user_can_delete() == FALSE THEN
    ERROR "You are not allowed to delete rows"
    CANCEL DELETE
  END IF
```

The instructions that appear after CANCEL DELETE will be skipped.

If the row deletion condition is known before the delete action occurs, disable the delete action to prevent the user from performing a delete row action with the DIALOG.setActionActive() method:

```plaintext
CALL DIALOG.setActionActive("delete", FALSE)
```

It is also possible to prevent the user from deleting rows with the DELETE ROW = FALSE option in the ATTRIBUTE clause.

**Related concepts**
- BEFORE DELETE block on page 1547

**CANCEL INSERT instruction**

In a list controlled by an INPUT ARRAY, row creation can be canceled by the program with the CANCEL INSERT instruction. This instruction can only be used in the BEFORE INSERT and AFTER INSERT control blocks. If it appears at a different place, the compiler will generate an error.

The instructions that appear after CANCEL INSERT will be skipped.

If the row creation condition is known before the insert/append action occurs, disable the insert and/or append actions to prevent the user from creating new rows, with DIALOG.setActionActive():

```plaintext
CALL DIALOG.setActionActive("insert", FALSE)
CALL DIALOG.setActionActive("append", FALSE)
```

However, this will not prevent the user from appending a new temporary row at the end of the list, when moving down after the last row. To prevent row creation completely, use the INSERT ROW = FALSE and APPEND ROW
CANCEL INSERT in BEFORE INSERT

A CANCEL INSERT executed inside a BEFORE INSERT block prevents the new row creation. The following tasks are performed:

1. No new row will be created (the new row is not yet shown to the user).
2. The BEFORE INSERT block is terminated (further instructions are skipped).
3. The BEFORE ROW and BEFORE FIELD triggers are executed.
4. Control goes back to the user.

You can, for example, cancel a row creation if the user is not allowed to create rows:

```dialog
BEFORE INSERT
  IF NOT user_can_insert THEN
    ERROR "You are not allowed to insert rows"
    CANCEL INSERT
  END IF
```

Executing CANCEL INSERT in BEFORE INSERT will also cancel a temporary row creation, except when there are no more rows in the list. In this case, CANCEL INSERT will just be ignored and leave the new row as is (otherwise, the instruction would loop without end). You can prevent automatic temporary row creation with the AUTO APPEND=FALSE attribute. If AUTO APPEND=FALSE and a CANCEL INSERT is executed in BEFORE INSERT (user has invoked an append action), the temporary row will be deleted and list will remain empty if it was the last row.

CANCEL INSERT in AFTER INSERT

A CANCEL INSERT executed inside an AFTER INSERT block removes the newly created row. The following tasks are performed:

1. The newly created row is removed from the list (the row exists now and user has entered data).
2. The AFTER INSERT block is terminated (further instructions are skipped).
3. The BEFORE ROW and BEFORE FIELD triggers are executed.
4. The control goes back to the user.

You can, for example, cancel a row insertion if a database error occurs when you try to insert the row into a database table:

```dialog
AFTER INSERT
  WHENEVER ERROR CONTINUE
  LET r = DIALOG.getCurrentRow("s_items")
  INSERT INTO items VALUES ( p_items[r].* )
  WHENEVER ERROR STOP
  IF SQLCA.SQLCODE<>0 THEN
    ERROR SQLERRMESSAGE
    CANCEL INSERT
  END IF
```

Related concepts
BEFORE DELETE block on page 1547
Appending rows in INPUT ARRAY on page 1848
Rows appended at the end of an editable list are temporary until they are edited.

**Examples**

DIALOG (multiple dialog) usage examples.

**Example 1: DIALOG controlling two lists**

Form file "lists.per":

```plaintext
LAYOUT
GRID
{
  <t t1          >
  [f11  |f12         ]
  <
  <t t2          >
  [f21  |f22         ]
  <
}
END
END
ATTRIBUTES
EDIT f11 = FORMONLY.column_11;
EDIT f12 = FORMONLY.column_12;
EDIT f21 = FORMONLY.column_21;
EDIT f22 = FORMONLY.column_22;
END
INSTRUCTIONS
SCREEN RECORD sr1(FORMONLY.column_11,FORMONLY.column_12);
SCREEN RECORD sr2(FORMONLY.column_21,FORMONLY.column_22);
END
```

Program source code:

```plaintext
DEFINE
   arr1 DYNAMIC ARRAY OF RECORD
   column_11 INTEGER,
   column_12 VARCHAR(10)
END RECORD,
arr2 DYNAMIC ARRAY OF RECORD
   column_21 INTEGER,
   column_22 VARCHAR(10)
END RECORD
MAIN
DEFINE i INTEGER
FOR i = 1 TO 20
  LET arr1[i].column_11 = i
  LET arr1[i].column_12 = "aaa " || i
  LET arr2[i].column_21 = i
  LET arr2[i].column_22 = "aaa " || i
END FOR
OPTIONS INPUT WRAP
OPEN FORM f FROM "lists"
DISPLAY FORM f
DIALOG ATTRIBUTES(UNBUFFERED)
   DISPLAY ARRAY arr1 TO sr1.*
   BEFORE DISPLAY
      MESSAGE "We are in list one"
END DISPLAY
   DISPLAY ARRAY arr2 TO sr2.*
   BEFORE DISPLAY
      MESSAGE "We are in list two"
END DISPLAY
```
Example 2: DIALOG with CONSTRUCT and DISPLAY ARRAY

Form file "form1.per":

LAYOUT
GRID
{
    <g g1>
    Name:     [f1]
    State:    [f2]
    City:     [f3]
    Zip-code: [f4]
    [ :cc :sr ]
    <
    <g g2>
    <t t1>
    Id        Name
    [c1 |c2 ]
    [c1 |c2 ]
    [c1 |c2 ]
    <
    <
    [:cw ]
}
END

ATTRIBUTES
GROUP g1: TEXT = "Search criteria";
EDIT f1 = FORMONLY.cust_name TYPE VARCHAR;
EDIT f2 = FORMONLY.cust_state TYPE VARCHAR;
EDIT f3 = FORMONLY.cust_city TYPE VARCHAR;
EDIT f4 = FORMONLY.cust_zipcode TYPE VARCHAR;
BUTTON cc: clear, TEXT="Clear";
BUTTON sr: fetch, TEXT="Fetch";
GROUP g2: TEXT = "Customer list";
EDIT c1 = FORMONLY.c_id TYPE INTEGER;
EDIT c2 = FORMONLY.c_name TYPE VARCHAR;
BUTTON cw: close;
END

INSTRUCTIONS
SCREEN RECORD sr (FORMONLY.c_id, FORMONLY.c_name);
END

Program source code:

MAIN
    DEFINE custarr DYNAMIC ARRAY OF RECORD
        c_id INTEGER,
        c_name VARCHAR(50)
    END RECORD
    DEFINE where_clause STRING
    OPTIONS INPUT WRAP
    OPEN FORM f1 FROM "form1"
    DISPLAY FORM f1
DIALOG ATTRIBUTES(FIELD ORDER FORM, UNBUFFERED)

    CONSTRUCT BY NAME where_clause
        ON cust_name, cust_state, cust_city, cust_zipcode
        ON ACTION clear
            CLEAR cust_name, cust_state, cust_city, cust_zipcode
        END CONSTRUCT

    DISPLAY ARRAY custarr TO sr.*
        BEFORE ROW
            MESSAGE SFMT("Row: %1/%2", DIALOG.getCurrentRow("sr"),
                DIALOG.getArrayLength("sr"))
        END DISPLAY

    ON ACTION fetch
        MESSAGE "Where: ", where_clause
        -- Execute SQL query here to fill custarr ... 

    ON ACTION close
        EXIT DIALOG

END DIALOG

END MAIN

Example 3: DIALOG with SUBDIALOG

Form file "comment.per":

LAYOUT
    GRID
    {
        [cmt ]
    }
END

TEXTEDIT cmt = FORMONLY.the_comment, STRETCH=BOTH;
END

The module "comment.4gl":

PUBLIC TYPE cb_comment_event FUNCTION (event STRING)
PRIVATE DEFINE cb_ce cb_comment_event

PRIVATE DEFINE the_comment VARCHAR(200)

PUBLIC FUNCTION set_event_callback(f cb_comment_event)
    LET cb_ce = f
END FUNCTION

PUBLIC FUNCTION set_event_callback_2(f FUNCTION (event STRING))
    LET cb_ce = f
END FUNCTION

PUBLIC FUNCTION set_comment(s STRING)
    LET the_comment = s
END FUNCTION

PUBLIC FUNCTION get_comment()
    RETURN the_comment
END FUNCTION
DIALOG comment_input(
    INPUT BY NAME the_comment ATTRIBUTES(WITHOUT DEFAULTS)
    ON ACTION add_sep ATTRIBUTES(TEXT="Add sep")
        LET the_comment = the_comment || "\n---\n"
        CALL fgl_dialog_setcursor(LENGTH(the_comment)+1)
        IF cb_ce IS NOT NULL THEN
            CALL cb_ce("comment_changed")
        END IF
    ON ACTION clr_cmt ATTRIBUTES(TEXT="Clear")
        LET the_comment = NULL
        IF cb_ce IS NOT NULL THEN
            CALL cb_ce("comment_changed")
        END IF
    END INPUT
END DIALOG

Form file "form1.per":

LAYOUT
  VBOX
  GRID
  {
    Id: [f1 ]
    Name: [f2 ]
  }
END
FORM "comment"
END
END
ATTRIBUTES
EDIT f1 = FORMONLY.cust_id TYPE INTEGER;
EDIT f2 = FORMONLY.cust_name TYPE VARCHAR;
END

Program source code:

IMPORT FGL comment

MAIN
  DEFINE cust RECORD
    cust_id INTEGER,
    cust_name VARCHAR(50)
  END RECORD

  OPTIONS INPUT WRAP

  OPEN FORM f1 FROM "form1"
  DISPLAY FORM f1

  CALL comment.set_event_callback(FUNCTION event_callback)

  LET cust.cust_id = 199
  LET cust.cust_name = "Mike Patterson"
  CALL comment.set_comment("Some comment about this customer...")

  DIALOG ATTRIBUTES(FIELD ORDER FORM, UNBUFFERED)
    INPUT BY NAME cust.* ATTRIBUTES(WITHOUT DEFAULTS)
    ON ACTION check_exists ATTRIBUTES(TEXT="Check")
        MESSAGE CURRENT HOUR TO FRACTION(3), ": Check if customer exists"
    END INPUT
SUBDIALOG comment_input

ON ACTION close
   DISPLAY "Comment:", comment.get_comment()
   EXIT DIALOG

END DIALOG
END MAIN

FUNCTION event_callback(event STRING)
   CASE event
      WHEN "comment_changed"
         MESSAGE CURRENT HOUR TO FRACTION(3), ": Comment changed..."
         DISPLAY "Comment: ", comment.get_comment()
   END CASE
END FUNCTION

Declarative dialogs (DIALOG - at module level)
DIALOG/END DIALOG defined at module level implement declarative dialogs that can be used in procedural dialogs or in parallel dialogs.

Understanding declarative dialogs
Declarative dialogs are defined with DIALOG blocks at the module level.

A declarative dialog block is a module element defined at the same level as a FUNCTION or REPORT routine:

```
-- Module orders.4gl
SCHEMA stock
DEFINE arr DYNAMIC ARRAY OF RECORD LIKE orders.*
DIALOG orders_dlg()
   DEFINE x INT
   DISPLAY ARRAY arr TO sr_orders.*
   ...
   END DISPLAY
END DIALOG
```

The name of a declarative dialog is mandatory. It can be referenced by a SUBDIALOG clause, by a START DIALOG and TERMINATE DIALOG instruction, and can identify sub-dialog actions with a prefix. Specifically, the name of the declarative dialog will be referenced in a START DIALOG and TERMINATE DIALOG instruction to implement parallel dialogs.

When using the DIALOG keyword inside a declarative dialog block to use ui.Dialog class methods, it references the current instance of the dialog object.

Related concepts
Split views on page 1900
These topics describe split view programming in the language.

Windows and forms on page 1131
The section describes the concept of windows and forms in the language.

fgl_eventloop() on page 2284
Waits for a user interaction event.

**Syntax of the declarative DIALOG block**
The declarative DIALOG block defines an interactive instruction that can be used by a parent DIALOG, or as parallel dialog.

**Syntax**

```
PRIVATE | PUBLIC | DIALOG dialog-name ()
   define-block
   menu-block
   record-input-block
   construct-block
   display-array-block
   input-array-block
END DIALOG
```

1. `dialog-name` defines the identifier for the declarative DIALOG block.

**Note:** Unlike procedural DIALOG blocks, a declarative DIALOG block defines only one MENU, INPUT, CONSTRUCT, DISPLAY ARRAY or INPUT ARRAY sub-block.

where `define-block` is a local variable declaration block.

where `menu-block` is:

```
MENU
   BEFORE MENU
   menu-statement
   [...]
   menu-option
   [...]
END MENU
```

where `menu-option` is:

```
COMMAND option-name
   [option-comment] HELP help-number
   menu-statement
   [...]
COMMAND KEY (key-name) option-name
   [option-comment] HELP help-number
   menu-statement
   [...]
COMMAND KEY (key-name)
   menu-statement
   [...]
ON ACTION action-name
  _ATTRIBUTES (action-attributes-menu)
   menu-statement
   [...]
```

where `action-attributes-menu` is:

```
TEXT = string
COMMENT = string
IMAGE = string
ACCELERATOR = string
DEFAULTVIEW = YES | NO | AUTO
```
CONTEXTMENU = { YES ↓ NO ↓ AUTO ↓ ↓,.... ↓ }

DISCLOSUREINDICATOR ↓,.... ↓

where menu-statement is:

statement
  → NEXT OPTION option
  → SHOW OPTION ALL ↓ option ↓,.... ↓
  → HIDE OPTION ALL ↓ option ↓,.... ↓

1. key-name is a hot-key identifier (like F11 or Control-z).
2. option-name is a string expression defining the label of the menu option and identifying the action that can be executed by the user.
3. option-comment is a string expression containing a description for the menu option, displayed when option-name is the current.
4. help-number is an integer that allows you to associate a help message number with the menu option.
5. action-name identifies an action that can be executed by the user.
6. idle-seconds is an integer literal or variable that defines a number of seconds.
7. action-name identifies an action that can be executed by the user.
8. action-attributes are dialog-specific action attributes.

where record-input-block is:

INPUT ↓ BY NAME ↓ variable ↓ record.* } ↓,.... ↓ variable ↓ record.* } ↓,.... ↓ FROM field-list ↓ ↓ ATTRIBUTES ( input-control-attribute ↓,.... ↓ ) ↓ ↓ input-control-block ↓ ↓ END INPUT

where input-control-attribute is:

HELP = help-number
NAME = "sub-dialog-name"
WITHOUT DEFAULTS ↓ = boolean ↓

where input-control-block is one of:

BEFORE INPUT
BEFORE FIELD field-spec ↓,.... ↓
ON CHANGE field-spec ↓,.... ↓
AFTER FIELD field-spec ↓,.... ↓
AFTER INPUT
ON ACTION action-name
  → INFIELD field-spec ↓ ATTRIBUTES ( action-attributes-input ) ↓
  ↓ ON KEY ( key-name ↓,.... ↓ )↓ dialog-statement ↓,.... ↓

where action-attributes-input is:

TEXT = string
COMMENT = string
IMAGE = string
ACCELERATOR = string
DEFAULTVIEW = { YES NO AUTO }
VALIDATE = NO
CONTEXTMENU = { YES NO AUTO }

where construct-block is:

```
CONSTRUCT { BY NAME variable ON column-list
variable ON column-list FROM field-list
ATTRIBUTES ( construct-control-attribute [,..] )
construct-control-block
END CONSTRUCT
```

where construct-control-attribute is:

```
HELP = help-number
NAME = "sub-dialog-name"
```

where construct-control-block is one of:

```
BEFORE CONSTRUCT
BEFORE FIELD field-spec [,..]
ON CHANGE field-spec [,..]
AFTER FIELD field-spec [,..]
AFTER CONSTRUCT
ON ACTION action-name
  [INFIELD field-spec
  ATTRIBUTES ( action-attributes-construct )]
  ON KEY ( key-name [,..] )
dialog-statement
```

where action-attributes-construct is:

```
TEXT = string
COMMENT = string
IMAGE = string
ACCELERATOR = string
DEFAULTVIEW = { YES NO AUTO }
CONTEXTMENU = { YES NO AUTO }
```

where display-array-block is:

```
DISPLAY ARRAY array TO screen-array.*
  ATTRIBUTES ( display-array-control-attribute [,..] )
  display-array-control-block
END DISPLAY
```

where display-array-control-attribute is:

```
HELP = help-number
COUNT = row-count
KEEP CURRENT ROW = [ = boolean ]
```
DETAILACTION = action-name
DOUBLECLICK = action-name
ACCESSORYTYPE = { DETAILBUTTON | DISCLOSUREINDICATOR | CHECKMARK }
FOCUSONFIELD

where display-array-control-block is one of:

BEFORE DISPLAY
BEFORE ROW
AFTER ROW
AFTER DISPLAY
ON ACTION action-name
  [ ATTRIBUTES ( action-attributes-display-array ) ]
ON KEY ( key-name [, . . . ] )
ON FILL BUFFER
ON SELECTION CHANGE
ON SORT
ON APPEND [ ATTRIBUTES ( action-attributes-listmod-triggers ) ]
ON INSERT [ ATTRIBUTES ( action-attributes-listmod-triggers ) ]
ON UPDATE [ ATTRIBUTES ( action-attributes-listmod-triggers ) ]
ON DELETE [ ATTRIBUTES ( action-attributes-listmod-triggers ) ]
ON EXPAND ( row-index )
ON COLLAPSE ( row-index )
ON DRAG_START ( dnd-object )
ON DRAG_FINISH ( dnd-object )
ON DRAG_ENTER( dnd-object )
ON DRAG_OVER ( dnd-object )
ON DROP ( dnd-object )
dialog-statement
[ . . . ]

where action-attributes-display-array is:

TEXT = string
COMMENT = string
IMAGE = string
ACCELERATOR = string
DEFAULTVIEW = { YES | NO | AUTO }
CONTEXTMENU = { YES | NO | AUTO }
ROWBOUND
[ . . . ]

where action-attributes-listmod-triggers is:

TEXT = string
COMMENT = string
IMAGE = string
ACCELERATOR = string
DEFAULTVIEW = { YES | NO | AUTO }
CONTEXTMENU = { YES | NO | AUTO }

where input-array-block is:

INPUT ARRAY array FROM screen-array.*
  [ ATTRIBUTES ( input-array-control-attribute [, . . . ] ) ]
  [ input-array-control-block
    [...]
  ]
END INPUT
where **input-array-control-attribute** is:

```plaintext
| APPEND ROW | boolean |
| AUTO APPEND | boolean |
| COUNT | row-count |
| DELETE ROW | boolean |
| HELP | help-number |
| INSERT ROW | boolean |
| KEEP CURRENT ROW | boolean |
| MAXCOUNT | max-row-count |
| WITHOUT DEFAULTS | boolean |
```

where **input-array-control-block** is one of:

```plaintext
| BEFORE INPUT |
| BEFORE ROW |
| BEFORE FIELD |
| ON CHANGE field-spec |
| AFTER FIELD field-spec |
| ON ROW CHANGE |
| ON SORT |
| AFTER ROW |
| BEFORE DELETE |
| AFTER DELETE |
| BEFORE INSERT |
| AFTER INSERT |
| AFTER INPUT |
| ON ACTION action-name |
| [INFIELD field-spec] |
| ATTRIBUTES ( action-attributes-input-array ) |
| ON KEY ( key-name ) |
```

where **action-attributes-input-array** is:

```plaintext
| TEXT | string |
| COMMENT | string |
| IMAGE | string |
| ACCELERATOR | string |
| DEFAULTVIEW | YES | NO | AUTO |
| VALIDATE | NO |
| CONTEXTMENU | YES | NO | AUTO |
| ROWBOUND |
```

where **dialog-statement** is one of:

```plaintext
| statement |
| ACCEPT DIALOG |
| CANCEL DIALOG |
| CONTINUE DIALOG |
| EXIT DIALOG |
| NEXT FIELD |
| [CURRENT |
| NEXT |
| PREVIOUS |
| field-spec ] |
```
where *field-list* defines a list of fields with one or more of:

```
[ field-name
  table-name.*
  table-name.field-name
  screen-array[line].*
  screen-array[line].field-name
  screen-record.*
  screen-record.field-name
  [, ...]
```

where *field-spec* identifies a unique field with one of:

```
[ field-name
  table-name.field-name
  screen-array.field-name
  screen-record.field-name
  
```

where *column-list* defines a list of database columns as:

```
[ column-name
  table-name.*
  table-name.column-name
  [, ...]
```

1. *variable-definition* is a variable declaration with data type as in a regular DEFINE statement.
2. *array* is the array of records used by the DIALOG statement.
3. *help-number* is an integer that allows you to associate a help message number with the command.
4. *field-name* is the identifier of a field of the current form.
5. *option-name* is a string expression defining the label of the action and identifying the action that can be executed by the user.
6. *option-comment* is a string expression containing a description for the menu option, displayed when *option-name* is the current.
7. *column-name* is the identifier of a database column of the current form.
8. *table-name* is the identifier of a database table of the current form.
9. *variable* is a simple program variable (not a record).
10. *record* is a program record (structured variable).
11. *screen-array* is the screen array that will be used in the current form.
12. *line* is a screen array line in the form.
13. *screen-record* is the identifier of a screen record of the current form.
14. *action-name* identifies an action that can be executed by the user.
15. *seconds* is an integer literal or variable that defines a number of seconds.
16. *key-name* is a hot-key identifier (like F11 or Control-z).
17. *row-index* identifies the program variable which holds the row index corresponding to the tree node that has been expanded or collapsed.
18. *dnd-object* references a ui.DragDrop variable defined in the scope of the dialog.
19. *statement* is any instruction supported by the language.
20. *action-attributes* are dialog-specific action attributes for the action.

**Related concepts**

Syntax of the START DIALOG instruction on page 1702
Starts the instance of a declarative dialog.

Syntax of the TERMINATE DIALOG instruction on page 1703
Terminates the instance of a declarative dialog.

**Declarative dialog programming steps**

This procedure describes how to implement a declarative DIALOG block.

To implement a declarative DIALOG block:

1. Create a form specification file containing screen record(s) and/or screen array(s). The screen records and screen arrays identify the presentation elements to be used by the runtime system to display the data models (the content of program variables bound to the DIALOG blocks).

2. Create a dedicated .4gl module to implement the declarative DIALOG block.

3. With the DEFINE instruction, declare program variables (records and arrays) that will be used as data models. These will typically be defined as PRIVATE module variables. For record lists (DISPLAY ARRAY or INPUT ARRAY), the members of the program array must correspond to the elements of the screen array, by number and data types. To handle record lists, use dynamic arrays instead of static arrays.

4. Define the declarative DIALOG block in the module, to handle interaction. Define a sub-dialog with program variables to be used as data models. The sub-dialog will define how variables will be used (display or input).
   a) Inside the sub-dialog instruction, define the behavior with control blocks such as BEFORE ROW, AFTER ROW, BEFORE FIELD, and interaction blocks such as ON ACTION.

5. A declarative dialog can be used as SUBDIALOG in a procedural DIALOG block, or as parallel dialog with the START DIALOG/TERMINATE DIALOG instructions.

**Related concepts**

Using declarative dialogs on page 1655
Dialog coding concepts, configuration and code structure.

Using declarative dialogs
Dialog coding concepts, configuration and code structure.

**Structure of a declarative DIALOG block**

A declarative DIALOG instruction is made of a single sub-dialog block, with an optional DEFINE clause to declare local variables.

**Important:** Unlike procedural DIALOG blocks, declarative DIALOG blocks can only define one sub-dialog block.

The dialog instruction in the declarative DIALOG block binds program variables to form fields and defines the type of interaction that will take place for the data model (simple input, list input or query). The dialog implements individual control blocks that allows you to control the behavior of the interactive instruction. The dialog can also hold action handlers.

The declarative DIALOG block can define the following dialog types:

- A list of choices controlled by a MENU sub-dialog block.
- Simple record input with the INPUT sub-dialog block.
- Query by example input with the CONSTRUCT sub-dialog block.
- Read-only record list navigation with the DISPLAY ARRAY sub-dialog block.
- Editable record list handling with the INPUT ARRAY sub-dialog block.

**Related concepts**

Structure of a procedural DIALOG block on page 1594

*The DEFINE clause*

The DEFINE clause can be used to define program variables with a scope that is local to the declarative dialog block.

This clause must be placed before any other sub-dialog block:

```plaintext
DIALOG ()
    DEFINE checked BOOLEAN,
         tmp STRING
    INPUT BY NAME ...
```
The **DEFINE** clause is only allowed in declarative dialog blocks. Variables used locally in a procedural dialog block must be defined in the scope of the function containing the procedural dialog block.

**Related concepts**

**Variables** on page 390
Explains how to define program variables.

**The MENU sub-dialog**
The MENU sub-dialog implements a list of choices for the user by using action handlers.

**MENU implements a list of action handlers**
The following code example shows a MENU sub-dialog implementing a couple of action handlers with an **ON ACTION** clause or with a **COMMAND** clause (action views of COMMAND can get the focus):

```dialog
DIALOG ()
  MENU
    ON ACTION customer_view
      ...
    ON ACTION order_view
      ...
  END MENU
END DIALOG
```

**Control blocks in MENU**
Simple record input declared with the INPUT sub-dialog can raise the following triggers:

- **BEFORE MENU**

In the singular MENU instruction, **BEFORE MENU** and **AFTER MENU** blocks are typically used as initialization and finalization blocks. In an MENU sub-dialog of a DIALOG block, **BEFORE MENU** and **AFTER MENU** blocks will be executed each time the focus goes to (**BEFORE**) or leaves (**AFTER**) the action views (buttons) controlled by this sub-dialog.

**Related concepts**

**Dialog actions** on page 1739
Describes how to program action handling when the end user triggers an action on the front-end.

**DIALOG control blocks** on page 1607
*Dialog control blocks* are predefined dialog triggers where you can implement specific code to control the interactive instruction.

**Ring menus (MENU)** on page 1467
The MENU instruction implements a list of options the end user can choose from.

**The INPUT sub-dialog**
The INPUT sub-dialog implements single record input in fields of the current form.

**Program variable to form field binding**
Each record member variable is bound to the corresponding field of a screen record, in order to manipulate the values that the user enters in the form fields.

The **INPUT** clause can be used in two forms:

1. **INPUT BY NAME** `variable-list`
2. **INPUT variable-list FROM field-list**

The **BY NAME** clause implicitly binds the fields to the variables that have the same identifiers as the field names. The variables must be declared with the same names as the fields from which they accept input. The runtime system ignores any record name prefix when making the match. The unqualified names of the variables and of the fields must be unique and unambiguous within their respective domains. If they are not, the runtime system generates an exceptions, and sets the **STATUS** variable to a negative value.

```plaintext
DEFINE p_cust RECORD
    cust_num INTEGER,
    cust_name VARCHAR(50),
    cust_address VARCHAR(100)
END RECORD
...
DIALOG
    INPUT BY NAME p_cust.*
        BEFORE FIELD cust_name
    ...
    END INPUT
...
END DIALOG
```

The **FROM** clause explicitly binds the fields in the screen record to a list of program variables by position. The number of variables or record members must equal the number of fields listed in the **FROM** clause. Each variable must be of the same (or a compatible) data type as the corresponding screen field. When the user enters data, the runtime system checks the entered value against the data type of the variable, not the data type of the screen field.

```plaintext
DEFINE c_name VARCHAR(50)
    c_addr VARCHAR(100)
...
DIALOG
    INPUT c_name,
    c_addr
        FROM FORMONLY.field01,
        FORMONLY.field02
        BEFORE FIELD cust_name
    ...
    END INPUT
...
END DIALOG
```

### Identifying an INPUT sub-dialog

The name of an **INPUT** sub-dialog can be used to qualify **sub-dialog actions** with a prefix.

In order to identify the **INPUT** sub-dialog with a specific name, you can use the **ATTRIBUTES** clause to set the **NAME** attribute:

```plaintext
INPUT BY NAME p_cust.*
    ATTRIBUTES (NAME = "cust")
...
```

### Control blocks in INPUT

Simple record input declared with the **INPUT** sub-dialog can raise the following triggers:

- **BEFORE INPUT**
- **BEFORE FIELD**
- **ON CHANGE**
- **AFTER FIELD**
• **AFTER INPUT**

In the singular **INPUT** instruction, **BEFORE** **INPUT** and **AFTER** **INPUT** blocks are typically used as initialization and finalization blocks. In an **INPUT** sub-dialog of a **DIALOG** block, **BEFORE** **INPUT** and **AFTER** **INPUT** blocks will be executed each time the focus goes to (**BEFORE** or leaves (**AFTER**) the group of fields defined by this sub-dialog.

**Related concepts**

**INPUT ATTRIBUTES clause** on page 1602

**INPUT** specific attributes can be defined in the **ATTRIBUTE** clause of the sub-dialog header.

**The CONSTRUCT sub-dialog**

The **CONSTRUCT** sub-dialog provides database query by example feature, converting search criteria entered by the user into an SQL **WHERE** condition that can be used to execute a **SELECT** statement.

**Defining query by example fields**

The **CONSTRUCT** sub-dialog requires a character string variable to hold the **WHERE** clause, and a list of **screen fields** where the user can enter search criteria.

```plaintext
DEFINE sql_condition STRING
    ...;
DIALOG
    CONSTRUCT BY NAME sql_condition
        ON customer.cust_name, customer.cust_address
        BEFORE FIELD cust_name
    ...;
    END CONSTRUCT
    ...;
END DIALOG
```

Make sure the character string variable is large enough to store all possible SQL conditions. It is better to use a **STRING** data type to avoid any size problems.

**CONSTRUCT** uses the field data types defined in the current form file to produce the SQL conditions. This is different from other interactive instructions, where the data types of the program variables define the way to handle input/display. It is strongly recommended (but not mandatory) that the form field data types correspond to the data types of the program variables used for input. This is implicit if both form fields and program variables are based on the **database schema file**.

The **CONSTRUCT** clause can be used in two forms:

1. **CONSTRUCT BY NAME** string-variable **ON** column-list
2. **CONSTRUCT** string-variable **ON** column-list **FROM** field-list

The **BY NAME** clause implicitly binds the form fields to the columns, where the form field identifiers match the column names specified in the column-list after the **ON** keyword. You can specify the individual column names (separated by commas) or use the **tablename.*** shortcut to include all columns defined for a table in the database schema file.

The **FROM** clause explicitly binds the form fields listed after the **FROM** keyword with the column definitions listed after the **ON** keyword.

In both cases, the name of the columns in **column-list** will be used to produce the SQL condition in **string-variable**.

**Identifying a CONSTRUCT sub-dialog**

The name of a **CONSTRUCT** sub-dialog can be used to qualify **sub-dialog actions** with a prefix. In order to identify the **CONSTRUCT** sub-dialog with a specific name, use the **ATTRIBUTES** clause to set the **NAME** attribute:

```plaintext
CONSTRUCT BY NAME sql_condition ON customer.*
    ATTRIBUTES (NAME = "q_cust")
```
Control blocks in CONSTRUCT

A Query By Example declared with the CONSTRUCT clause can raise the following triggers:

- BEFORE CONSTRUCT
- BEFORE FIELD
- AFTER FIELD
- AFTER CONSTRUCT

In the singular CONSTRUCT instruction, BEFORE CONSTRUCT and AFTER CONSTRUCT blocks are typically used as initialization and finalization blocks. In DIALOG block, BEFORE CONSTRUCT and AFTER CONSTRUCT blocks will be executed each time the focus goes to (BEFORE) or leaves (AFTER) the group of fields defined by this sub-dialog.

Related concepts

Query operators in CONSTRUCT on page 1569
CONSTRUCT ATTRIBUTES clause on page 1605
CONSTRUCT specific attributes can be defined in the ATTRIBUTE clause of the sub-dialog header.

The DISPLAY ARRAY sub-dialog

The DISPLAY ARRAY sub-dialog is the controller to implement the navigation in a list of records, with option data modification actions.

Program array to screen array binding

The DISPLAY ARRAY sub-dialog binds the members of the flat record (or the primitive member) of an array to the screen-array or screen-record fields specified with the TO keyword.

The number of variables in each record of the program array must be the same as the number of fields in each screen record (that is, in a single row of the screen array).

You typically bind a program array to a screen-array in order to display a page of records. However, the DIALOG instruction can also bind the program array to a simple flat screen-record. In this case, only one record will be visible at a time.

The next code example defines an array with a flat record and binds it to a screen array:

```define p_items DYNAMIC ARRAY OF RECORD
  item_num INTEGER,
  item_name VARCHAR(50),
  item_price DECIMAL(6,2)
END RECORD
...```

```dialog
  DISPLAY ARRAY p_items TO sa.*
  BEFORE ROW
    ...
  END DISPLAY
  ...
END DIALOG```

If the screen array is defined with one field only, you can bind an array defined with a primitive type:

```define p_names DYNAMIC ARRAY OF VARCHAR(50)
...```

```dialog
  DISPLAY ARRAY p_names TO sa.*
  BEFORE DELETE
    ...
END DIALOG```
Identifying a DISPLAY ARRAY sub-dialog

The name of the screen array specified with the TO clause identifies the list. The dialog class method takes the name of the screen array as the parameter, identifying the list. For example, you would use DIALOG.getCurrentRow("screen-array") to query for the current row in the list identified by 'screen-array'. The name of the screen-array is also used to qualify sub-dialog actions with a prefix.

Control blocks in DISPLAY ARRAY

Read-only record lists declared with the DISPLAY ARRAY sub-dialog can raise the following triggers:

- BEFORE DISPLAY
- BEFORE ROW
- AFTER ROW
- AFTER DISPLAY

In the singular DISPLAY ARRAY instruction, BEFORE DISPLAY and AFTER DISPLAY blocks are typically used as initialization and finalization blocks. In a DISPLAY ARRAY sub-dialog of a DIALOG block, BEFORE DISPLAY and AFTER DISPLAY blocks will be executed each time the focus goes to (BEFORE) or leaves (AFTER) the group of fields defined by this sub-dialog.

Related concepts

DISPLAY ARRAY ATTRIBUTES clause on page 1603
DISPLAY ARRAY specific attributes can be defined in the ATTRIBUTE clause of the sub-dialog header.

The INPUT ARRAY sub-dialog

The INPUT ARRAY sub-dialog is the controller to implement the navigation and edition in a list of records.

Important: This feature is not supported on mobile platforms.

Program array to screen array binding

The INPUT ARRAY sub-dialog binds the members of the flat record (or the primitive member) of an array to the screen-array or screen-record fields specified with the FROM keyword. The number of variables in each record of the program array must be the same as the number of fields in each screen record (that is, in a single row of the screen array).

You typically bind a program array to a screen-array in order to display a page of records. However, the DIALOG instruction can also bind the program array to a simple flat screen-record. In this case, only one record will be visible at a time.

The next code example defines an array with a flat record and binds it to a screen array:

```sql
DEFINE p_items DYNAMIC ARRAY OF RECORD
    item_num INTEGER,
    item_name VARCHAR(50),
    item_price DECIMAL(6,2)
END RECORD
...
DIALOG
    INPUT ARRAY p_items FROM sa.*
    BEFORE INSERT
    ...
    END INPUT
    ...
END DIALOG
```
If the screen array is defined with one field only, you can bind an array defined with a primitive type:

```
DEFINE p_names DYNAMIC ARRAY OF VARCHAR(50)
...
DIALOG
  INPUT ARRAY p_names FROM sa.*
  BEFORE DELETE
  ...
  END INPUT
  ...
END DIALOG
```

### Identifying an INPUT ARRAY sub-dialog

The name of the screen array specified with the **FROM** clause will be used to identify the list. For example, the dialog class method such as `DIALOG.getCurrentRow("screen-array")` takes the name of the screen array as the parameter, to identify the list you want to query for the current row. The name of the screen-array is also used to qualify sub-dialog actions with a prefix.

### Control blocks in INPUT ARRAY

Editable record lists declared with the **INPUT ARRAY** sub-dialog can raise the following triggers:

- **BEFORE INPUT**
- **BEFORE ROW**
- **BEFORE FIELD**
- **ON CHANGE**
- **AFTER FIELD**
- **ON ROW CHANGE**
- **AFTER ROW**
- **BEFORE DELETE**
- **AFTER DELETE**
- **BEFORE INSERT**
- **AFTER INSERT**
- **AFTER INPUT**

In the singular **INPUT ARRAY** instruction, **BEFORE INPUT** and **AFTER INPUT** blocks are typically used as initialization and finalization blocks. In the **INPUT ARRAY** sub-dialog of a **DIALOG** block, **BEFORE INPUT** and **AFTER INPUT** blocks are executed each time the focus goes to (**BEFORE**) or leaves (**AFTER**) the group of fields defined by this sub-dialog.

### Related concepts

- **INPUT ARRAY ATTRIBUTES clause** on page 1603
- **INPUT ARRAY** specific attributes can be defined in the **ATTRIBUTE** clause of the sub-dialog header.

### Declarative DIALOG block configuration

Attributes defined in the **ATTRIBUTES** clause of dialogs can be used to configure a declarative **DIALOG** block and its sub-dialogs.

The **ATTRIBUTES** clause of dialogs overrides all default attributes and temporarily override any display attributes that the **OPTIONS** or the **OPEN WINDOW** statement specified for these fields.
**INPUT ATTRIBUTES clause**

INPUT specific attributes can be defined in the ATTRIBUTE clause of the sub-dialog header.

**HELP option**

The HELP attribute defines the number of the help message to be displayed when invoked and focus is in the list controlled by the INPUT sub-dialog. The predefined 'help' action is automatically created by the runtime system. You can bind action views to the 'help' action. The HELP clause overrides the HELP attribute.

**NAME option**

The NAME attribute can be used to identify the INPUT sub-dialog, especially useful to qualify sub-dialog actions.

**WITHOUT DEFAULTS option**

By default, sub-dialogs use the default values defined in the form files. If you want to use the values stored in the program variables bound to the dialog, you must use the WITHOUT DEFAULTS attribute. For more details see WITHOUT DEFAULTS option.

**DISPLAY ARRAY ATTRIBUTES clause**

DISPLAY ARRAY specific attributes can be defined in the ATTRIBUTE clause of the sub-dialog header.

**HELP option**

The HELP attribute defines the number of the help message to be displayed when invoked and focus is in the list controlled by the DISPLAY ARRAY sub-dialog. The predefined 'help' action is automatically created by the runtime system. You can bind action views to the 'help' action.

The HELP clause overrides the HELP attribute.

**COUNT option**

The COUNT attribute defines the number of valid rows in the static array to be displayed as default rows. If you do not use the COUNT attribute, the runtime system cannot determine how much data to display, so the screen array remains empty. The COUNT option is ignored when using a dynamic array, unless page mode is used. In this case, the COUNT attribute must be used to define the total number of rows, because the dynamic array will only hold a page of the entire row set. If the value of COUNT is negative or zero, it defines an empty list.

See also Controlling the number of rows on page 1834.

**DOUBLECLICK option**

The DOUBLECLICK option can be used to define the action that will be fired when the user chooses a row from the list. Different configuration options are available to control the row selection action of desktop and mobile devices. For more details, see Defining the action for a row choice on page 1867.

**ACCESSORYTYPE option**

Important: This feature is only for mobile platforms.

The ACCESSORYTYPE attribute can be used to define the decoration of rows, typically used on an iOS device. Values can be DETAILBUTTON, DISCLOSUREINDICATOR, CHECKMARK to respectively get an (i), > or check mark icon. For more details, see Row configuration on iOS devices on page 1876.

**DETAILACTION option**

Important: This feature is only for mobile platforms.

The DETAILACTION attribute can be used to define the action that will be fired when the user selects the detail button of a row. The detail button is typically shown with an (i) icon on iOS devices. Note that the DOUBLECLICK
attribute can be used to distinguish the action when the user selects the row instead of the detail button in the row. For more details, see Row configuration on iOS devices on page 1876.

FOCUSONFIELD option

Important: This feature is not supported on mobile platforms.

When the FOCUSONFIELD option is used, the DISPLAY ARRAY allows focus at the field (or cell) level. It then possible to implement BEFORE FIELD and AFTER FIELD blocks, as well as using NEXT FIELD instructions. However, the dialog still manages a read-only list. For more details, see Field-level focus in DISPLAY ARRAY on page 1840.

INPUT ARRAY ATTRIBUTES clause

INPUT ARRAY specific attributes can be defined in the ATTRIBUTE clause of the sub-dialog header.

HELP option

The HELP clause specifies the number of a help message to display if the user invokes the help the INPUT ARRAY dialog. The predefined 'help' action is automatically created by the runtime system. You can bind action views to the 'help' action. The HELP clause overrides the HELP attribute.

COUNT option

The COUNT attribute defines the number of valid rows in the static array to be displayed as default rows. If you do not use the COUNT attribute, the runtime system cannot determine how much data to display, so the screen array remains empty. The COUNT option is ignored when using a dynamic array. If you specify the COUNT attribute, the WITHOUT DEFAULTS option is not required because it is implicit. If the COUNT attribute is greater than MAXCOUNT, the runtime system will take MAXCOUNT as the actual number of rows. If the value of COUNT is negative or zero, it defines an empty list.

MAXCOUNT option

The MAXCOUNT attribute defines the maximum number of rows that can be inserted in the program array. This attribute allows you to give an upper limit of the total number of rows the user can enter. It can be used with static or dynamic arrays.

When binding a static array, MAXCOUNT is used as upper limit if it is lower or equal to the actual declared static array size. If MAXCOUNT is greater than the array size, the size of the static array is used as the upper limit. If MAXCOUNT is lower than the COUNT attribute (or to the SET_COUNT() parameter when using a singular INPUT ARRAY), the actual number of rows in the array will be reduced to MAXCOUNT.

When binding a dynamic array, the user can enter an infinite number of rows unless the MAXCOUNT attribute is used. If MAXCOUNT is lower than the actual size of the dynamic array, the number of rows in the array will be reduced to MAXCOUNT.

If MAXCOUNT is negative or equal to zero, the user cannot insert rows.

APPEND ROW option

The APPEND ROW attribute can be set to FALSE to avoid the append default action, and deny the user to add rows at the end of the list. If APPEND ROW = FALSE, it is still possible to insert rows in the middle of the list. Use the INSERT ROW attribute to disallow the user from inserting rows. Additionally, even with APPEND ROW=FALSE and INSERT ROW=FALSE, you can still get automatic temporary row creation if AUTO APPEND is not set to FALSE.

INSERT ROW option

The INSERT ROW attribute can be set to FALSE to avoid the insert default action, and deny the user to insert new rows in the middle of the list. However, even if INSERT ROW is FALSE, it is still possible to append rows at the end of the list. Use the APPEND ROW attribute to disallow the user from appending rows. Additionally, even with
APPEND ROW=FALSE and INSERT ROW=FALSE, you can still get automatic temporary row creation if AUTO APPEND is not set to FALSE.

**DELETE ROW option**

The DELETE ROW attribute can be set to FALSE to avoid the delete default action, and deny the user to remove rows from the list.

**AUTO APPEND option**

By default, an INPUT ARRAY controller creates a temporary row when needed (for example, when the user deletes the last row of the list, a new row will be automatically created). You can prevent this default behavior by setting the AUTO APPEND attribute to FALSE. When this attribute is set to FALSE, the only way to create a new temporary row is to execute the append action.

If both the APPEND ROW and INSERT ROW attributes are set to FALSE, the dialog automatically behaves as if AUTO APPEND equals FALSE.

**KEEP CURRENT ROW option**

Depending on the list container used in the form, the current row may be highlighted during the execution of the dialog, and cleared when the instruction ends. You can change this default behavior by using the KEEP CURRENT ROW attribute, to force the runtime system to keep the current row highlighted.

**WITHOUT DEFAULTS option**

You typically use the INPUT ARRAY sub-dialog with the WITHOUT DEFAULTS attribute. If this attribute is not set when using an INPUT ARRAY sub-dialog, the list is empty even if the array holds data. For more details see WITHOUT DEFAULTS option.

**CONSTRUCT ATTRIBUTES clause**

CONSTRUCT specific attributes can be defined in the ATTRIBUTE clause of the sub-dialog header.

**HELP option**

The HELP attribute defines the number of the help message to be displayed when invoked and focus is in the list controlled by the CONSTRUCT sub-dialog. The predefined 'help' action is automatically created by the runtime system. You can bind action views to the 'help' action.

The HELP clause overrides the HELP attribute.

**NAME option**

The NAME attribute can be used to identify the CONSTRUCT sub-dialog; this is especially useful to qualify sub-dialog actions.

**Default actions created by a DIALOG block**

Default actions ease the implementation of the controller by providing expected actions.

The runtime system creates a set of default actions based on the sub-dialogs defined in a (declarative or procedural) DIALOG block. These actions are provided to ease the implementation of the controller. For example, when using an INPUT ARRAY sub-dialog, the dialog instruction will automatically create the insert, append and delete default actions.

Table 380: Default actions created for the DIALOG block on page 1665 lists the default actions created for the DIALOG interactive instruction, for each type of sub-dialogs:
### Table 380: Default actions created for the DIALOG block

<table>
<thead>
<tr>
<th>Default action</th>
<th>Control Block execution order</th>
</tr>
</thead>
<tbody>
<tr>
<td>help</td>
<td>Shows the help topic defined by the HELP clause. Only created when a HELP clause or option is defined for the sub-dialog.</td>
</tr>
<tr>
<td>insert</td>
<td>Inserts a new row before current row. Only for INPUT ARRAY dialogs. Action creation can be avoided with INSERT ROW = FALSE attribute.</td>
</tr>
<tr>
<td>append</td>
<td>Appends a new row at the end of the list. Only for INPUT ARRAY dialogs. Action creation can be avoided with APPEND ROW = FALSE attribute.</td>
</tr>
<tr>
<td>delete</td>
<td>Deletes the current row. Only for INPUT ARRAY dialogs. Action creation can be avoided with DELETE ROW = FALSE attribute.</td>
</tr>
<tr>
<td>nextrow</td>
<td>Moves to the next row in a list displayed in one row of fields. See note (1).</td>
</tr>
<tr>
<td>prevrow</td>
<td>Moves to the previous row in a list displayed in one row of fields. See note (1).</td>
</tr>
<tr>
<td>firstrow</td>
<td>Moves to the first row in a list displayed in one row of fields. See note (1).</td>
</tr>
<tr>
<td>lastrow</td>
<td>Moves to the last row in a list displayed in one row of fields. See note (1).</td>
</tr>
<tr>
<td>find</td>
<td>Opens the fglfind dialog window to let the user enter a search value, and seeks to the row matching the value. See note (2).</td>
</tr>
<tr>
<td>findnext</td>
<td>Seeks to the next row matching the value entered during the fglfind dialog. See note (2).</td>
</tr>
</tbody>
</table>

**Notes:**

1. The action is only created with a DISPLAY ARRAY or INPUT ARRAY using a screen record bound to a set of form fields in a GRID container, and this set of fields show only a single row of the list. The action is not created when using a screen array bound to a list container such as TABLE, TREE and SCROLLGRID, or to a set of fields in a GRID container, that repeat on several lines to show more than one single row.

2. The action is only created if the context allows built-in find.

The insert, append and delete default actions can be avoided with dialog control attributes:

```plaintext
INPUT ARRAY arr TO sr.* ATTRIBUTES( INSERT ROW=FALSE, APPEND ROW=FALSE, ... )
...```
Related concepts
DISPLAY ARRAY ATTRIBUTES clause on page 1603
DISPLAY ARRAY specific attributes can be defined in the ATTRIBUTE clause of the sub-dialog header.

INPUT ARRAY ATTRIBUTES clause on page 1603
INPUT ARRAY specific attributes can be defined in the ATTRIBUTE clause of the sub-dialog header.

DIALOG data blocks
Dialog data blocks are dialog triggers invoked when the dialog controller needs data to feed the view with values.

Such blocks are typically used when record list data is provided dynamically, with the paged mode or when implementing dynamic tree-views.

ON FILL BUFFER block
The ON FILL BUFFER block is used to fill a page of rows into the dynamic array, based on an offset and a number of rows.

This data block is only used in DISPLAY ARRAY dialog blocks.

The offset can be retrieved with the FGL_DIALOG_GETBUFFERSTART() built-in function and the number of rows to provide is defined by the FGL_DIALOG_GETBUFFERLENGTH() built-in function.

The ON FILL BUFFER block is executed when the runtime system needs data rows to fill the current page of the list dialog. This can happen before a BEFORE DISPLAY of a singular DISPLAY ARRAY, or before the BEFORE DIALOG block of a DIALOG / END DIALOG instruction containing DISPLAY ARRAY sub-dialogs.

For more details about ON FILL BUFFER usage, see Paged mode of DISPLAY ARRAY on page 1843.

ON EXPAND block
The ON EXPAND block is executed when a tree view node is expanded (opened).

This data block is used to implement dynamic trees in a DISPLAY ARRAY, where nodes are added, depending on the nodes opened by the end user.

For more details, see Dynamic filling of very large trees on page 1896.

Related concepts
ON COLLAPSE block on page 1508

ON COLLAPSE block
The ON COLLAPSE block is executed when a tree view node is collapsed (i.e. closed).

This data block is used to implement dynamic trees in a DISPLAY ARRAY, where nodes are removed from view by the user closing or collapsing them.

For more details, see Dynamic filling of very large trees on page 1896.

Related concepts
ON EXPAND block on page 1508

DIALOG control blocks
Dialog control blocks are predefined dialog triggers where you can implement specific code to control the interactive instruction.

The code may involve using ui.Dialog class methods or dialog specific instructions such as NEXT FIELD or CONTINUE DIALOG.

Control block execution order with declarative dialogs
Note: Depending on the usage context, dialog-starting and ending control blocks such as BEFORE INPUT / AFTER INPUT are executed each time the focus goes to an element controlled by the dialog, or only once, when the dialog is started/ended. See control block specific topics for more details.
Control block execution order when in multiple dialogs

When the declarative dialog is used with a SUBDIALOG keyword in the context of a procedural DIALOG block, the control blocks are executed in the same order as if the declarative dialog was included in the procedural DIALOG block.

For more details, see Control block execution order in multiple dialogs on page 1607.

Control block execution order in parallel dialogs

The order in which control blocks are executed in a declarative DIALOG used as parallel dialog is the same as when executing a singular dialog.

According to the type of dialog defined in the declarative DIALOG, see:

- INPUT control blocks execution order on page 1488
- DISPLAY ARRAY control blocks execution order on page 1508
- CONSTRUCT control blocks execution order on page 1573
- INPUT ARRAY control blocks execution order on page 1540

BEFORE FIELD block

In dialog instructions INPUT, INPUT ARRAY, CONSTRUCT or in a DISPLAY ARRAY using the FOCUSONFIELD attribute, the BEFORE FIELD block is executed every time the specified field gets the focus.

For single record inputs driven by INPUT or query by example (QBEs) driven by CONSTRUCT, the BEFORE FIELD block is executed when moving the focus from field to field.

For editable lists driven by INPUT ARRAY, the BEFORE FIELD block is executed when moving the focus from field to field in the same row, or when moving to another row in the same column.

For record lists driven by DISPLAY ARRAY using the FOCUSONFIELD attribute, the BEFORE FIELD block is executed when moving the focus from field to field. However, the fields will not be editable as in an INPUT ARRAY.

Important: The BEFORE FIELD block is also executed when performing a NEXT FIELD instruction.

The BEFORE FIELD keywords must be followed by a list of form field specification. The screen-record name can be omitted.

BEFORE FIELD is executed after BEFORE INPUT, BEFORE CONSTRUCT, BEFORE ROW and BEFORE INSERT.

Use this block to do some field value initialization, or to display a message to the user:

```
INPUT BY NAME p_cust.* ...
BEFORE FIELD cust_status
  LET p_cust.cust_comment = NULL
  MESSAGE "Enter customer status"
```

When using the default FIELD ORDER CONSTRAINT mode, the dialog executes the BEFORE FIELD block of the field corresponding to the first variable of an INPUT or INPUT ARRAY, even if that field is not editable (NOENTRY, hidden or disabled). The block is executed when you enter the dialog and every time you create a new row in the case of INPUT ARRAY. This behavior is supported for backward compatibility. The block is not executed when using the FIELD ORDER FORM, the mode recommended for DIALOG instructions.

With the FIELD ORDER FORM mode, for each dialog executing for the first time with a specific form, the BEFORE FIELD block will be invoked for the first field of the initial tabbing list defined by the form, even if that field was hidden or moved around in a table. The dialog then behaves as if a NEXT FIELD first-visible-column had been executed in the BEFORE FIELD of that field.

When form-level validation occurs and a field contains an invalid value, the dialog gives the focus to the field, but no BEFORE FIELD trigger will be executed.
Related concepts
Form-level validation rules on page 1727
Form-level validation rules can be defined for each field controlled by a dialog.

AFTER FIELD block on page 1492

AFTER FIELD block
In dialog instructions INPUT, INPUT ARRAY, CONSTRUCT or in a DISPLAY ARRAY using the FOCUSONFIELD attribute, the AFTER FIELD block is executed every time the focus leaves the specified field.

For single record inputs driven by INPUT or query by example (QBEs) driven by CONSTRUCT, the AFTER FIELD block is executed when moving the focus from field to field.

For editable lists driven by INPUT ARRAY, the AFTER FIELD block is executed when moving the focus from field to field in the same row, or when moving to another row in the same column.

For record lists driven by DISPLAY ARRAY using the FOCUSONFIELD attribute, the AFTER FIELD block is executed when moving the focus from field to field. However, the fields will not be editable as in an INPUT ARRAY.

The AFTER FIELD keywords must be followed by a list of form field specifications. The screen-record name can be omitted.

AFTER FIELD is executed before AFTER INSERT, ON ROW CHANGE, AFTER ROW, AFTER INPUT or AFTER CONSTRUCT.

When a NEXT FIELD instruction is executed in an AFTER FIELD block, the cursor moves to the specified field, which can be the current field. This can be used to prevent the user from moving to another field / row during data input. Note that the BEFORE FIELD block is also executed when NEXT FIELD is invoked.

The AFTER FIELD block of the current field is not executed when performing a NEXT FIELD; only BEFORE INPUT, BEFORE CONSTRUCT, BEFORE ROW, and BEFORE FIELD of the target item might be executed, depending on the sub-dialog type.

When ACCEPT DIALOG, ACCEPT INPUT, or ACCEPT CONSTRUCT is performed, the AFTER FIELD trigger of the current field is executed.

Use the AFTER FIELD block to implement field validation rules:

```
INPUT BY NAME p_item.* ...
AFTER FIELD item_quantity
  IF p_item.item_quantity <= 0 THEN
    ERROR "Item quantity cannot be negative or zero"
    LET p_item.item_quantity = 0
  NEXT_FIELD item_quantity
END IF
```

Related concepts
ACCEPT DIALOG instruction on page 1640
ON CHANGE block on page 1492

ON CHANGE block
The ON CHANGE block can be used to detect when a field has been changed by user input. The ON CHANGE block is executed, if the value has changed since the field got the focus and the modification flag is set.

The ON CHANGE block can be used in INPUT, INPUT ARRAY and CONSTRUCT dialogs.

For editable fields defined as EDIT, TEXTEDIT or BUTTONEDIT, the ON CHANGE block is executed when leaving a field, if the value of the specified field has changed since the field got the focus and the modification flag is set for the field. The field is left when user validates the dialog, when moving to another field, or when moving to another row in an INPUT ARRAY. However, if the text edit field is defined with the COMPLETER attribute to enable autocompletion, the ON CHANGE trigger will be fired after a short period of time, when the user has typed characters in.
For editable fields defined as CHECKBOX, COMBOBOX, DATEEDIT, DATETIMEEDIT, TIMEEDIT, RADIOGROUP, SPINEDIT, SLIDER or URL-based WEBCOMPONENT (when the COMPONENTTYPE attribute is not used), the ON CHANGE block is invoked immediately when the user changes the value with the widget edition feature. For example, when toggling the state of a CHECKBOX, when selecting an item in a COMBOBOX list, or when choosing a date in the calendar of a DATEEDIT. Note that for such item types, when ON CHANGE is fired, the modification flag is always set.

```
ON CHANGE order_checked -- Defined as CHECKBOX
CALL setup_dialog(DIALOG)
```

Note: If both an ON CHANGE block and AFTER FIELD block are defined for a field, the ON CHANGE block is executed before the AFTER FIELD block.

When changing by program the value of the current field in an ON ACTION block, the ON CHANGE block will be executed when leaving the field, if the value is different from the previous value and the modification flag is set (after previous user input or when the touched flag has been changed by program).

In an INPUT or INPUT ARRAY, the field value change is related to the value of the variable bound to the field. In a CONSTRUCT dialog, the field value change is related to the input buffer / displayed value.

Note: With a NEXT FIELD instruction, the state of the field value change is reset, as if the user had left and reentered the field. When using NEXT FIELD in an ON CHANGE block or in an ON ACTION block, the ON CHANGE block will only be re-executed, if the value is changed since NEXT FIELD, and the modification flag is set. Therefore, ON CHANGE should not be used for field validation with a NEXT FIELD, because ON CHANGE will not get triggered again, if the (invalid) value has not changed; Field validation rules must be implemented in AFTER FIELD blocks and/or AFTER INPUT blocks.

Related concepts
Form item types on page 1266
The form item types defines the purpose of form elements.
AFTER FIELD block on page 1492

BEFORE INPUT block

BEFORE INPUT block in singular and parallel INPUT, INPUT ARRAY dialogs
In a singular INPUT, INPUT ARRAY instruction, or when used as parallel dialog, the BEFORE INPUT is only executed once when the dialog is started.

The BEFORE INPUT block is executed once at dialog startup, before the runtime system gives control to the user. This block can be used to display messages to the user, initialize program variables and set up the dialog instance by deactivating unused fields or actions the user is not allowed to execute.

```
INPUT BY NAME cust_rec.* ...
BEFORE INPUT
  MESSAGE "Input customer information"
  CALL DIALOG.setActionActive("check_info", is_super_user() )
  CALL DIALOG.setFieldActive("cust_comment", is_super_user() )
...
```

The fields are initialized with the defaults values before the BEFORE INPUT block is executed. When the INPUT instruction uses the WITHOUT DEFAULTS option, the default values are taken from the program variables bound to the fields, otherwise (with defaults), the DEFAULT attributes of the form fields are used.

Use the NEXT FIELD control instruction in the BEFORE INPUT block, to jump to a specific field when the dialog starts.
BEFORE INPUT block in INPUT and INPUT ARRAY of procedural DIALOG

In an INPUT or INPUT ARRAY sub-dialog of a procedural DIALOG instruction, the BEFORE INPUT block is executed when the focus goes to a group of fields driven by the sub-dialog. This trigger is only invoked if a field of the sub-dialog gets the focus, and none of the other fields had the focus.

When the focus is in a list driven by an INPUT ARRAY sub-dialog, moving to a different row will not invoke the BEFORE INPUT block.

BEFORE INPUT is executed after the BEFORE DIALOG block and before the BEFORE ROW, BEFORE FIELD blocks.

In this example, the BEFORE INPUT block is used to set up a specific action and display a message:

```
INPUT BY NAME p_order.*
BEFORE INPUT
    CALL DIALOG.setActionActive("validate_order", TRUE)
```

Related concepts
BEFORE CONSTRUCT block on page 1573
BEFORE DISPLAY block on page 1509
AFTER INPUT block on page 1490

AFTER INPUT block

AFTER INPUT block in singular and parallel INPUT, INPUT ARRAY dialogs

In a singular INPUT, INPUT ARRAY instruction, or when used as parallel dialog, the AFTER INPUT is only executed once when dialog ends.

The AFTER INPUT block is executed after the user has validated or canceled the INPUT or INPUT ARRAY dialog with the accept or cancel default actions, or when the ACCEPT INPUT instruction is executed.

The AFTER INPUT block is not executed when the EXIT INPUT instruction is performed.

In singular and parallel dialogs, this block is typically used to implement global dialog validation rules for several fields. If the values entered by the user do not satisfy the constraints, use the NEXT FIELD instruction to force the dialog to continue. The CONTINUE INPUT instruction can be used instead of NEXT FIELD, when no particular field has to be selected.

Before checking the validation rules, make sure that the INT_FLAG variable is FALSE: because if the user cancels the dialog, the validation rules must be skipped.

```
INPUT BY NAME cust_rec.*
    WITHOUT DEFAULTS ATTRIBUTES ( UNBUFFERED )
    ...

AFTER INPUT
    IF NOT INT_FLAG THEN
        IF cust_rec.cust_address IS NOT NULL
            AND cust_rec.cust_zipcode IS NULL THEN
                ERROR "Address is incomplete, enter a zipcode."
            NEXT FIELD zipcode
        END IF
    END IF
END INPUT
```

To limit the validation to fields that have been modified by the end user, you can call the FIELD_TOUCHED() function or the DIALOG.getFieldTouched() method to check if a field has changed during the dialog execution. This will make your validation code faster if the user has only modified a couple of fields in a large form.
**AFTER INPUT block in INPUT and INPUT ARRAY of procedural DIALOG**

In an INPUT or INPUT ARRAY sub-dialog of a procedural DIALOG instruction, the AFTER INPUT block is executed when the focus is lost by a group of fields driven by an INPUT or INPUT ARRAY sub-dialog. This trigger is invoked if a field of the sub-dialog loses the focus, and a field of a different sub-dialog gets the focus. When the focus is in a list driven by an INPUT ARRAY sub-dialog, moving to a different row will not invoke the AFTER INPUT block.

If the focus leaves the current group and goes to an action view, this trigger is not executed, because the focus has not gone to another sub-dialog yet.

AFTER INPUT is executed after the AFTER FIELD, AFTER ROW blocks and before the AFTER DIALOG block.

Executing a NEXT FIELD in the AFTER INPUT control block will keep the focus in the group of fields. Within an INPUT ARRAY sub-dialog, NEXT FIELD will keep the focus in the list and stay in the current row. You typically use this behavior to control user input.

In this example, the AFTER INPUT block is used to validate data and disable an action that can only be used in the current group:

```plaintext
INPUT BY NAME p_order.*
AFTER INPUT
  IF NOT check_order_data(DIALOG) THEN
    NEXT FIELD CURRENT
  END IF
  CALL DIALOG.setFieldActive("validate_order", FALSE)
```

**Related concepts**

- AFTER DISPLAY block on page 1510
- AFTER CONSTRUCT block on page 1574
- BEFORE INPUT block on page 1489

**BEFORE CONSTRUCT block**

**BEFORE CONSTRUCT block in singular and parallel CONSTRUCT dialogs**

In a singular CONSTRUCT instruction, or when used as parallel dialog, the BEFORE CONSTRUCT is only executed once when dialog is started.

The BEFORE CONSTRUCT block is executed once at dialog start-up, before the runtime system gives control to the user for criteria input. This block can be used to display messages to the user, initialize form fields with default search criteria values, and setup the dialog instance by deactivating unused fields or actions the user is not allowed to execute.

```plaintext
CONSTRUCT BY NAME where_part ON ...
BEFORE CONSTRUCT
  MESSAGE "Enter customer search filter"
  CALL DIALOG.setActionActive("clean", FALSE )
...
```

The fields are cleared before the BEFORE CONSTRUCT block is executed.

You can use the NEXT FIELD control instruction in the BEFORE CONSTRUCT block, to jump to a specific field when the dialog starts.

**BEFORE CONSTRUCT block in CONSTRUCT of procedural DIALOG**

In a CONSTRUCT sub-dialog of a procedural DIALOG instruction, the BEFORE CONSTRUCT block is executed when the focus goes to a group of fields driven by a CONSTRUCT sub-dialog. This trigger is only invoked if a field of the sub-dialog gets the focus, and none of the other fields had the focus.

BEFORE CONSTRUCT is executed after the BEFORE DIALOG block and before the BEFORE FIELD blocks.
In this example, the BEFORE CONSTRUCT block is used to display a message:

```
CONSTRUCT BY NAME sql ON customer.*
    BEFORE CONSTRUCT
        MESSAGE "Enter customer search filter"
```

Related concepts
BEFORE INPUT block on page 1489
BEFORE DISPLAY block on page 1509
AFTER CONSTRUCT block on page 1574

AFTER CONSTRUCT block

AFTER CONSTRUCT block in singular and parallel CONSTRUCT dialogs

In a singular CONSTRUCT instruction, or when used as parallel dialog, the AFTER CONSTRUCT is only executed once when dialog is ended.

Use an AFTER CONSTRUCT block to execute instructions after the user has finished search criteria input.

AFTER CONSTRUCT is not executed if an EXIT CONSTRUCT is performed.

The code in AFTER CONSTRUCT can for example check if a criteria combination of different fields is required or denied, and force the end use to enter all

Before checking the content of the fields used in the CONSTRUCT, make sure that the INT_FLAG variable is FALSE.

In the case that the user cancels the dialog, the validation rules must be skipped.

Since no program variables are associated with the form fields, you must query the input buffers of the fields to get the values entered by the user.

```
CONSTRUCT BY NAME where_part ON ...
    ...
    AFTER CONSTRUCT
        IF NOT INT_FLAG THEN
            IF length(DIALOG.getFieldBuffer(cust_name))==0
                OR length(DIALOG.getFieldBuffer(cust_addr))==0 THEN
                ERROR "Enter a search criteria for customer name and address fields."
                NEXT FIELD CURRENT
            END IF
        END IF
END CONSTRUCT
```

To limit the validation to fields that have been modified by the end user, you can call the FIELD_TOUCHED() function or the DIALOG.getFieldTouched() method to check if a field has changed during the dialog execution. This makes your validation code execute faster if the user has only modified a couple of fields in a large form.

AFTER CONSTRUCT block in CONSTRUCT of procedural DIALOG

In a CONSTRUCT sub-dialog of a procedural DIALOG instruction, the AFTER CONSTRUCT block is executed when the focus is lost by a group of fields driven by a CONSTRUCT sub-dialog. This trigger is invoked if a field of the sub-dialog loses the focus, and a field of a different sub-dialog gets the focus.

If the focus leaves the current group and goes to an action view, this trigger is not executed, because the focus did not go to another sub-dialog yet.

AFTER CONSTRUCT is executed after the AFTER FIELD and before the AFTER DIALOG block.

Executing a NEXT FIELD in the AFTER CONSTRUCT control block will keep the focus in the group of fields.
In this example, the AFTER CONSTRUCT block is used to build the SELECT statement:

```sql
CONSTRUCT BY NAME sql ON customer.*
   AFTER CONSTRUCT
       LET sql = "SELECT * FROM customers WHERE " || sql
```

**Related concepts**

AFTER DISPLAY block on page 1510  
AFTER INPUT block on page 1490  
BEFORE CONSTRUCT block on page 1573

**BEFORE DISPLAY block**

**BEFORE DISPLAY block in singular and parallel DISPLAY ARRAY dialogs**

In a singular DISPLAY ARRAY instruction, or when used as parallel dialog, the BEFORE DISPLAY is only executed once when the dialog is started.

The BEFORE DISPLAY block is executed once at dialog startup, before the runtime system gives control to the user. This block can be used to display messages to the user, initialize program variables, and set up the dialog instance by deactivating actions the user is not allowed to execute.

```pascal
DISPLAY ARRAY p_items TO s_items.*
   BEFORE DISPLAY
       CALL DIALOG.setActionActive("clear_item_list", is_super_user())
```

**BEFORE DISPLAY block DISPLAY ARRAY of procedural DIALOG**

In a DISPLAY ARRAY sub-dialog of a procedural DIALOG instruction, the BEFORE DISPLAY block is executed when a DISPLAY ARRAY list gets the focus.

BEFORE DISPLAY is executed before the BEFORE ROW block.

In this example the BEFORE DISPLAY block enables an action and displays a message:

```pascal
DISPLAY ARRAY p_items TO s_items.*
   BEFORE DISPLAY
       CALL DIALOG.setActionActive("print_list", TRUE)
       MESSAGE "You are now in the list of items"
```

**Related concepts**

BEFORE INPUT block on page 1489  
BEFORE CONSTRUCT block on page 1573  
AFTER DISPLAY block on page 1510

**AFTER DISPLAY block**

**AFTER DISPLAY block in singular and parallel DISPLAY ARRAY dialogs**

In a singular DISPLAY ARRAY instruction, or when used as parallel dialog, the AFTER DISPLAY is only executed once when the dialog is ended.

You typically implement dialog finalization in this block.

```pascal
DISPLAY ARRAY p_items TO s_items.*
   AFTER DISPLAY
       DISPLAY "Current row is: ", arr_curr()
```
**AFTER DISPLAY block in DISPLAY ARRAY of procedural DIALOG**

In a DISPLAY ARRAY sub-dialog of a procedural DIALOG instruction, the AFTER DISPLAY block is executed when a DISPLAY ARRAY list loses the focus and the focus goes to another sub-dialog.

If the focus leaves the current group and goes to an action view, this trigger is not executed, because the focus has not gone to another sub-dialog yet.

AFTER DISPLAY is executed after the AFTER ROW block.

In this example, the AFTER DISPLAY block disables an action that is specific to the current list:

```plaintext
DISPLAY ARRAY p_items TO s_items.*
       AFTER DISPLAY
           CALL DIALOG.setActionActive("clear_item_list", FALSE)
```

**Related concepts**

- AFTER INPUT block on page 1490
- AFTER CONSTRUCT block on page 1574
- BEFORE DISPLAY block on page 1509

**BEFORE ROW block**

**BEFORE ROW block in singular and parallel DISPLAY ARRAY, INPUT ARRAY dialogs**

In a singular DISPLAY ARRAY, INPUT ARRAY instruction, or when used as parallel dialog, the BEFORE ROW block is executed each time the user moves to another row. This trigger can also be executed in other situations, such as when you delete a row, or when the user tries to insert a row but the maximum number of rows in the list is reached.

You typically do some dialog setup / message display in the BEFORE ROW block, because it indicates that the user selected a new row or entered in the list.

When the dialog starts, BEFORE ROW will be executed for the current row, but only if there are data rows in the array.

When called in this block, `DIALOG.getCurrentRow()` / `arrCurr()` return the index of the current row.

In this example, the BEFORE ROW block gets the new row number and displays it in a message:

```plaintext
DISPLAY ARRAY ...
    ...
    BEFORE ROW
        MESSAGE "We are on row # ", arrCurr()
    ...
```

**BEFORE ROW block in DISPLAY ARRAY and INPUT ARRAY of procedural DIALOG**

In an INPUT or INPUT ARRAY sub-dialog of a procedural DIALOG instruction, the BEFORE ROW block is executed when a DISPLAY ARRAY or INPUT ARRAY list gets the focus, or when the user moves to another row inside a list. This trigger can also be executed in other situations, for example when you delete a row, or when the user tries to insert a row but the maximum number of rows in the list is reached.

You typically do some dialog setup / message display in the BEFORE ROW block, because it indicates that the user selected a new row. Do not use this trigger to detect focus changes; Use the BEFORE DISPLAY or BEFORE INPUT blocks instead.

In DISPLAY ARRAY, BEFORE ROW is executed after the BEFORE DISPLAY block. In INPUT ARRAY, BEFORE ROW is executed before the BEFORE INSERT and BEFORE FIELD blocks and after the BEFORE INPUT blocks.

When the procedural dialog starts, BEFORE ROW will only be executed if the list has received the focus and there is a current row (the array is not empty). If you have other elements in the form which can get the focus before the list,
BEFORE ROW will not be triggered when the dialog starts. You must pay attention to this, because this behavior is different to the behavior of singular DISPLAY ARRAY or INPUT ARRAY. In singular dialogs, the BEFORE ROW block is always executed when the dialog starts (and when there are rows in the array).

When called in this block, DIALOG.getCurrentRow() / arr_curr() return the index of the current row.

In this example the BEFORE ROW block displays a message with the current row number:

```plaintext
DISPLAY ARRAY p_items TO s_items.*
BEFORE ROW
  MESSAGE "We are in items, on row ", DIALOG.getCurrentRow("s_items")
```

**Related concepts**

BEFORE INPUT block on page 1489
BEFORE DISPLAY block on page 1509

**ON ROW CHANGE block**

The ON ROW CHANGE block is executed in a list controlled by an INPUT ARRAY, when leaving the current row and when the row has been modified since it got the focus. This is typically used to detect row modification.

The code in ON ROW CHANGE will not be executed when leaving new rows created by the user with the default append or insert action. To detect row creation, you must use the BEFORE INSERT or AFTER INSERT control blocks.

The ON ROW CHANGE block is only executed if at least one field value in the current row has changed since the row was entered, and the modification flag of the field is set. The modified field(s) may not be the current field, and several field values can be changed. Values may have been changed by the user or by the program. The modification flag is reset for all fields when entering another row, when going to another sub-dialog, or when leaving the dialog instruction.

ON ROW CHANGE is executed after the AFTER FIELD block and before the AFTER ROW block.

When called in this block, DIALOG.getCurrentRow() / arr_curr() return the index of the current row that has been changed.

You can, for example, code database modifications (UPDATE) in the ON ROW CHANGE block:

```plaintext
INPUT ARRAY p_items FROM s_items.*

ON ROW CHANGE
  LET r = DIALOG.getCurrentRow("s_items")
  UPDATE items SET
    items.item_code       = p_items[r].item_code,
    items.item_description = p_items[r].item_description,
    items.item_price      = p_items[r].item_price,
    items.item_updatedate = TODAY
  WHERE items.item_num   = p_items[r].item_num
```

**Related concepts**

Input field modification flag on page 1723
Each input field controlled by a dialog instruction has a modification flag.

AFTER ROW block on page 1511

**AFTER ROW block**

**AFTER ROW block in singular and parallel DISPLAY ARRAY, INPUT ARRAY dialogs**

In a singular DISPLAY ARRAY, INPUT ARRAY instruction, or when used as parallel dialog, the AFTER ROW block is executed each time the user moves to another row, before the current row is left. This trigger can also be executed in other situations, such as when you delete a row, or when the user inserts a new row.
A NEXT FIELD instruction executed in the AFTER ROW control block will keep the user entry in the current row. Use this behavior to implement row validation and prevent the user from leaving the list or moving to another row.

When called in this block, DIALOG.getCurrentRow() / arr_curr() returns the index of the row that you are leaving.

**AFTER ROW block in DISPLAY ARRAY and INPUT ARRAY of procedural DIALOG**

In an INPUT or INPUT ARRAY sub-dialog of a procedural DIALOG instruction, the AFTER ROW block is executed when a DISPLAY ARRAY or INPUT ARRAY list loses the focus, or when the user moves to another row in a list. This trigger can also be executed in other situations, for example when you delete a row, or when the user inserts a new row.

AFTER ROW is executed after the AFTER FIELD, AFTER INSERT and before AFTER DISPLAY or AFTER INPUT blocks.

When called in this block, DIALOG.getCurrentRow() / arr_curr() returns the index of the of the row that you are leaving.

For both INPUT ARRAY and DISPLAY ARRAY sub-dialogs, a NEXT FIELD executed in the AFTER ROW control block will keep the focus in the list and stay in the current row. Use this feature to implement row validation and prevent the user from leaving the list or moving to another row.

**AFTER ROW and temporary rows in INPUT ARRAY**

Important: After creating a temporary row at the end of a list driven by INPUT ARRAY, if you leave that row to go to a previous row without data input (setting the touched flag), or when the cancel action is invoked, the temporary row will be automatically removed. The AFTER ROW block will be executed for the temporary row, but ui.Dialog.getCurrentRow() / arr_curr() will be one row greater than ui.Dialog.getArrayLength() / ARR_COUNT(). In this case, it is recommended that you ignore the AFTER ROW event. For example, it is recommended that you avoid executing a NEXT FIELD or CONTINUE INPUT instruction, and trying to access the dynamic array with a row index that is greater than the total number of rows, otherwise the runtime system will adapt the total number of rows to the actual number of rows in the program array.

In this example, the AFTER ROW block checks the current row index and verifies a variable value to force the focus to stay in the current row if the value is wrong:

```plaintext
INPUT ARRAY p_items FROM s_items.*
...
AFTER ROW
  LET r = DIALOG.getCurrentRow("s_items")
  IF r <= DIALOG.getArrayLength("s_items") THEN
    IF NOT item_is_valid_quantity(p_item[r].item_quantity) THEN
      ERROR "Item quantity is not valid"
      NEXT FIELD item_quantity
    END IF
  END IF
END IF
```

Another way to handle the case of temporary rows in AFTER ROW is to use a flag to know if the AFTER INSERT block was executed. The AFTER INSERT block is not executed if the temporary row is automatically removed. By setting a first value in BEFORE INSERT and changing the flag in AFTER INSERT, you can detect if the row was permanently added to the list:

```plaintext
INPUT ARRAY p_items FROM s_items.*
...
BEFORE INSERT
  LET op = "T"
...
AFTER INSERT
  LET op = "I"
```
... AFTER ROW
  IF op == "I" THEN
    IF NOT item_is_valid_quantity(p_item[arr_curr()].item_quantity) THEN
      ERROR "Item quantity is not valid"
      NEXT FIELD item_quantity
      END IF
    WHENEVER ERROR CONTINUE
    INSERT INTO items (item_num, item_name, item_quantity) VALUES ( p_item[arr_curr()].*)
    WHENEVER ERROR STOP
    IF SQLCA.SQLCODE<0 THEN
      ERROR "Could not insert the record into database!"
      NEXT FIELD CURRENT
    ELSE
      MESSAGE "Record has been inserted successfully"
      END IF
  END IF
...  

Related concepts
NEXT FIELD instruction on page 1497
BEFORE ROW block on page 1510
ON ROW CHANGE block on page 1544

BEFORE INSERT block

The BEFORE INSERT block is executed when a new row is created in an INPUT ARRAY. You typically use this trigger to set some default values in the newly-created row. A new row can be created by moving down after the last row, by executing an insert action, or by executing an append action.

The BEFORE INSERT block is executed after the BEFORE ROW block and before the BEFORE FIELD block.

When called in this block, DIALOG.getCurrentRow("screen-array")/arr_curr() returns the index of the newly-created row.

To distinguish row insertion from an appended row, compare the current row (DIALOG.getCurrentRow("screen-array")) with the total number of rows (DIALOG.getArrayLength("screen-array")). If the current row index and the total number of rows correspond, the BEFORE INSERT concerns a temporary row, otherwise it concerns an inserted row.

Row creation can be stopped by using the CANCEL INSERT instruction inside BEFORE INSERT. If possible, it is however better to disable the insert and append actions to prevent the user executing the actions with DIALOG.setActionActive().

In this example, the BEFORE INSERT block checks if the user can create rows and prevents new row creation if needed; otherwise, it sets some default values:

INPUT ARRAY p_items FROM s_items.*
...  
  BEFORE INSERT
    IF NOT user_can_append THEN
      ERROR "You are not allowed to append rows"
      CANCEL INSERT
    END IF
    LET r = DIALOG.getCurrentRow("s_items")
    LET p_items[r].item_num = get_new_serial("items")
    LET p_items[r].item_name = "undefined"

Related concepts
Appending rows in INPUT ARRAY on page 1848
Rows appended at the end of an editable list are temporary until they are edited.

**BEFORE ROW block** on page 1510

**AFTER INSERT block**

The **AFTER INSERT** block of **INPUT ARRAY** is executed when the creation of a new row is validated. In this block, you can for example implement SQL to insert a new row in the database table.

The **AFTER INSERT** block is executed after the **AFTER FIELD** block and before the **AFTER ROW** block.

When called in this block, `DIALOG.getCurrentRow() / arr_curr()` returns the index of the newly-created row.

When the user appends a new row at the end of the list, then moves UP to another row or validates the dialog, the **AFTER INSERT** block is only executed if at least one field was edited. If no data entry is detected, the dialog automatically removes the new appended row and thus does not trigger the **AFTER INSERT** block.

When executing a **NEXT FIELD** in the **AFTER INSERT** block, the dialog will keep the focus in the list and stay in the current row. Use this behavior to implement row input validation and prevent the user from leaving the list or moving to another row. However, this will not cancel the row insertion and will not invoke the **BEFORE INSERT**/ **AFTER INSERT** triggers again. The only way to keep the focus in the current row after the row was inserted is to execute a **NEXT FIELD** in the **AFTER ROW** block.

In this example, the **AFTER INSERT** block inserts a new row in the database and cancels the operation if the SQL command fails:

```
INPUT ARRAY p_items FROM s_items.*

...  
AFTER INSERT  
WHENEVER ERROR CONTINUE  
INSERT INTO items VALUES  
( p_items[ DIALOG.getCurrentRow("s_items") ].* )  
WHENEVER ERROR STOP  
IF SQLCA.SQLCODE<>0 THEN  
ERROR SQLERRMESSAGE  
CANCEL INSERT  
END IF
```

**Related concepts**

- **NEXT FIELD instruction** on page 1497
- **AFTER ROW block** on page 1511

**BEFORE DELETE block**

The **BEFORE DELETE** block is executed each time the user deletes a row of an **INPUT ARRAY** list, before the row is removed from the list.

You typically code the database table synchronization in the **BEFORE DELETE** block, by executing a DELETE SQL statement using the primary key of the current row. In the **BEFORE DELETE** block, the row to be deleted still exists in the program array, so you can access its data to identify what record needs to be removed.

The **BEFORE DELETE** block is executed before the **AFTER DELETE** block.

If needed, the deletion can be canceled with the **CANCEL DELETE** instruction.

When called in this block, `DIALOG.getCurrentRow() / arr curr()` returns the index of the row that will be deleted.

This example uses the **BEFORE DELETE** block to remove the row from the database table and cancels the deletion operation if an SQL error occurs:

```
INPUT ARRAY p_items FROM s_items.*
BEFORE DELETE
```
\begin{verbatim}
LET r = DIALOG.getCurrentRow("s_items")
WHENEVER ERROR CONTINUE
DELETE FROM items
    WHERE item_num = p_items[r].item_num
WHENEVER ERROR STOP
IF SQLCA.SQLCODE<>0 VALUES
    ERROR SQLERRMESSAGE
    CANCEL DELETE
END IF
...
\end{verbatim}

Related concepts

**AFTER DELETE block** on page 1548

**AFTER DELETE block**

The **AFTER DELETE** block is executed each time the user deletes a row of an INPUT ARRAY list, after the row has been deleted from the list.

The **AFTER DELETE** block is executed after the **BEFORE DELETE** block and before the **AFTER ROW** block for the deleted row and the **BEFORE ROW** block of the new current row.

When an **AFTER DELETE** block executes, the program array has already been modified; the deleted row no longer exists in the array (except in the special case when deleting the last row). The **arr_curr()** function or the **ui.Dialog.getCurrentRow()** method returns the same index as in **BEFORE ROW**, but it is the index of the new current row. The **AFTER ROW** block is also executed just after the **AFTER DELETE** block.

**Important:** When deleting the last row of the list, **AFTER DELETE** is executed for the delete row, and **DIALOG.getCurrentRow() / arr_curr()** will be one greater than **DIALOG.getArrayLength() / ARR_COUNT()**. Ensure you avoid accessing a dynamic array with a row index that is greater than the total number of rows, otherwise the runtime system will adapt the total number of rows to the actual number of rows in the program array. When using a static array, you must ignore the values in the rows after **ARR_COUNT()**.

Here the **AFTER DELETE** block is used to re-number the rows with a new item line number (note that **DIALOG.getArrayLength() / ARR_COUNT()** may return zero):

\begin{verbatim}
INPUT ARRAY p_items FROM s_items.*
    AFTER DELETE
        LET r = DIALOG.getCurrentRow("s_items")
        FOR i=r TO DIALOG.getArrayLength("s_items")
            LET p_items[i].item_lineno = i
        END FOR
    ...
\end{verbatim}

It is not possible to use the **CANCEL DELETE** instruction in an **AFTER DELETE** block. At this time it is too late to cancel row deletion, as the data row no longer exists in the program array.

**Related concepts**

**BEFORE INSERT block** on page 1546

**AFTER ROW block** on page 1511

**BEFORE MENU block**

If the **MENU** block contains a **BEFORE MENU** clause, statements within this clause are executed before the menu dialog starts.

This block is typically used to hide or disable some menu options depending on the current context of the program. For example, when the current user is not allowed to create new records, the menu options can be disabled as follows:

\begin{verbatim}
MENU "Orders"
    BEFORE MENU
\end{verbatim}
CALL DIALOG.setActionActive("append", can_user_append() )
...
COMMAND "Append" -- creates "append" action (lowercase)
...

END MENU

In TUI mode, the menu options can also be disabled, but they will still be displayed on the screen. The end user will see the option, but cannot select it. In this case it's more convenient to hide the option from the end user with the DIALOG.setActionHidden() method, instead of disabling the action.

Related concepts
The Dialog class on page 2512
The ui.Dialog class provides a set of methods to configure, query and control the current interactive instruction.

DIALOG interaction blocks
Dialog interaction blocks are dialog triggers that can be used to execute specific code when the user executes an action in the dialog. For example, when pressing a button in the form, the corresponding ON ACTION interaction block will be executed.

Interaction blocks also include special handlers such as timeout event handler, drag & drop handlers, and modification triggers for DISPLAY ARRAY sub-dialogs.

ON ACTION block
The ON ACTION action-name blocks execute a sequence of instructions when the user triggers a specific action.

A typical action handler block looks like this:

```
ON ACTION action-name
  instruction
...
```

Action blocks are bound by name to action views (like buttons) in the current form. Action views can be BUTTON, TOOLBAR buttons, or TOPMENU options, and if no explicit action view is defined, actions are rendered with a default action view, depending on the type of front-end.

This example defines an action block to open a typical zoom window and let the user select a customer record:

```
ON ACTION zoom
  CALL zoom_customers() RETURNING st, rec.cust_id, rec.cust_name
```

In a dialog handling user input such as INPUT, INPUT ARRAY and CONSTRUCT, if an action is specific to a field, add the INFIELD clause to have the action automatically enabled when the corresponding field gets the focus:

```
ON ACTION zoom INFIELD cust_city
  CALL zoom_cities() RETURN st, rec.cust_city
```

In most cases actions are decorated with action defaults in form files, but there can be cases where the ON ACTION handler needs to define its own attributes at the program level. This can be done by adding the ATTRIBUTES() clause of ON ACTION:

```
ON ACTION custinfo ATTRIBUTES(DISCLOSUREINDICATOR, IMAGE="info")
  CALL show_customer_info()
```

For more details about action handlers, and action configuration, see Dialog actions on page 1739.

Related concepts
Configuring actions on page 1744
Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with action attributes.
Action handling basics on page 1739
This topic describes the basics of action views, action events, and action handlers.

**ON IDLE block**

The **ON IDLE** *seconds* clause defines a set of instructions that must be executed after a given period of user inactivity. This interaction block can be used, for example, to quit the dialog after the user has not interacted with the program for a specified period of time.

Do not mix **ON TIMER** and **ON IDLE** clauses.

As **ON IDLE** can fire field input validation, it is therefore not recommended in dialogs allowing input.

The parameter of **ON IDLE** must be an integer literal or variable. If the value is zero, the dialog timeout is disabled.

It is not recommended to use the **ON IDLE** trigger with a short timeout period such as 1 or 2 seconds; The purpose of this trigger is to give the control back to the program after a relatively long period of inactivity (10, 30 or 60 seconds). This is typically the case when the end user leaves the workstation, or gets a phone call. The program can then execute some code before the user gets the control back.

```plaintext
ON IDLE 30
  IF ask_question("Do you want to reload information from the database?") THEN
    -- Fetch data back from the db server
  END IF
```

**Important:** The timeout value is taken into account when the dialog initializes its internal data structures. If you use a program variable instead of an integer constant, any change of the variable will have no effect if the variable is changed after the dialog has initialized. If you want to change the value of the timeout variable, it must be done before the dialog block.

**Related concepts**

- [Get program control if user is inactive](#)
- [Execute some code after a given number of seconds, when the user does not interact with the program.](#)

**ON TIMER block** on page 1466

**ON KEY block**

An **ON KEY** (*key-name*) block defines an action with a hidden action view (no default button is visible), that executes a sequence of instructions when the user presses the specified key.

**Note:** The **ON KEY** block is supported for backward compatibility with TUI mode applications. In new developments, consider using **ON ACTION** with accelerators defined in action defaults.

An **ON KEY** block can specify up to four accelerator keys. Each key creates a specific action object that will be identified by the key name in lowercase.

For example, **ON KEY(F5,F6)** creates two actions with the names `f5` and `f6`. Each action object will get an `acceleratorName` attribute assigned, with the corresponding accelerator name. The specified keys must be one of the virtual keys.

**Note:** The **KEY()** clause allows a comma-separated list of keys. Up to four keys can be specified. For new developments, consider using a single key, or prefer **ON ACTION** handlers with a single accelerator definition in action defaults.

In GUI mode, action defaults are applied for **ON KEY** actions by using the name of the action (the key name). You can define secondary accelerator keys, as well as default decoration attributes like button text and image, by using the key name as action identifier. The action name is always in lowercase letters.

Check carefully **ON KEY CONTROL-?** statements to avoid having duplicate accelerators for multiple actions due to the accelerators defined by action defaults. Additionally, **ON KEY** statements used with ESC, TAB, UP, DOWN, LEFT, RIGHT, HELP, NEXT, PREVIOUS, INSERT, CONTROL-M, CONTROL-X, CONTROL-V, CONTROL-C and CONTROL-A should be avoided for use in GUI programs, because it's very likely to clash with default accelerators defined in the factory action defaults file provided by default.
By default, ON KEY actions are not decorated with a default button in the action frame (the default action view). You can show the default button by configuring a `text` attribute with the action defaults.

```
ON KEY (CONTROL-Z)
    CALL open_zoom()
```

**Related concepts**

- **Configuring actions** on page 1744
  Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with `action attributes`.

- **Default action views** on page 1743
  A default action view is created to render an action handler when no explicit action view exists for it.

**ON TIMER block**

The `ON TIMER` `seconds` clause defines a set of instructions that must be executed at regular intervals. This interaction block can be used, for example, to check if a message has arrived in a queue, and needs to be processed.

Do not mix `ON TIMER` and `ON IDLE` clauses.

As `ON TIMER` can fire field input validation, it is therefore not recommended in dialogs allowing input.

The parameter of `ON TIMER` must be an integer literal or variable. If the value is zero, the dialog timeout is disabled.

It is not recommended to use the `ON TIMER` trigger with a short timeout period, such as 1 or 2 seconds. The purpose of this trigger is to give the control back to the program after a reasonable period of time, such as 10, 20 or 60 seconds.

```
ON TIMER 30
    CALL check_for_messages()
```

**Important:** The timer value is taken into account when the dialog initializes its internal data structures. If you use a program variable instead of an integer constant, a change of the variable has no effect if the change takes place after the dialog has initialized. If you want to change the value of the timeout variable, it must be done before the dialog block.

**Related concepts**

- **Get program control on a regular (timed) basis** on page 1713
  Execute some code after a given number of seconds, with or without user interaction with the program.

- **ON IDLE block** on page 1465

**ON APPEND block**

Similar to the `ON INSERT` control block, the `ON APPEND` trigger can be used to enable row creation during a `DISPLAY ARRAY` dialog. If this block is defined, the dialog will automatically create the append action. This action can be decorated, enabled and disabled as a regular action.

If the dialog defines an `ON ACTION append` interaction block and the `ON APPEND` block is used, the compiler will stop with error `-8408`.

When the user fires the append action, the dialog first executes the user code of the `AFTER ROW` block if defined. Then the dialog moves to the end of the list, and creates a new row after the last existing row. After creating the row, the dialog executes the user code of the `ON APPEND` block.

The dialog handles only row creation actions and navigation, you must program the record input with a regular `INPUT` statement, to let the end user enter data for the newly-created row. This is typically done with an `INPUT` binding explicitly array fields to the screen record fields. The new current row in the program array is identified with `arr_curr()`, and the current screen line in the form is defined by `SCR_LINE()`:

```
DISPLAY ARRAY arr TO sr.*
...
ON APPEND
```
INPUT arr[arr_curr()].* FROM sr[scr_line()].*;
...

Pay attention to the semicolon ending the INPUT instruction, which is usually needed here to solve a language grammar conflict when nested dialog instructions are implemented.

After the user code is executed, the dialog gets the control back and processes the new row as follows:

- If the INT_FLAG global variable is FALSE and STATUS is zero, the new row is kept in the program array, and the BEFORE ROW block is executed for the newly-created row.
- If the INT_FLAG global variable is TRUE or STATUS is different from zero, the new row is removed from the program array, and the BEFORE ROW block is executed for the row that existed at the current position, before the new row was created.

The DISPLAY ARRAY dialog always resets INT_FLAG to FALSE and STATUS to zero before executing the user code of the ON APPEND block.

The append action is disabled if the maximum number of rows is reached.

If needed, the ON APPEND handler can be configured with action attributes by added an ATTRIBUTES() clause, as with user-defined action handlers:

ON APPEND ATTRIBUTES(TEXT=\"custlist.delete\", IMAGE=\"listdel\")

Related concepts
Record input (INPUT) on page 1482
The INPUT instruction provides single record input control in an application form.

ON UPDATE block on page 1518
ON DELETE block on page 1519

ON INSERT block

Similar to the ON APPEND control block, the ON INSERT trigger can be used to enable row creation during a DISPLAY ARRAY dialog. If this block is defined, the dialog will automatically create the insert action. This action can be decorated, enabled and disabled as a regular action.

If the dialog defines an ON ACTION insert interaction block and the ON INSERT block is used, the compiler will stop with error -8408.

When the user fires the insert action, the dialog first executes the user code of the AFTER ROW block if defined. Then the new row is created: The insert action creates a new row before current row in the list. After creating the row, the dialog executes the user code of the ON INSERT block.

The dialog handles only row creation actions and navigation, you must program the record input with a regular INPUT statement, to let the end user enter data for the newly-created row. This is typically done with an INPUT binding explicitly array fields to the screen record fields. The new current row in the program array is identified with arr_curr(), and the current screen line in the form is defined by scr_line():

DISPLAY ARRAY arr TO sr.*
...
ON INSERT
  INPUT arr[arr_curr()].* FROM sr[scr_line()].*;
...

Pay attention to the semicolon ending the INPUT instruction, which is usually needed here to solve a language grammar conflict when nested dialog instructions are implemented.

After the user code is executed, the dialog gets the control back and processes the new row as follows:

- If the INT_FLAG global variable is FALSE and STATUS is zero, the new row is kept in the program array, and the BEFORE ROW block is executed for the new created row.
• If the INT_FLAG global variable is TRUE or STATUS is different from zero, the new row is removed from the program array, and the BEFORE_ROW block is executed for the row that was existing at the current position, before the new row was created.

The DISPLAY_ARRAY dialog always resets INT_FLAG to FALSE and STATUS to zero before executing the user code of the ON_INSERT block.

The insert action is disabled if the maximum number of rows is reached.

If needed, the ON_INSERT handler can be configured with action attributes by added an ATTRIBUTES() clause, as with user-defined action handlers:

```
ON_INSERT ATTRIBUTES(TEXT="%custlist.delete", IMAGE="listdel")
```

**Related concepts**

Record input (INPUT) on page 1482
The INPUT instruction provides single record input control in an application form.

ON UPDATE block on page 1518
ON DELETE block on page 1519

**ON UPDATE block**

The ON UPDATE trigger can be used to enable row modification during a DISPLAY_ARRAY dialog. If this block is defined, the dialog will automatically create the update action. This action can be decorated, enabled and disabled as regular actions.

You typically configure the TABLE container in the form by defining the DOUBLECLICK attribute to "update", in order to trigger the update action when the user double-clicks on a row.

If the dialog defines an ON ACTION update interaction block and the ON_UPDATE block is used, the compiler will stop with error -8408.

When the user fires the update action, the dialog executes the user code of the ON_UPDATE block.

The dialog handles only the row modification action and navigation, you must program the record input with a regular INPUT statement, to let the end user modify the data of the current row. This is typically done with an INPUT binding explicitly array fields to the screen record fields, with the WITHOUT_DEFAULTS clause. The current row in the program array is identified with arr_curr(), and the current screen line in the form is defined by scr_line():

```
DISPLAY ARRAY arr TO sr.*
...
ON_UPDATE
  INPUT arr[arr_curr()].* WITHOUT_DEFAULTS FROM sr[scr_line()].* ;
...
```

Pay attention to the semicolon ending the INPUT instruction, which is usually needed here to solve a language grammar conflict when nested dialog instructions are implemented.

After the user code is executed, the dialog gets the control back and processes the current row as follows:

• If the INT_FLAG global variable is FALSE and STATUS is zero, the modified values of the current row are kept in the program array.

• If the INT_FLAG global variable is TRUE or STATUS is different from zero, the old values of the current row are restored in the program array.

The DISPLAY_ARRAY dialog always resets INT_FLAG to FALSE and STATUS to zero before executing the user code of the ON UPDATE block.
If needed, the **ON UPDATE** handler can be configured with action attributes by added an `ATTRIBUTES()` clause, as with user-defined action handlers:

```plaintext
ON UPDATE ATTRIBUTES(TEXT="%custlist.delete", IMAGE="listdel")
```

**Related concepts**

- [Record input (INPUT)](on_page_1482)
The `INPUT` instruction provides single record input control in an application form.

- [ON INSERT block](on_page_1517)
- [ON APPEND block](on_page_1516)
- [ON DELETE block](on_page_1519)

**ON DELETE block**
The **ON DELETE** trigger can be used to enable row deletion during a `DISPLAY ARRAY` dialog. If this block is defined, the dialog will automatically create the delete action. This action can be decorated, enabled and disabled as regular actions.

If the dialog defines an **ON ACTION** delete interaction block and the **ON DELETE** block is used, the compiler will stop with error **-8408**.

When the user fires the delete action, the dialog executes the user code of the **ON DELETE** block.

The dialog handles only the row deletion action and navigation, you can typically program a validation dialog box to let the user confirm the deletion. The current row in the program array is identified with `arr_curr()`:

```plaintext
DISPLAY ARRAY arr TO sr.*
...
ON DELETE
  IF fgl_winQuestion("Delete",
    "Do you want to delete this record?",
    "yes", "no\yes", "help", 0) == "no"
  THEN
    LET int_flag = TRUE
  END IF
...
```

After the user code is executed, the dialog gets the control back and processes the current row as follows:

- If the `INT_FLAG` global variable is `FALSE` and `STATUS` is zero, the current row is deleted from the program array, and the **BEFORE ROW** block is executed for the next row in the list.
- If the `INT_FLAG` global variable is `TRUE` or `STATUS` is different from zero, the current row is kept in the program array, and the **BEFORE ROW** block is executed again for the current row.

The `DISPLAY ARRAY` dialog always resets `INT_FLAG` to `FALSE` and `STATUS` to zero before executing the user code of the **ON DELETE** block.

If needed, the **ON DELETE** handler can be configured with action attributes by adding an `ATTRIBUTES()` clause, as with user-defined action handlers:

```plaintext
ON DELETE ATTRIBUTES(TEXT="%custlist.delete", IMAGE="listdel")
```

**Related concepts**

- [ON APPEND block](on_page_1516)
- [ON INSERT block](on_page_1517)
- [ON UPDATE block](on_page_1518)
**ON SELECTION CHANGE block**

The ON SELECTION CHANGE trigger can be used to enable multi-row selection and detect when rows are selected or de-selected by the end user during a DISPLAY ARRAY dialog. If this block is defined, multi-row selection is automatically enabled. However, the feature can be enabled/disabled with the `setSelectionMode()` dialog method.

**Related concepts**

Multiple row selection on page 1851

Multiple row selection allows the end user to select several rows within a list of records.

**ON SORT block**

**Basics**

The ON SORT interaction block can be used to detect when rows have to be sorted in a DISPLAY ARRAY or INPUT ARRAY dialog.

ON SORT is used in two different contexts:

1. In a regular full-list DISPLAY ARRAY / INPUT ARRAY dialog, the ON SORT trigger can be used to detect that a list sort was performed.
2. In a DISPLAY ARRAY using paged mode (ON FILL BUFFER), use ON SORT to detect a sort request from the user and re-fetch the rows from the database in the required order.

**ON SORT in regular full-list DISPLAY ARRAY or INPUT ARRAY**

In a regular DISPLAY ARRAY / INPUT ARRAY dialog not using paged mode, the ON SORT trigger can be used to detect that a list sort was performed.

When the ON SORT block executes in this context, the (visual) sort is already done by the runtime system and the ON SORT block is only used to execute post-sort tasks, such as displaying current row information.

To display the row position information, use the `arrayToVisualIndex()` dialog method to convert the current program row number to the visual row number:

```plaintext
DISPLAY ARRAY arr TO sr.* ...
... ON SORT
  MESSAGE SFMT( "Row: %1/%2",
    DIALOG.arrayToVisualIndex( "sr", DIALOG.getCurrentRow("sr") ),
    DIALOG.getArrayLength( "sr" )
  )
...
```

If needed, you can get the sort column and sort order with the `getSortKey()` and `isSortReverse()` dialog methods:

```plaintext
DISPLAY ARRAY arr TO sr.* ...
... ON SORT
  MESSAGE SFMT( "Sort on %1, %2 order",
    DIALOG.getSortKey("sr"),
    IIF( DIALOG.isSortReverse("sr"), "descending", "ascending" )
  )
...
```

**ON SORT in DISPLAY ARRAY using the paged mode**

In a DISPLAY ARRAY implementing paged mode with ON FILL BUFFER trigger, built-in row sorting is not available because data is provided by pages.
Use the **ON SORT** trigger, to detect a sort request and perform a new SQL query to re-order the rows. In this context, the sort column and sort order are available with the `getSortKey()` and `isSortReverse()` dialog methods:

```plaintext
DEFINE key STRING, rev BOOLEAN

DISPLAY ARRAY arr TO sr.* ...
  ...
  ON SORT
    -- Re-execute the SQL statement to fill the page of rows in ON FILL BUFFER
    -- Assuming that form field names match table column names
    LET key = DIALOG.getSortKey("sa")
    LET rev = DIALOG.isSortReverse("sa")
    IF key IS NULL THEN
      CALL execute_sql( NULL )
    ELSE
      CALL execute_sql( "ORDER BY " || key || IIF(rev," DESC"," " ) )
    END IF
  END ON SORT
```

See [Paged mode of DISPLAY ARRAY](#) on page 1843 for more details about the paged mode in `DISPLAY ARRAY` and how to implement sort in this type of record list dialog.

**Related concepts**

- **List ordering** on page 1862
  List controllers implement a built-in sort. This feature can be disabled if not required.

**ON DRAG_START block**

The **ON DRAG_START** block is executed when the end user begins the drag operation. If this dialog trigger has not been defined, default dragging is enabled for this dialog.

In the **ON DRAG_START** block, the program typically specifies the type of drag & drop operation by calling `ui.DragDrop.setOperation()` with "move" or "copy". This call will define the default and unique drag operation. If needed, the program can allow another type of drag operation with `ui.DragDrop.addPossibleOperation()`. The end user can then choose to move or copy the dragged object, if the drag & drop target allows it.

If the dragged object can be dropped outside the program, the MIME type and drag/drop data must be defined with `ui.DragDrop.setMimeType()` and `ui.DragDrop.setBuffer()` methods.

Example:

```plaintext
DEFINE dnd ui.DragDrop
  ...
  DISPLAY ARRAY arr TO sr.* ...
  ...
  ON DRAG_START (dnd)
    CALL dnd.setOperation("move") -- Move is the default operation
    CALL dnd.addPossibleOperation("copy") -- User can toggle to copy if needed
    CALL dnd.setMimeType("text/plain")
    CALL dnd.setBuffer(arr[arr_curr()].cust_name)
  ...
END DISPLAY
```

**Related concepts**

- **Handle drag & drop data with MIME types** on page 1919
  How to handle MIME types with drag & drop?

- **The DragDrop class** on page 2570
  The `ui.DragDrop` class is used to control the events related to drag & drop events.

- **Drag & drop** on page 1917
Explains programming techniques for the drag & drop feature.

**ON DRAG_FINISHED block**

Execution of the **ON DRAG_FINISHED** block notifies the dialog where the drag started and that the drop operation has been completed or terminated.

Call `ui.DragDrop.getOperation()` to get the final type of operation of the drop. On successful completion, the method returns "move" or "copy"; otherwise the function returns NULL. If NULL is returned, the **ON DRAG_FINISHED** trigger can be ignored.

In cases of successful moves to a target out of the current DISPLAY ARRAY, the application must remove the transferred data from the source model. For example, if a row was moved from dialog A to B, dialog A will get an **ON DRAG_FINISHED** execution after the row was dropped into B, which removes the row from the list A.

The **ON DRAG_FINISHED** interaction block is optional.

```plaintext
DEFINE dnd ui.DragDrop
... DISPLAY ARRAY arr TO sr.* ...
... ON DRAG_START (dnd) 
  LET last_dragged_row = arr_curr()
... ON DRAG_FINISHED (dnd) 
  IF dnd.getOperation() == "move" THEN
    CALL DIALOG.deleteRow(last_dragged_row)
  END IF
... END DISPLAY
```

**Related concepts**

The DragDrop class on page 2570

The `ui.DragDrop` class is used to control the events related to drag & drop events.

**Drag & drop** on page 1917

Explains programming techniques for the drag & drop feature.

**ON DRAG_ENTER block**

When the **ON DROP** control block is defined, the **ON DRAG_ENTER** block will be executed when the mouse cursor enters the visual boundaries of the drop target dialog. Entering the target dialog is accepted by default if no **ON DRAG_ENTER** block is defined. However, when **ON DROP** is defined, it is recommended that you also define **ON DRAG_ENTER** to prevent the drop of objects with an unsupported MIME type coming from other applications.

The program can decide to disallow or allow a specific drop operation with a call to `ui.DragDrop.setOperation()`: passing a NULL to the method will prevent the drop.

To check what MIME type is available in the drag & drop buffer, the program uses the `ui.DragDrop.selectMimeType()` method. This method takes the MIME type as a parameter and returns TRUE if the passed MIME type is used. You can call this method several times to check the availability of different MIME types.

You may also define the visual effect when hovering over the target list with `ui.DragDrop.setFeedback()`.

```plaintext
DEFINE dnd ui.DragDrop
... DISPLAY ARRAY arr TO sr.* ...
... ON DRAG_ENTER (dnd) 
  IF dnd.selectMimeType("text/plain") THEN
    CALL dnd.setOperation("copy")
    CALL dnd.setFeedback("all")
  END IF
... END DISPLAY
```
Once the mouse has entered the target area, subsequent mouse cursor moves can be detected with the ON DRAG_OVER trigger.

When using a table or tree-view as drop target, you can control the visual effect when the mouse moves over the rows, depending on the type of drag & drop you want to achieve.

Basically, a dragged object can be:

1. Inserted in between two rows (visual effect must show where the object will be inserted)
2. Copied/merged to the current row (visual effect must show the row under the mouse)
3. Dropped somewhere on the target widget (the exact location inside the widget does not matter)

The visual effect can be defined with the ui.DragDrop.setFeedback() method, typically called in the ON DRAG_ENTER block.

The values to pass to the setFeedback() method to get the desired visual effects described are respectively:

1. insert (default)
2. select
3. all

**Related concepts**

**The DragDrop class** on page 2570

The ui.DragDrop class is used to control the events related to drag & drop events.

**Drag & drop** on page 1917

Explains programming techniques for the drag & drop feature.

**ON DRAG_OVER block**

When the ON DROP control block is defined, the ON DRAG_OVER block will be executed after ON DRAG_ENTER, when the mouse cursor is moving over the drop target, or when the drag & drop operation has changed (toggling copy/move).

ON DRAG_OVER will be called only once per row, even if the mouse cursor moves over the row.

In the ON DRAG_OVER block, the method ui.DragDrop.getLocationRow() returns the index of the row in the target array, and can be used to allow or disallow the drop. When using a tree-view, you must also check the index returned by the ui.DragDrop.getLocationParent() method to detect if the object was dropped as a sibling or as a child node, and allow/disallow the drop operation accordingly.
The program can change the drop operation at any execution of the ON DRAG_OVER block. You can disallow or allow a specific drop operation with a call to ui.DragDrop.setOperation(); passing a NULL to the method will disallow the drop.

The current operation (returned by ui.DragDrop.getOperation()) is the value set in previous ON DRAG_ENTER or ON DRAG_OVER events, or the operation selected by the end user, if it can toggle between copy and move. Thus, ON DRAG_OVER can occur even if the mouse position has not changed.

If dropping has been prevented with ui.DragDrop.setOperation(NULL) in the previous ON DRAG_OVER event, the program can reset the operation to allow a drop with a call to ui.DragDrop.setOperation() with the operation parameter "move" or "copy".

ON DRAG_OVER will not be called if drop has been disabled in ON DRAG_ENTER with ui.DragDrop.setOperation(NULL)

ON DRAG_OVER is optional, and must only be defined if the operation or the acceptance of the drag object depends on the target row of the drop target.

```
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr TO sr.* ...
...
ON DRAG_ENTER (dnd)
...
ON DRAG_OVER (dnd)
  IF arr[dnd.getLocationRow()].acceptsCopy THEN
    CALL dnd.setOperation("copy")
  ELSE
    CALL dnd.setOperation(NULL)
  END IF
ON DROP (dnd)
...
END DISPLAY
```

During a drag & drop process, the end user (or the target application) can decide to modify the type of the operation, to indicate whether the dragged object has to be copied or moved from the source to the target. For example, in a typical file explorer, by default files are moved when doing a drag & drop on the same disk. To make a copy of a file, you must press the Ctrl key while doing the drag & drop with the mouse.

In the drop target dialog, you can detect such operation changes in the ON DRAG_OVER trigger and query the ui.DragDrop object for the current operation with ui.DragDrop.getOperation(). In the drag source dialog, you typically check ui.DragDrop.getOperation() in the ON DRAG_FINISHED trigger to know what type of operation occurred, to keep ("copy" operation) or delete ("move" operation) the original dragged object.

This example tests the current operation in the drop target list and displays a message accordingly:

```
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr TO sr.* ...
...
ON DRAG_ENTER (dnd)
...
ON DRAG_OVER (dnd)
  CASE dnd.getOperation()
    WHEN "move"
      MESSAGE "The object will be moved to row ", dnd.getLocationRow()
    WHEN "copy"
      MESSAGE "The object will be copied to row ", dnd.getLocationRow()
  END CASE
...
ON DROP (dnd)
```

**Related concepts**

The DragDrop class on page 2570
The `ui.DragDrop` class is used to control the events related to drag & drop events.

Tree views on page 1889
Describes how to implement tree views.

Drag & drop on page 1917
Explains programming techniques for the drag & drop feature.

**ON DROP block**

To enable drop actions on a list, you must define the `ON DROP` block; otherwise the list will not accept drop actions.

The `ON DROP` block is executed after the end user has released the mouse button to drop the dragged object. `ON DROP` will not occur if drop has been disallowed in the previous `ON DRAG_OVER` event or in `ON DRAG_ENTER` with a call to `ui.DragDrop.setOperation(NULL)`.

When `ON DROP` executes, the MIME type of the dragged object can be checked with `ui.DragDrop.getSelectedMimeType()`. Then call the `ui.DragDrop.getBuffer()` method to retrieve drag & drop data from external applications.

Ideally, the drop operation is accepted (there is no need for additional calls to `ui.DragDrop.setOperation()`).

In this block, the `ui.DragDrop.getLocationRow()` method returns the index of the row in the target array, and can be used to execute the code to get the drop data / object into the row that has been chosen by the user.

```define
d dnd ui.DragDrop
...
DISPLAY ARRAY arr TO sr.* ...
  ...  
  ON DROP (dnd)  
    LET arr[dnd.getLocationRow()].capacity == dnd.getBuffer()
  ...
END DISPLAY```

If the drag & drop operations are local to the same list or tree-view controller, you can use the `ui.DragDrop.dropInternal()` method to simplify the code. This method implements the typical move of the dragged rows or tree-view node. This is especially useful in case of a tree-view, but is also the preferred way to move rows around in simple tables.

This `ON DROP` code example uses the `dropInternal()` method:

```define
d dnd ui.DragDrop
...
DISPLAY ARRAY arr_tree TO sr_tree.* ...
  ...  
  ON DROP (dnd)  
    CALL dnd.dropInternal()
  ...
END DISPLAY```

If you want to implement by hand the code to drop a node in a tree-view, you must check the index returned by the `ui.DragDrop.getLocationParent()` method to detect if the object was dropped as a sibling or as a child node, and execute the code corresponding to the drop operation. If the drop target row index returned by `getLocationRow()` is a child of the parent row index returned by `getLocationParent()`, the new row must be inserted before `getLocationRow()`; otherwise the new row must be added as a child of the parent node identified by `getLocationParent()`.
Related concepts
The DragDrop class on page 2570
The `ui.DragDrop` class is used to control the events related to drag & drop events.

Drag & drop on page 1917
Explains programming techniques for the drag & drop feature.

**DIALOG control instructions**
Dialogue control instructions are language instructions dedicated to dialog control, to programmatically force the dialog to behave in a given way.

For example the NEXT FIELD instruction forces the focus to a specific form field.

**NEXT FIELD instruction**

**Understanding the NEXT FIELD instruction**

The NEXT FIELD `field-name` instruction gives the focus to the specified field. This instruction can be used to control field input, in BEFORE FIELD, ON CHANGE or AFTER FIELD blocks, it can also force a DISPLAY ARRAY or INPUT ARRAY to stay in the current row when NEXT FIELD is used in the AFTER ROW block.

If it exists, the BEFORE FIELD block of the corresponding field is executed.

In editable dialogs, the purpose of the NEXT FIELD instruction is to give the focus to an editable field. Make sure that the field specified in NEXT FIELD is active and editable, or use NEXT FIELD CURRENT.

In a DISPLAY ARRAY using the FOCUSONFIELD attribute, NEXT FIELD can be used in conjunction with DIALOG.setCurrentRow(), to set the focus to a specific cell in the list.

Instead of the NEXT FIELD instruction, you can use the DIALOG.nextField("field-name") method to register a field, for example when the name is not known at compile time. However, this method only registers the field. It does not stop code execution, like the NEXT FIELD instruction does. You must execute a CONTINUE DIALOG to get the same behavior as NEXT FIELD.

**Form field identification with NEXT FIELD**

With the NEXT FIELD instruction, fields are identified by the form field name specification, not the program variable name used by the dialog.

Form fields are bound to program variables with the binding clause of the dialog instruction (INPUT variable-list FROM field-list, INPUT BY NAME variable-list, CONSTRUCT BY NAME sql ON column-list, CONSTRUCT sql ON column-list FROM field-list, INPUT ARRAY array-name FROM screen-array.*).

The field name specification can be any of the following:

- `field-name`
- `table-name.field-name`
- `screen-record-name.field-name`
- `FORMONLY.field-name`

Here are some examples:

- "cust_name"
- "customer.cust_name"
- "cust_screen_record.cust_name"
- "item_screen_array.item_label"
- "formonly.total"

When no field name prefix is used, the first form field matching that simple field name is used.

When using a prefix in the field name specification, it must match the field prefix assigned by the dialog field binding method used at the beginning of the interactive statement: When no screen-record has been explicitly specified in the
field binding clause (for example, when using `INPUT BY NAME variable-list`), the field prefix must be the database table name or FORMONLY, as defined in the form file, or any valid screen-record using that field. When the FROM clause of the dialog specifies an explicit screen-record (for example, in `INPUT variable-list FROM screen-record.* / field-list-with-screen-record-prefix` or `INPUT ARRAY array-name FROM screen-array.*`), the field prefix must be the screen-record name used in the FROM clause.

Abstract field identification is supported with the CURRENT, NEXT and PREVIOUS keywords. These keywords represent the current, next and previous fields respectively. When using FIELD ORDER FORM, the NEXT and PREVIOUS options follow the tabbing order defined by the form. Otherwise, they follow the order defined by the input binding list (with the FROM or BY NAME clause).

In a procedural dialog, if the focus is in the first field of an INPUT or CONSTRUCT sub-dialog, NEXT FIELD will jump out of the current sub-dialog and set the focus to the previous sub-dialog. If the focus is in the last field of an INPUT or CONSTRUCT sub-dialog, NEXT FIELD will jump out of the current sub-dialog and set the focus to the next sub-dialog. NEXT FIELD NEXT or NEXT FIELD PREVIOUS also jumps to another sub-dialog when the focus is in a DISPLAY ARRAY sub-dialog. However, when using an INPUT ARRAY sub-dialog, NEXT FIELD NEXT from within the first column will loop to the first column of the current row, and NEXT FIELD PREVIOUS from within the first column will jump to the last column of the current row - the focus stays in the current INPUT ARRAY sub-dialog. When another sub-dialog gets the focus because of a NEXT FIELD NEXT/ PREVIOUS, the newly-selected field depends on the sub-dialog type, following the tabbing order as if the end-user had pressed the tab or Shift-Tab key combination.

### NEXT FIELD to a non-editable field

Non-editable fields are:

- Fields defined with the NOENTRY attribute in the form.
- Fields defined as PHANTOM fields in the form.
- Fields disabled at runtime with `DIALOG.setFieldActive()`.
- Fields using a widget that does not allow input, such as a LABEL.

In an INPUT, INPUT ARRAY or CONSTRUCT instruction, if a NEXT FIELD instruction specifies a non-editable field, the BEFORE FIELD block of that field is executed. Then the dialog tries to give the focus to that field. Since the field cannot get the focus, the dialog will perform the last pressed navigation key (Tab, Shift-Tab, Left, Right, Up, Down, Accept) and execute the related control blocks, including the AFTER FIELD block of the non-editable field. If no last key is identified, the dialog considers Tab as fallback and moves to the next editable field as defined by the FIELD ORDER mode used by the dialog.

**Note:** In an INPUT ARRAY, doing a NEXT FIELD to a non-editable field which has no editable fields in the last navigation direction, the dialog may move to the next or previous row and give the focus to the first or last editable field of that row.

When selecting a non-editable field with NEXT FIELD NEXT, the runtime system will re-select the current field since it is the next editable field in the dialog. As a result the end user sees no change.

**Important:** Doing a NEXT FIELD to a non-editable field can lead to infinite loops in the dialog; To stay in the current field / row, use NEXT FIELD CURRENT instead.

### NEXT FIELD in procedural DIALOG blocks

In a procedural dialog block, the NEXT FIELD field-name instruction gives the focus to the specified field controlled by INPUT, INPUT ARRAY or CONSTRUCT, or to a read-only list when using DISPLAY ARRAY.

When using a DISPLAY ARRAY sub-dialog, it is possible to give the focus to the list, by specifying the name of the first column as argument for NEXT FIELD.

If the target field specified in the NEXT FIELD instruction is inside the current sub-dialog, neither AFTER FIELD nor AFTER ROW will be invoked for the field or list you are leaving. However, the BEFORE FIELD control blocks of the destination field (or the BEFORE ROW in case of read-only list) will be executed.
If the target field specified in the NEXT FIELD instruction is outside the current sub-dialog, the AFTER FIELD, AFTER INSERT, AFTER ROW, and AFTER INPUT, AFTER DISPLAY, AFTER /CONSTRUCT control blocks will be invoked for the field or list you are leaving. Form-level validation rules will also be checked, as if the user had selected the new sub-dialog himself. This guarantees the current sub-dialog is left in a consistent state. The BEFORE INPUT, BEFORE DISPLAY, BEFORE CONSTRUCT, BEFORE ROW and the BEFORE FIELD control blocks of the destination field / list will then be executed.

**NEXT FIELD in record list control blocks**

When using NEXT FIELD in AFTER ROW or in ON ROW CHANGE of a DISPLAY ARRAY or INPUT ARRAY, the dialog will stay in the current row and give control back to the user.

This behavior allows you to implement data input rules:

```plaintext
AFTER ROW
   IF NOT int_flag AND arr_count() <= arr_curr() THEN
      IF arr[arr_curr()].it_count * arr[arr_curr()].it_value > maxval THEN
         ERROR "Amount of line exceeds max value."
      NEXT FIELD item_count
   END IF
END IF
```

If a condition is not met for a given field, do a NEXT FIELD to that field to point the user to the problem. If a global condition for the current row is not met, use NEXT FIELD CURRENT to stay in the current row and current field.

**Related concepts**

- Giving the focus to a form element on page 1731
- How to force the focus to move or stay in a specific form element using program code.

**CLEAR instruction in dialogs**

The CLEAR field-list and CLEAR SCREEN ARRAY screen-array.* instructions clear the value buffer of specified form fields. The buffers are directly changed in the current form, and the program variables bound to the dialog are left unchanged. CLEAR can be used outside any dialog instruction, such as the DISPLAY BY NAME / TO instructions.

When a dialog is configured with the UNBUFFERED mode, there is no reason to clear field buffers since any variable assignment will synchronize field buffers. Actually, changing the field buffers with DISPLAY or CLEAR instruction in an UNBUFFERED dialog will have no visual effect, because the variables bound to the dialog will be used to reset the field buffer just before giving control back to the user. To clear fields of an UNBUFFERED dialog, just set to NULL the variables bound to the dialog. However, when using a CONSTRUCT, no program variables are associated with the dialog and no UNBUFFERED concept exits, and the CLEAR or DISPLAY TO / BY NAME instructions are the only way to modify the CONSTRUCT fields.

A screen array with a screen-line specification doesn't make much sense in a GUI application using TABLE containers, you can therefore use the CLEAR SCREEN ARRAY instruction to clear all rows of a list.

**Related concepts**

- Static display (DISPLAY/ERROR/MESSAGE/CLEAR) on page 1452
- This section explains the instructions displaying static information to application forms, such as DISPLAY, ERROR, MESSAGE, CLEAR.

**DISPLAY TO / BY NAME instruction**

The DISPLAY variable-list TO field-list or DISPLAY BY NAME variable-list instruction fills the value buffers of specified form fields with the values contained in the specified program variables. The DISPLAY TO / BY NAME instruction changes the buffers directly in the current form, not the program variables
bound to the dialog. DISPLAY can be used outside any dialog instruction, in the same way as the CLEAR instruction. DISPLAY also sets the modification flag of fields.

As DIALOG is typically used with the UNBUFFERED mode, there is no reason to set field buffers in a DIALOG block since any variable assignment will synchronize field buffers. Actually, changing the field buffers with the DISPLAY or CLEAR instruction will have no visual effect if the fields are used by a dialog working in UNBUFFERED mode, because the variables bound to the dialog will be used to reset the field buffer just before giving control back to the user. So if you want to set field values, just assign the variables and the fields will be synchronized. However, when using a CONSTRUCT binding, you may want to set field buffers with this DISPLAY instruction, as there are no program variables bound to fields (with CONSTRUCT, only one string variable is bound to hold the SQL condition).

Instead of using a DISPLAY instruction to set the modification flag of fields to simulate user input, use the DIALOG.setFieldTouched() method instead.

**Related concepts**

Input field modification flag on page 1723
Each input field controlled by a dialog instruction has a modification flag.

**CONTINUE DIALOG instruction**

The CONTINUE DIALOG statement continues the execution of a DIALOG instruction, skipping all statements appearing after this instruction.

Control returns to the dialog instruction, which executes remaining control blocks as if the program reached the end of the current control block. Then the control goes back to the user and the dialog waits for a new event.

The CONTINUE DIALOG statement is useful when program control is nested within multiple conditional statements, and you want to return control to the user by skipping the rest of the statements.

In the following code example, an ON ACTION block gives control back to the dialog, skipping all instructions below line 04:

```
ON ACTION zoom
  IF p_cust.cust_id IS NULL OR p_cust.cust_name IS NULL THEN
    ERROR "Zoom window cannot be opened if no info to identify customer"
    CONTINUE DIALOG
  END IF
  IF p_cust.cust_address IS NULL THEN
    ...
```

If CONTINUE DIALOG is called in a control block that is not AFTER DIALOG, further control blocks might be executed depending on the context. Actually, CONTINUE DIALOG just instructs the dialog to continue as if the code in the control block was terminated (it is a kind of GOTO end_of_control_block). However, when executed in AFTER DIALOG, the focus returns to the current field or read-only list. In this case the BEFORE ROW and BEFORE FIELD triggers will be invoked.

A CONTINUE DIALOG in AFTER FIELD, AFTER INPUT, AFTER DISPLAY or AFTER CONSTRUCT will only stop the program flow of the current block of statements; instructions after CONTINUE DIALOG will not be executed. If the user has selected a field in a different sub-dialog, this new field will get the focus and all necessary AFTER / BEFORE control blocks will be executed.

In case of input error in a field, the best practice is to use a NEXT FIELD instruction to stay in the dialog and set the focus to the field that the user has to correct.

**Related concepts**

NEXT FIELD instruction on page 1497

**EXIT DIALOG instruction**

The EXIT DIALOG statement terminates a procedural DIALOG block without any further control block execution.

**Note:** When used in a declarative DIALOG block, the EXIT DIALOG instruction does only make sense when the declarative dialog block is included in a procedural dialog block with the SUBDIALOG clause.
Program flow resumes at the Instruction following the END DIALOG keywords. Blocks such as AFTER DIALOG will not be executed.

```
ON ACTION quit
   EXIT DIALOG
```

When leaving the DIALOG instruction, all form items used by the dialog will be disabled until another interactive statement takes control.

The EXIT DIALOG instruction leaves the DIALOG block immediately, while CANCEL DIALOG makes some additional tasks.

**Related concepts**
- CANCEL DIALOG instruction on page 1641
- ACCEPT DIALOG instruction on page 1640

**ACCEPT DIALOG instruction**

The ACCEPT DIALOG statement validates all input fields bound to the DIALOG instruction and leaves the block if no error is raised.

**Note:** When used in a declarative DIALOG block, the ACCEPT DIALOG instruction does only make sense when the declarative dialog block is included in a procedural dialog block with the SUBDIALOG clause.

When defined in the dialog block, ON CHANGE, AFTER FIELD, AFTER ROW, AFTER INPUT, AFTER DISPLAY, AFTER CONSTRUCT control blocks will be executed when ACCEPT DIALOG is performed.

The statements appearing after the ACCEPT DIALOG instruction will be skipped.

You typically code an ACCEPT DIALOG in an ON ACTION accept block:

```
ON ACTION accept
   ACCEPT DIALOG
```

**Important:** Any usage of ACCEPT DIALOG outside an ON ACTION accept block is not intended and its behavior is undefined.

Input field validation is a process that does several successive validation tasks:

1. The current field value is checked, depending on the variable data type (for example, the user must input a valid date in a DATE field).
2. **NOT NULL** field attributes are checked for all input fields. This attribute forces the field to have a value set by program or entered by the user. If the field contains no value, the constraint is not satisfied. Input values are right-trimmed, so if the user inputs only spaces, this corresponds to a NULL value which does not fulfill the NOT NULL constraint.
3. **REQUIRED** field attributes are checked for all input fields. This attribute forces the field to have a default value, or to be modified by the user or by program with a DISPLAY TO / BY NAME or DIALOG.setFieldTouched() call. If the field was not modified during the dialog, the REQUIRED constraint is not satisfied.
4. **INCLUDE** field attributes are checked for all input fields. This attribute forces the field to contain a value that is listed in the include list. If the field contains a value that is not in the list, the constraint is not satisfied.

If a field does not satisfy one of these constraints, dialog termination is canceled, an error message is displayed, and the focus goes to the first field causing a problem.

After input field validation has succeeded, different types of control blocks will be executed, such as AFTER FIELD, AFTER ROW, AFTER INPUT and AFTER DIALOG.

In order to validate some parts of the dialog without leaving the block, use the DIALOG.validate() method.

**Related concepts**
- Input field modification flag on page 1723
Each input field controlled by a dialog instruction has a modification flag.

CANCEL DIALOG instruction on page 1641
EXIT DIALOG instruction on page 1640
ui.Dialog.accept on page 2520
Validates and terminates the dialog.

CANCEL DIALOG instruction

The CANCEL DIALOG statement terminates a procedural DIALOG block, after executing the AFTER INPUT, AFTER DISPLAY, AFTER CONSTRUCT control block of the current sub-dialog, and the AFTER DIALOG control block.

**Note:** When used in a declarative DIALOG block, the CANCEL DIALOG instruction does only make sense when the declarative dialog block is included in a procedural dialog block with the SUBDIALOG clause.

The CANCEL DIALOG instruction can be used from multiple dialogs to mimic the cancel default action of single dialogs.

The CANCEL DIALOG instruction makes the following:

1. Set the INT_FLAG register to TRUE.
2. If defined, execute the code in the AFTER INPUT, AFTER DISPLAY or AFTER CONSTRUCT block of the current sub-dialog.
3. If defined, execute the code in the AFTER DIALOG block.

The statements appearing after the CANCEL DIALOG instruction will be skipped.

You typically code an CANCEL DIALOG in an ON ACTION cancel block:

```plaintext
ON ACTION cancel CANCEL DIALOG
```

**Note:** The default settings regarding action attributes for the cancel action define the validate attribute to "no", in order to avoid current field validation for this action. This is important when using the UNBUFFERED mode. For more details, see Actions configuration for field validation on page 1722.

**Related concepts**

EXIT DIALOG instruction on page 1640
ACCEPT DIALOG instruction on page 1640
ui.Dialog.cancel on page 2527
Cancels a parent dialog from a sub-dialog.

CANCEL DELETE instruction

In a list controlled by an INPUT ARRAY, row deletion can be canceled by using the CANCEL DELETE instruction in the BEFORE DELETE block. Using this instruction in a different place will generate a compilation error.

When the CANCEL DELETE instruction is executed, the current BEFORE DELETE block is terminated without any other trigger execution (no BEFORE ROW or BEFORE FIELD is executed), and the program execution continues in the user event loop.

You can, for example, prevent row deletion based on some condition:

```plaintext
BEFORE DELETE
  IF user_can_delete() == FALSE THEN
    ERROR "You are not allowed to delete rows"
    CANCEL DELETE
  END IF
```

The instructions that appear after CANCEL DELETE will be skipped.
If the row deletion condition is known before the delete action occurs, disable the delete action to prevent the user from performing a delete row action with the DIALOG.setActionActive() method:

```
CALL DIALOG.setActionActive("delete", FALSE)
```

It is also possible to prevent the user from deleting rows with the DELETE ROW = FALSE option in the ATTRIBUTE clause.

**Related concepts**

BEFORE DELETE block on page 1547

**CANCEL INSERT instruction**

In a list controlled by an INPUT ARRAY, row creation can be canceled by the program with the CANCEL INSERT instruction. This instruction can only be used in the BEFORE INSERT and AFTER INSERT control blocks. If it appears at a different place, the compiler will generate an error.

The instructions that appear after CANCEL INSERT will be skipped.

If the row creation condition is known before the insert/append action occurs, disable the insert and/or append actions to prevent the user from creating new rows, with DIALOG.setActionActive():

```
CALL DIALOG.setActionActive("insert", FALSE)
CALL DIALOG.setActionActive("append", FALSE)
```

However, this will not prevent the user from appending a new temporary row at the end of the list, when moving down after the last row. To prevent row creation completely, use the INSERT ROW = FALSE and APPEND ROW = FALSE options in the ATTRIBUTE clause of INPUT ARRAY, or combine with the AUTO APPEND = FALSE attribute.

**CANCEL INSERT in BEFORE INSERT**

A CANCEL INSERT executed inside a BEFORE INSERT block prevents the new row creation. The following tasks are performed:

1. No new row will be created (the new row is not yet shown to the user).
2. The BEFORE INSERT block is terminated (further instructions are skipped).
3. The BEFORE ROW and BEFORE FIELD triggers are executed.
4. Control goes back to the user.

You can, for example, cancel a row creation if the user is not allowed to create rows:

```
BEFORE INSERT
  IF NOT user_can_insert THEN
    ERROR "You are not allowed to insert rows"
    CANCEL INSERT
  END IF
```

Executing CANCEL INSERT in BEFORE INSERT will also cancel a temporary row creation, except when there are no more rows in the list. In this case, CANCEL INSERT will just be ignored and leave the new row as is (otherwise, the instruction would loop without end). You can prevent automatic temporary row creation with the AUTO APPEND=FALSE attribute. If AUTO APPEND=FALSE and a CANCEL INSERT is executed in BEFORE INSERT (user has invoked an append action), the temporary row will be deleted and list will remain empty if it was the last row.

**CANCEL INSERT in AFTER INSERT**

A CANCEL INSERT executed inside an AFTER INSERT block removes the newly created row. The following tasks are performed:

1. The newly created row is removed from the list (the row exists now and user has entered data).
2. The **AFTER INSERT** block is terminated (further instructions are skipped).
3. The **BEFORE ROW** and **BEFORE FIELD** triggers are executed.
4. The control goes back to the user.

You can, for example, cancel a row insertion if a database error occurs when you try to insert the row into a database table:

```
AFTER INSERT
  WHENEVER ERROR CONTINUE
  LET r = DIALOG.getCurrentRow("s_items")
  INSERT INTO items VALUES ( p_items[r].* )
  WHENEVER ERROR STOP
  IF SQLCA.SQLCODE<>0 THEN
    ERROR SQLERRMESSAGE
    CANCEL INSERT
  END IF
```

**Related concepts**

- **BEFORE DELETE block** on page 1547
- **Appending rows in INPUT ARRAY** on page 1848

Rows appended at the end of an editable list are temporary until they are edited.

**Examples**

Declarative dialogs usage examples.

**Example 1: Simple input**

The module "comment.4gl":

```
PUBLIC TYPE cb_comment_event FUNCTION (event STRING)  
PRIVATE DEFINE cb_ce cb_comment_event

PRIVATE DEFINE the_comment VARCHAR(200)

PUBLIC FUNCTION set_event_callback(f cb_comment_event)
  LET cb_ce = f
END FUNCTION

PUBLIC FUNCTION set_event_callback_2(f FUNCTION (event STRING))
  LET cb_ce = f
END FUNCTION

PUBLIC FUNCTION set_comment(s STRING)
  LET the_comment = s
END FUNCTION

PUBLIC FUNCTION get_comment()
  RETURN the_comment
END FUNCTION

DIALOG comment_input()
  INPUT BY NAME the_comment ATTRIBUTES(WITHOUT DEFAULTS)
  ON ACTION add_sep ATTRIBUTES(TEXT="Add sep")
    LET the_comment = the_comment || "\n---\n"
    CALL fgl_dialog_setcursor(LENGTH(the_comment)+1)
    IF cb_ce IS NOT NULL THEN
      CALL cb_ce("comment_changed")
    END IF
  ON ACTION clr_cmt ATTRIBUTES(TEXT="Clear")
    LET the_comment = NULL
    IF cb_ce IS NOT NULL THEN
      CALL cb_ce("comment_changed")
    END IF
```
For the complete example, see Example 3: DIALOG with SUBDIALOG on page 1646.

**Example 2: Simple list**

The module "list1.4gl":

```gl
DEFINE arr DYNAMIC ARRAY OF RECORD
    id INTEGER,
    name VARCHAR(50)
END RECORD

FUNCTION start_list1()
    DEFINE i INTEGER
    IF ui.Window.forName("w_list1") IS NULL THEN
        FOR i=1 TO 10
            LET arr[i].id = i
            LET arr[i].name = "Record ||i"
        END FOR
        OPEN WINDOW w_list1 WITH FORM "simple_list"
        START DIALOG control_list1
    ELSE
        CURRENT WINDOW IS w_list1
    END IF
END FUNCTION

FUNCTION terminate_list1()
    TERMINATE DIALOG control_list1
    CLOSE WINDOW w_list1
END FUNCTION

DIALOG control_list1()
    DISPLAY ARRAY arr TO sr.*
    ON ACTION add_row
        CALL DIALOG.appendRow("sr")
        LET arr[arr.getLength()].id = arr.getLength()
        LET arr[arr.getLength()].name = "[new record]"
    ON ACTION close
        CALL terminate_list1()
    END DISPLAY
END DIALOG
```

For the complete example, see Example 1: Two independent record lists on page 1705.

**Parallel dialogs (START DIALOG)**

The **START DIALOG** and **TERMINATE DIALOG** instructions provide for dialog functionality executing concurrently in different application forms.

**Understanding parallel dialogs**

*Parallel dialogs* allow the control of several forms simultaneously.

*Parallel dialogs* use different declarative **DIALOG** blocks, in conjunction with the **START DIALOG** and **TERMINATE DIALOG** instructions, and an event loop using the **fgl_eventLoop()** built-in function, in order to control several forms simultaneously.

**Important:** This feature is only for mobile platforms.

Each dialog acts independently to control several elements of a window/form. During the execution of parallel dialogs, the user can switch to a window/form that is controlled by another running declarative **DIALOG** block. For more details about categories of dialogs, see **What are dialog controllers?** on page 1707.
**Important:** On mobile devices, parallel dialogs do not support INPUT ARRAY declarative dialogs.

The `START DIALOG` and `TERMINATE DIALOG` instructions use declarative dialogs, described in detail in the Declarative dialogs (DIALOG - at module level) on page 1648 chapter.

The parallel dialog feature was introduced to implement mobile applications, where several forms can be accessed simultaneously, for example to get "split views" on mobile devices:

![Form with Split View (Android™)](image)

**Figure 84: Form with Split View (Android™)**

In terms of semantics, behavior and control block execution, a declarative dialog started with a `START DIALOG` instruction behaves like a procedural `DIALOG` block.

**Important:** Parallel dialogs implicitly use the `UNBUFFERED mode`. It is not possible to change this mode when using parallel dialogs.

In order to execute parallel dialogs, you must implement a main interaction event loop, by using the `fgl_eventLoop()` built-in function. The minimal event loop code to implement is:

```plaintext
WHILE fgl_eventLoop() 
END WHILE
```
Once the declarative dialogs and the interaction event loop are defined, it is possible to create the windows with OPEN WINDOW, and initiate the dialogs with the START DIALOG instruction.

If needed, show a given dialog window with the CURRENT WINDOW instruction. Additionally, (especially when implementing split views), you may want to "restart" a detail dialog, for example when selecting a new row in the main record list. To restart the detail dialog, execute TERMINATE DIALOG, followed by START DIALOG for the detail dialog. See split view programming for more details.

To finish a given dialog, execute the TERMINATE DIALOG instruction and close the dedicated window with CLOSE WINDOW window-name.

From a set of running parallel dialogs, it is possible to switch to a modal dialog by creating a dedicated window, and executing a procedural dialog instruction. When the procedural dialog is terminated, close the dedicated window, and the control will go back to the parallel dialog set.

**Related concepts**

Windows and forms on page 1131
The section describes the concept of windows and forms in the language.

**Syntax of the START DIALOG instruction**

Starts the instance of a declarative dialog.

**Syntax**

```plaintext
START DIALOG dialog-name
```

1. `dialog-name` is the identifier of a declarative DIALOG block.

**Usage**

The START DIALOG instruction starts the declarative dialog block identified by the name passed.

The current window/form will be used to attach form fields and action views to the variables and action handlers implemented in the referenced declarative dialog.

The START DIALOG does in fact register the specified dialog to be activated when the parallel dialog event loop executes.

The started dialog can be terminated with TERMINATE DIALOG.

**Example**

This example shows a START DIALOG instruction in a function that initializes a parallel dialog in a split view context:

```plaintext
FUNCTION params()
    IF ui.Window.forName("w_params") IS NULL THEN
        OPEN WINDOW w_params WITH FORM "parameters" ATTRIBUTES(TYPE=LEFT)
        LET params.user_name="Tom"
        LET params.auto_sync="Y"
        DISPLAY BY NAME params.*
        START DIALOG d_params_menu
    END IF
    CURRENT WINDOW IS w_params
END FUNCTION
```

**Related concepts**

Syntax of the TERMINATE DIALOG instruction on page 1703
Terminates the instance of a declarative dialog.

Syntax of the declarative DIALOG block on page 1649
The declarative DIALOG block defines an interactive instruction that can be used by a parent DIALOG, or as parallel dialog.

fgl_eventloop() on page 2284
Waits for a user interaction event.

Syntax of the TERMINATE DIALOG instruction
Terminates the instance of a declarative dialog.

Syntax

| TERMINATE DIALOG dialog-name |

1. **dialog-name** is the identifier of a declarative DIALOG block.

Usage

The TERMINATE DIALOG instruction stops a declarative dialog identified by the name passed.

If the intent is to finish the parallel dialog, the corresponding window/form bound to the dialog should be closed after TERMINATE DIALOG.

However, TERMINATE DIALOG can also be used in conjunction with START DIALOG, to achieve a "restart" of the parallel dialog.

**Note:** TERMINATE DIALOG will not raise an error, if the dialog was not yet started with START DIALOG. This is required to implement the "restart" pattern.

The next code example shows a typical restart pattern on a detail parallel dialog, when a new row is selected in the master list:

```plaintext
DIALOG d_list_view()
   DISPLAY ARRAY arr TO sr.*
   ATTRIBUTES(ACCESSORIETYPER=DISCLOSUREINDICATOR)
   BEFORE ROW -- in BEFORE ROW, we restart the details view
      CURRENT WINDOW IS w_right
      TERMINATE DIALOG d_detail_view
      LET curr_pa = arr_curr()
      DISPLAY BY NAME arr[curr_pa].*
      DISPLAY SFMT("tapped row %1",arr_curr()) TO info
      START DIALOG d_detail_view
      CURRENT WINDOW IS w_left
   ...
```

Related concepts

Syntax of the START DIALOG instruction on page 1702
Starts the instance of a declarative dialog.

Syntax of the declarative DIALOG block on page 1649
The declarative DIALOG block defines an interactive instruction that can be used by a parent DIALOG, or as parallel dialog.

fgl_eventloop() on page 2284
Waits for a user interaction event.

Parallel dialog programming steps
This procedure describes how to implement parallel dialogs with a declarative DIALOG block.

1. Create the forms and declarative dialog modules as described in Declarative dialog programming steps on page 1655.
2. Define a FUNCTION to create the dialog instance.
a) Add a test to check if the window and form combination dedicated to the dialog is already created, using `ui.Window.forName()` function. If the window does not yet exist, create it by using the `OPEN WINDOW window-name WITH FORM` instruction. If the window exists, make it current with the `CURRENT WINDOW IS window-name` instruction.

b) Fill the module variables (the data model) with data. For lists, you typically use a result set cursor.

c) Start the dialog with the `START DIALOG dialog-name` instruction.

3. Define a FUNCTION to terminate the dialog instance.

   a) In the function, finish the dialog with `TERMINATE DIALOG dialog-name`.
   
   b) Close the window dedicated to the dialog with `CLOSE WINDOW window-name`.
   
   c) If needed, free the data model (clear large program arrays) and database cursors, to save memory.

4. If needed, add an `ON ACTION close` action handler to the declarative dialog, that calls the terminate function. This allows the end user to close the front-end window and stop the dialog.

5. In another module, implement the `WHILE` loop using the `fgl_eventLoop()` built-in function to handle interaction events for parallel dialogs. This module uses the start and terminate functions to control the individual dialog modules.

   The simplest form of the user interaction event loop is:

   ```
   WHILE fgl_eventLoop()
   END WHILE
   ```

**Related concepts**

*Syntax of the declarative DIALOG block* on page 1649

The declarative DIALOG block defines an interactive instruction that can be used by a parent DIALOG, or as parallel dialog.

**Using parallel dialogs**

Dialog coding concepts, configuration and code structure.

Parallel dialogs are implemented with declarative DIALOG blocks:

```dialog
DIALOG control_list1(
    DISPLAY ARRAY arr TO sr.*
    ...
END DISPLAY
END DIALOG
```

Instances of declarative dialogs are started with the `START DIALOG` instruction (typically preceded by an `OPEN WINDOW` instruction that creates the corresponding window+form):

```dialog
OPEN WINDOW w_list1 WITH FORM "simple_list"
START DIALOG control_list1
```

Instances of declarative dialogs are stopped with the `TERMINATE DIALOG` instruction:

```dialog
TERMINATE DIALOG control_list1
```

In order to initiate parallel dialogs, the program must run the user interface event loop by calling the `fgl_eventloop()` on page 2284 built-in function in a `WHILE` loop.

The simplest form of the user interaction event loop is:

```dialog
WHILE fgl_eventLoop()
END WHILE
```

Before or during the user interface event loop, modal dialogs can be executed without conflicts. For more details about parallel dialogs versus modal dialogs, see *What are dialog controllers?* on page 1707.
Examples
Parallel dialogs usage examples.

Example 1: Two independent record lists

Form file "simple_list.per":

```
LAYOUT
GRID
{
  <T t1               >
    [c1  |c2                  ]
    [c1  |c2                  ]
    [c1  |c2                  ]
}
END
END
ATTRIBUTES
  c1 = FORMONLY.col1;
  c2 = FORMONLY.col2;
END
INSTRUCTIONS
  SCREEN RECORD sr(FORMONLY.*);
END
```

The module "list1.4gl":

```
DEFINE arr DYNAMIC ARRAY OF RECORD
  id INTEGER,
  name VARCHAR(50)
END RECORD

FUNCTION start_list1()
  DEFINE i INTEGER
  IF ui.Window.forName("w_list1") IS NULL THEN
    FOR i=1 TO 10
      LET arr[i].id = i
      LET arr[i].name = "Record "||i
    END FOR
    OPEN WINDOW w_list1 WITH FORM "simple_list"
    START DIALOG control_list1
  ELSE
    CURRENT WINDOW IS w_list1
  END IF
END FUNCTION

FUNCTION terminate_list1()
  TERMINATE DIALOG control_list1
  CLOSE WINDOW w_list1
END FUNCTION

DIALOG control_list1()
  DISPLAY ARRAY arr TO sr.*
  ON ACTION add_row
    CALL DIALOG.appendRow("sr")
    LET arr[arr.getLength()].id = arr.getLength()
    LET arr[arr.getLength()].name = "[new record]"
  ON ACTION close
    CALL terminate_list1()
  END DISPLAY
END DIALOG
```
The module "list2.4gl" (quite the same code as list1.4gl):

```plaintext
DEFINE arr DYNAMIC ARRAY OF RECORD
    id INTEGER,
    name VARCHAR(50)
END RECORD

FUNCTION start_list2()
    DEFINE i INTEGER
    IF ui.Window.forName("w_list2") IS NULL THEN
        FOR i=1 TO 10
            LET arr[i].id = i
            LET arr[i].name = "Record "||i
        END FOR
        OPEN WINDOW w_list2 WITH FORM "simple_list"
        START DIALOG control_list2
    ELSE
        CURRENT WINDOW IS w_list2
    END IF
END FUNCTION

FUNCTION terminate_list2()
    TERMINATE DIALOG control_list2
    CLOSE WINDOW w_list2
END FUNCTION

DIALOG control_list2()
    DISPLAY ARRAY arr TO sr.*
    ON ACTION clear_row
        INITIALIZE arr[arr_curr()].* TO NULL
    ON ACTION close
        CALL terminate_list2()
END DISPLAY
END DIALOG
```

Program source code:

```plaintext
IMPORT FGL list1
IMPORT FGL list2
MAIN
    OPTIONS INPUT WRAP
    CALL start_list1()
    CALL start_list2()
    WHILE fgl_eventloop()
        END WHILE
END MAIN
```

---

**User interface programming**

Describes how to program user interface and dialog instructions.

**Dialog programming basics**

This section describes basic dialog programming concepts.

**The model-view-controller paradigm**

The dynamic user interface architecture is based on the Model-View-Controller (MVC) paradigm.

The model defines the object to be displayed (typically the application data that is stored in program variables). The view defines the decoration of the model (how the model must be displayed to the screen, this is typically the form). The controller is the interactive instruction that implements the program code to handle the model.
Views are defined in the abstract user interface tree from compiled .42f forms loaded by programs. The program variables act as models, and you implement the controllers with interactive instructions, such as DIALOG or INPUT. Controllers also define action handlers that contain the program code to be executed when an action view is triggered.

Note: Normally the controllers are not intended to provide any decoration information, as that is the purpose of views. Over the course of the history of the language, however, some interactive instructions such as MENU define both the controller and some presentation information such as menu title, command labels, and comments. In this case, the runtime system automatically creates the view with that information; you can still associate other views to the same controller.

Related concepts

The dynamic user interface on page 1105
The dynamic user interface is the base concept of the Genero user interaction components.

The abstract user interface tree on page 1106
The abstract user interface tree is the XML representation of the application forms displayed to the end user.

What are dialog controllers?
Application forms are controlled by interactive instruction blocks called dialogs. These blocks perform the common tasks associated with the form, such as field input and action handling.

The interactive instructions allow the program to respond to user actions and data input.

Simple display (non-interactive)

Instructions such as DISPLAY BY NAME display program variable data in the fields of a form and continue the program flow without giving control to the end user. This is in fact not an interactive instruction, as it just displays data to the current form, and returns immediately. However, it may be used in interactive instructions to display information to the end user.

Note: When using the UNBUFFERED mode of a dialog, you do not need to use the DISPLAY BY NAME instruction to synchronize program variables and form fields.

The MESSAGE and ERROR instructions are also simple display instructions without user interaction. These instructions are typically used to display a warning message to the end user.

Modal dialogs and parallel dialogs

Interactive instructions can be implemented as modal or parallel dialogs. Modal dialogs control a given window, and that window closes when the dialog is accepted or canceled. The window displays on top of any existing windows which are not accessible while the modal dialog executes. Parallel dialogs allow access to several windows simultaneously; the user can switch from one window to the other. Parallel dialogs are mainly used to implement split views on mobile platforms.

The interactive dialog blocks

The singular MENU instruction handles a list of choices to activate a specific function of the program. No field input is possible with this instruction. The user can only select an action from the list.

The singular INPUT instruction is designed for simple record input. It enables the fields in a form for input, waits while the user types data into the fields, and proceeds after the user accepts or cancels the dialog.

The singular DISPLAY ARRAY instruction is used to browse a list of records. It allows the user to view the contents of a program array of records, scrolling the record list on the screen and choosing a specific record. DISPLAY ARRAY implements by default a read-only list of records, but can be extended to become a modifiable list with list modification triggers such as ON INSERT.

The singular INPUT ARRAY instruction supports record list input. It allows the user to alter the contents of records of a program array, and to insert and delete records.
The singular **CONSTRUCT** instruction is designed to let the user enter search criteria for a database query. The user can enter a value or a range of values for one or several form fields, and your program looks up the database rows that satisfy the requirements.

The **procedural DIALOG** instruction (placed in the program flow) allows you to combine several **INPUT**, **DISPLAY ARRAY**, **INPUT ARRAY** and **CONSTRUCT** functionality within the same form.

The **declarative DIALOG** block (defined in a module at the same level as a function) allows you to implement individual **MENU**, **INPUT**, **DISPLAY ARRAY**, **INPUT ARRAY** and **CONSTRUCT** functionality, that will perform in parallel on several forms, when used with the **START DIALOG** and **TERMINATE DIALOG** instructions. Declarative **DIALOG** blocks can also be associated with a procedural **DIALOG** instruction through the **SUBDIALOG** clause, it will then act as a procedural **DIALOG** sub-dialog. See also Using declarative dialogs on page 1655.

**Dynamic dialogs**

When the form structure and field definitions can only be determined at runtime, it is possible to implement Dynamic Dialogs on page 2004, using generic code that will control a form generated on the fly.

Dynamic dialogs can implement all types of dialogs like static dialogs do. However, this feature should only be used in specific cases where regular static dialog instructions cannot be used. Static dialog code is more readable than generic code implementing dynamic dialogs. A generic record selection from a list (zoom) is the perfect candidate for a dynamic dialog.

**Related concepts**

Form specification files on page 1237
Form specification files are the source files defining the layout and content of application forms.

**Dialog configuration with FGLPROFILE**

FGLPROFILE parameters can be used to configure dialog behavior.

By setting global parameters in FGLPROFILE, you can control the behavior of all dialogs of the program. These options are provided as global parameters to define a common pattern for all dialogs of your application. A complete description is available in the runtime configuration section.

List of FGLPROFILE entries affecting the behavior of dialogs:

1. **Dialog.fieldOrder** (only used by singular dialogs like **INPUT**)
2. **Dialog.currentRowVisibleAfterSort**

**The Dialog.fieldOrder entry**

```plaintext
Dialog.fieldOrder = [true|false]
```

The **Dialog.fieldOrder** FGLPROFILE entry defines the execution of **BEFORE FIELD** and **AFTER FIELD** triggers of intermediate fields.

When this parameter is set to **true**, as the end user moves to a new field with a mouse click, the runtime system executes the **BEFORE FIELD** and **AFTER FIELD** dialog control blocks of the input fields between the source field and the destination field. When the parameter is set to **false**, intermediate field triggers are not executed.

The **Dialog.fieldOrder** configuration parameter is ignored by the **DIALOG** multiple-dialog instruction or when using the **FIELD ORDER FORM** option in singular dialogs such as **INPUT**.

Do not use this feature for new developments: GUI applications allow users to jump from one field to any other field of the form by using the mouse. Therefore, it makes no sense to execute the **BEFORE FIELD** and **AFTER FIELD** triggers of intermediate fields in a graphical application.

**Important:** The default setting for the runtime system is **false**; while the default setting in FGLPROFILE for **Dialog.fieldOrder** is **true**. As a result, the overall setting after installation is **true**. To modify the behavior of intermediate field trigger execution, change the setting of **Dialog.fieldOrder** in FGLPROFILE to **false**, or use the **FIELD ORDER FORM** program option.
The Dialog.currentRowVisibleAfterSort entry

Dialog.currentRowVisibleAfterSort = \{true\,false\}

The Dialog.currentRowVisibleAfterSort FGLPROFILE entry controls the visibility of the current row after a sort in tables.

When this parameter is set to true, the offset of table page is automatically adapted to show the current row after a sort. By default, the offset is not changed and current row may not be visible after sorting the rows of a table. Changing this parameter has no impact on existing code, it is just an indicator to force the dialog to shift to the page of rows having the current row, as if the end-user had used the scrollbar. You can use this parameter to get the same behavior as well known e-mail readers.

Related concepts
The FGLPROFILE file(s) on page 255
FGLPROFILE environment variable defines Genero BDL configuration files

The DIALOG control class
This topic explains the purpose of the ui.DIALOG class.

Inside a dialog instruction, the DIALOG predefined keyword represents the current dialog object. This dialog object can be used to execute methods provided by the ui.DIALOG built-in class.

For example, you can enable or disable an action with the setActionActive() dialog method, or you can hide or show the default action view with the setActionHidden() method:

```
BEFORE INPUT
    CALL DIALOG.setActionActive("zoom",FALSE)
AFTER FIELD field1
    CALL DIALOG.setActionHidden("zoom",TRUE)
```

The setFieldActive() method can be used to enable or disable a field during the dialog:

```
ON CHANGE custname
    CALL DIALOG.setFieldActive( "custaddr", 
        (rec.custname IS NOT NULL) )
```

The ui.Dialog class also provides methods to configure the dialog, for example to enable multiple row selection:

```
BEFORE DIALOG
    CALL DIALOG.setSelectionMode( "sr1", 1 )
```

Note:
When using methods of the ui.Dialog class that alter the data model or change the current field or row, control blocks like BEFORE FIELD, AFTER ROW, BEFORE DELETE are not executed: These are only fired to detect and control end user activity. Program code is considered as part of the dialog implementation. For example, methods such as ui.Dialog.deleteRow() must not execute BEFORE DELETE/AFTER DELETE control blocks. These control blocks are only fired by an end-user "delete" action.

Related concepts
Dynamic Dialogs on page 2004
Dialogs can be created at runtime with the ui.Dialog class.

Dialog control functions
The language provides several built-in functions and operators to be used in a dialog instruction.

Use the dialog functions and operators to keep track of the relative states of the current row, the program array, and the screen array, or to access the field buffers and keystroke buffers.
Typical control functions used in dialogs are: `arr_curr()`, `arr_count()`, `fgl_set_arr_curr()`, `set_count()`, `field_touched()`, `GET_FLDBUF()`, `INFIELD()`, `fgl_dialog_getfieldname()`, `fgl_dialog_getbuffer()`.

As an alternative to functions and operators (especially for those taking hard-coded parameters such as `INFIELD()`), use the methods provided in the `ui.Dialog` class.

**Refreshing the display when processing**

This topic explains when to use the `ui.Interface.refresh()` method.

The AUI tree on the front-end is synchronized with the runtime system AUI tree, when a user interaction instruction (dialog) gives the control back to the end user.

This means that the end user will not see any changes on the screen, as long as the program is doing batch processing, until an interactive statement is reached and waits for an event triggered by the end user.

To show something on the screen while the program is running in a batch procedure, force AUI tree synchronization by calling the `ui.Interface.refresh()` method.

**Important:** The AUI tree is automatically synchronized by the runtime system, when dialog instruction gives the control back to the end user. The `ui.Interface.refresh()` method must only be used in specific cases, to refresh the display while processing. For example, to show a "Please wait" message, or to implement a progress dialog window with a `PROGRESSBAR`. The `ui.Interface.refresh()` method should not be called more often than once in one second.

**Example**

Form file `form.per`:

```
LAYOUT
GRID
{ [pb ] [b1 ] }
END
END
ATTRIBUTES
PROGRESSBAR pb = FORMONLY.progress,
   VALUEMIN=0, VALUEMAX=30;
BUTTON b1: interrupt, TEXT="Interrupt";
END
```

Program file `main.4gl`:

```
MAIN
   DEFINE progress INTEGER
   DEFER INTERRUPT
   OPTIONS INPUT WRAP
   OPEN FORM f FROM "form"
   DISPLAY FORM f
   LET int_flag = FALSE
   FOR progress = 1 TO 30
      IF int_flag THEN
         EXIT FOR
      END IF
      DISPLAY BY NAME progress
      CALL ui.Interface.refresh()
      SLEEP 1
   END FOR
END MAIN
```
Related concepts

Dynamic Dialogs on page 2004
Dialogs can be created at runtime with the `ui.Dialog` class.

User interruption handling
Allow the end user to cancel the execution of a procedure in the program.

When do we need interruption handling?
If the program executes an interactive instruction, the GUI front-end can send action events based on user actions. When the program performs a long process like a loop, a report, or a database query, the front-end has no control. You might want to permit the user to stop a long-running process in such a case.

Detecting user interruptions in programs
To detect user interruptions coming from a GUI front-end, you define an action view with the name 'interrupt':

```plaintext
BUTTON sb: interrupt, TEXT="Stop";
```

When the runtime system takes control to process program code or execute a long running SQL query, the front-end automatically enables the local 'interrupt' action to let the user send an asynchronous interruption request to the program.

A program (the runtime system) can also receive a SIGINT interruption signal from the operating system. The interruption request that comes from the front-end is a different source. However, the runtime system handles both type of interruption events the same way.

When receiving an interrupt event from the front-end with an 'interrupt' special action, or from the system (SIGINT) the runtime system sets the `INT_FLAG` register to `TRUE`.

Consider using `DEFER INTERRUPT` and test the `INT_FLAG` register to properly handle user interruptions, and avoid immediate program termination. If the `DEFER INTERRUPT` instruction is not used, the program will stop immediately when an interruption event is caught. With `DEFER INTERRUPT`, the program continues, and can test `INT_FLAG` to check if an interruption event occurred. It is good practice to reset `INT_FLAG` to `FALSE` after detecting interruption:

```plaintext
WHILE ...
  IF INT_FLAG THEN
    LET INT_FLAG=FALSE
    ERROR "Procedure was interrupted by the user"
    EXIT WHILE
  END IF
...
END WHILE
```

SQL queries can be interrupted too, if the target database supports this feature. However, since the control is on the database server side while the SQL statement is running, it is not possible to execute program code to check `INT_FLAG`. In order to detect an SQL interruption, check the `SQLCA.SQLCODE` register after the query for SQL error -213, indicating that the last SQL statement was interrupted.

```plaintext
WHENEVER ERROR CONTINUE
-- Long running SQL statement
WHENEVER ERROR STOP
  IF SQLCA.SQLCODE == -213 THEN
    ERROR "Database query interrupted by user"
  ... IF
... END WHILE
```

When not using `DEFER INTERRUPT`, if the program enters in a long running procedure, a button with the action name 'interrupt' will become active. The user can then press that button, and the runtime system will stop the program,
since DEFER INTERRUPT is not used. However, this will not happen when a dialog is active, because the 'interrupt' button will be automatically disabled in that context. Such situation can confuse the end user, expecting that the 'interrupt' button can stop the program in any context.

Note that the front-end can not handle interruption requests properly if the display generates a lot of network traffic. In this case, the front-end has to process a lot of user interface modifications and has no time to detect a mouse click on the 'interrupt' action view. A typical example is a program doing a loop from 1 to 10000, just displaying the value of the counter to a field and doing a refresh. This would generate hundreds of AUI tree modifications in a short period of time. In such a case, we recommended that you calculate a modulo and display steps 10 by 10 or 100 by 100.

### Implementing interruption of a long running SQL query

```
-- db_busy.per
LAYOUT
GRID
{
    Database query in progress...
        [sb            ]
    }
END
END
ATTRIBUTES
    BUTTON sb: interrupt, TEXT="Stop";
END

MAIN
    DEFINE oc INT
    DEFER INTERRUPT
    OPTIONS SQL INTERRUPT ON
    DATABASE stores
    OPEN FORM f FROM "db_busy"
    DISPLAY FORM f
    CALL ui.Interface.refresh()
    WHENEVER ERROR CONTINUE
    SELECT COUNT(*) INTO oc FROM orders
    WHENEVER ERROR STOP
    IF SQLCA.SQLCODE == -213 THEN
        ERROR "Database query has been interrupted..."
    END IF
END MAIN
```

### Related concepts

- **Using SQL interruption** on page 601
  - Interrupt long running SQL queries, or interrupt queries waiting for locked data.

- **Configuring actions** on page 1744
  - Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with action attributes.

### Get program control if user is inactive

Execute some code after a given number of seconds, when the user does not interact with the program.

### When to use the ON IDLE trigger?

If an interactive instruction has the control, the program waits for a user interaction like an action or field input. If the end user leaves the workstation, or switches to another application, the program cannot get the control and is frozen until the user comes back. You might want to execute some code, after a period of inactivity, for example to refresh the displayed data by doing a new database query, or even after a long period, to terminate the program automatically.
**Implementing the ON IDLE trigger**

To detect user inactivity during a dialog, define an **ON IDLE** trigger in the dialog. This trigger is dialog specific, it is typically defined in the main dialog of the program, but it can also be defined in every dialog.

**Important:**

- Consider using the **ON IDLE** only in dialogs that do not handle field input, such as **DISPLAY ARRAY** and **MENU**. If used in input dialogs, this trigger may execute while the current field contains an incomplete value. The trigger will produce field value validation and raise an input error. However, **ON IDLE** can be used in input dialogs where the user cannot enter invalid values (for example when using **CHECKBOX**, **RADIOGROUP**, **COMBOBOX**, and character-type fields like **TEXTEDIT**).
- When implementing multiple or parallel dialogs with **DIALOG** block, do not mix several **ON IDLE** clauses in different sub-dialog blocks: specify a unique **ON IDLE** clause at the **DIALOG** block level.
- Obviously, it does not make much sense to mix **ON TIMER** and **ON IDLE** clauses.

For example:

```plaintext
DEFINE seconds SMALLINT
LET seconds = 120
DISPLAY ARRAY ...
   ...有用的 seconds
      ON IDLE seconds
      MESSAGE "Automatic data refresh..."
      -- Reload the array with a new database result set
...```

Note that the parameter of the **ON IDLE** trigger can be an integer variable, but it will only be read when the dialog is started. Changing the variable during dialog execution will have no effect.

A value of zero or less than zero disables the timeout trigger.

**Related concepts**

- Get program control on a regular (timed) basis on page 1713
- Execute some code after a given number of seconds, with or without user interaction with the program.

**Get program control on a regular (timed) basis**

Execute some code after a given number of seconds, with or without user interaction with the program.

**When to use the ON TIMER trigger?**

In some cases, an application needs to execute code at a regular interval, for example to process a message arrived in a queue, refresh data on a dashboard, or display resources in time-based graphs.

**Important:** Unlike the **ON IDLE** trigger which executes when there is no user activity, the **ON TIMER** trigger executes even when the user interacts with the application. Therefore, the code executed in an **ON TIMER** trigger must perform quickly, otherwise the end user will experience poor performance. As a general rule, make sure the time spent in the **ON TIMER** code is less than the timer interval. For example, if the processing time takes about 2 seconds, it doesn't make sense to have an **ON TIMER** that triggers every second.

**Implementing the ON TIMER trigger**

To return control to the program on regular intervals, use the **ON TIMER seconds** trigger in dialogs. This trigger is dialog specific. It is typically defined in the main dialog of the program, but it can be defined in every dialog.

**Important:**

- Consider using the **ON TIMER** only in dialogs that do not handle field input, such as **DISPLAY ARRAY** and **MENU**. If used in input dialogs, this trigger may execute in the middle of a field input, which can produce field value validation and raise an input error. However, **ON TIMER** can be used in input dialogs where the user cannot enter invalid values (for example when using **CHECKBOX**, **RADIOGROUP**, **COMBOBOX**, and character-type fields like **TEXTEDIT**).
• When implementing multiple or parallel dialogs with DIALOG block, do not mix several ON TIMER clauses in different sub-dialog blocks: Specify a unique ON TIMER clause at the DIALOG block level. Obviously, it does not make much sense to mix ON TIMER and ON IDLE clauses.

For example:

```plaintext
DEFINE seconds SMALLINT
LET seconds = 120
DISPLAY ARRAY ...
...
ON TIMER seconds
    MESSAGE "Check for messages in queue..."
    -- Query the message server for new messages.
...
```

Note that the parameter of the ON TIMER trigger can be an integer variable, but it will only be read when the dialog is started. Changing the variable during dialog execution will have no effect.

A value of zero or less than zero disables the timeout trigger.

**Related concepts**

- Get program control if user is inactive on page 1712

Execute some code after a given number of seconds, when the user does not interact with the program.

**Front-End function calls**

The language allows to execute specific functions on the front-end platform.

A set of built-in front calls is available in Genero front-ends, to execute a specific task on the platform where the front-end executes.

In order to perform a front call, use the `ui.Interface.frontCall()` method. For example, when using a mobile front-end, you can instruct the mobile device to take a picture and return the identifier of the asset containing the image:

```plaintext
DEFINE path STRING
CALL ui.Interface.frontCall( "mobile", "takePhoto", [], [path] )
```

For a complete list of available front calls, see Built-in front calls on page 2635.

**Input fields**

Describes various concepts related to form field management in dialogs.

**Field data type**

Depending on the type of dialog, the field data type is defined by program variables or form specification file.

**Field data types**

The field data type defines how a user can input values into a form field. For example, when defining a field to hold date values, it should only allow date value input. Therefore,

The data type used by the runtime to control field display and input depends on the type of the dialog:

- For INPUT, INPUT ARRAY and DISPLAY ARRAY, the data type is defined by the program variable bound to the field.
- For CONSTRUCT, the data type is defined in the (.per) form definition file, either by the TYPE attribute of a FORMONLY field for example, or with the corresponding database column in a schema file.

**Field validation rules**

Data validation rules can be defined at the form level, such as NOT NULL, REQUIRED and INCLUDE attributes.
Data validation constraints are checked when leaving a field, or when the dialog is validated (for example, with the ACCEPT DIALOG instruction inside a DIALOG multiple dialog block).

**Note:** Trailing blanks entered by the user will be removed when leaving the input field.

**Related concepts**

- **Binding variables to form fields** on page 1715
  Some dialogs need program variables to store form field values.

- **Input length of form fields** on page 1716
  Field input length defines the amount of characters the user can type in a form field.

- **Field configuration and decoration** on page 1718
  Form fields can be customized with specific decoration and settings.

**Binding variables to form fields**
Some dialogs need program variables to store form field values.

**Program variables are the data model for dialog instructions**

Dialogs handling data fields input or display (INPUT, INPUT ARRAY, DISPLAY ARRAY) need program variables to store the information displayed in form fields during the dialog execution. The exception is CONSTRUCT, which needs only one string variable that holds the SQL condition produced.

When implementing a dialog controlling form fields, you specify what program variables must be bound to the form fields of the current displayed form:

```
INPUT BY NAME custrec.* ...
... END INPUT
```

Program variables (simple records and arrays) used in dialogs can have a flat definition, or structured definition with sub-records.

**Dialog variables binding methods**

There are different ways to bind program variables to screen record fields.

Program variables can be bound to form fields by name or by position, depending on the binding clause used in the dialog definition.

**Note:** When binding program variables with a screen record followed by a .* (dot star), program variables are bound to screen record fields by position. Make sure that the program variables are defined (or listed) in the same order as the screen array fields. This is true for INPUT, DISPLAY ARRAY and INPUT ARRAY.

For more details about binding methods, see dialog-specific syntax reference topics in **Dialog instructions** on page 1452.

**Variables and form fields data types**

The program variables can be of any simple data type supported by the dialogs; the runtime system will adapt input and display rules to the variable type. When the user enters data for an INPUT or INPUT ARRAY instruction, the runtime system checks the entered value against the data type of the variable, not the data type of the form field. For example, if you want to use a DATE variable, the dialog will check for a valid date value when the user enters a value in the corresponding form field.

With CONSTRUCT, no program variable is used for fields. Only one string variable is required, to hold the generated SQL condition.

**Note:** The CONSTRUCT dialog uses the field data types defined in the form file.

Program variables are typically declared with a DEFINE LIKE clause to get the data type of a column as defined in the database schema file. When the form fields are also defined like a column of the database schema, this ensures
that the program variable and form field data type matches the underlying database column type. If a variable is
declared LIKE a SERIAL / SERIAL8 / BIGSERIAL column, the runtime system will treat the field as if it was
defined as NOENTRY in the form file. Since values of serial columns are automatically generated by the database
server, no user input is required for such fields.

**Note:** If the program variable has the same structure as a database table (this is the case when the variable is defined
with a DEFINE LIKE clause), you may want to use only some of the fields in your form layout. You can achieve
this by using PHANTOM fields in the screen array definition of the form. Phantom fields are only used to bind program
variables, and are not transmitted to the front-end for display.

**Related concepts**
- Binding tables to arrays in dialogs on page 1857
- Program arrays act as data model that are bound to form tables, when implementing list dialogs.
- Screen records / arrays on page 1251
- Form fields can be grouped in a screen record or screen array definition.

**Input length of form fields**
Field input length defines the amount of characters the user can type in a form field.

**Input length basics**
The field input length is used by interactive instructions to limit the size of the data that can be entered by the user.
The field input length also matters when displaying a program variable to a form field with the DISPLAY TO or
DISPLAY BY NAME instruction, that may truncate the text resulting from the data conversion.
The field input length is also used to define the input limit when using the AUTONEXT attribute.

**Note:** For non-character types, if the resulting text does not fit into the input length, the field will show * stars to
indicate an overflow.

**Input length definition**
The field input length is defined from:

1. The type of layout (grid-based or stack-based layout)
2. The data type of the program variable bound to the field by the interactive instruction.
3. In grid-based layout:
   - The size (number of cells) of the form item tag.
   - The usage of the SCROLL attribute for CHAR/VARCHAR/STRING types.

**Trailing blanks in input fields**
When the user enters data in a form field, the runtime system automatically truncates the trailing blanks.
The resulting value in the corresponding program variable or in the field input buffer
(DIALOG.getFieldBuffer()) is right-trimmed.

**Length semantics and input length**
Character string input limit depends also from the Byte Length Semantics (BLS) or Char Length Semantics (CLS)
usage.

**Note:** In this section, the term "character width" relates to the horizontal sizes of characters, especially in a fixed font
size. Typically, the width of an Chinese character is 2, while characters of occidental languages usually have a width
of 1. For more details, see Character size unit and length semantics on page 517.

**Using Byte Length Semantics**
When using BLS (the default), the input length represents the maximum number of bytes that the field can hold, in
the current character set of the runtime system.
For example:

- When using a Single Byte Character Set (SBCS) like ISO-8859-15 (and BLS): Each characters of this codeset uses one byte (and has a width of 1). A field with an input length of 5 cells can hold 5 characters of this codeset.
- When using a Chinese BIG5 encoding (and BLS): Latin characters (a,b,c) use one byte each, while Chinese characters use 2 bytes. A field with an input length of 5 cells can hold:
  - 5 Latin characters ($5 \times 1B = 5B$),
  - 3 Latin characters and 1 Chinese character ($3 \times 1B + 1 \times 2B = 5B$),
  - 1 Latin character and 2 Chinese characters ($1 \times 1B + 2 \times 2B = 5B$),
  - it cannot hold 3 Chinese characters ($3 \times 2B = 6B > 5B$).
- When using UTF-8 and BLS (not recommended, use CLS with UTF-8): ASCII characters (like "e") use one byte, Latin acute characters (like "é") use 2 bytes and Chinese characters use 3 bytes. A field with an input length of 5 cells can hold:
  - 5 ASCII characters ($5 \times 1B = 5B$),
  - 1 ASCII character and 2 Latin acute characters ($1 \times 1B + 2 \times 2B = 5B$),
  - 2 ASCII characters and 1 Chinese character ($2 \times 1B + 1 \times 3B = 5B$),
  - it cannot hold 3 Chinese characters ($3 \times 3B = 9B > 5B$).

### Using Char Length Semantics

When using character length semantics (FGL_LENGTH_SEMANTICS=CHAR), the input length is expressed in character width. The runtime system will truncate the entered text by computing the total width of the string. Trailing characters that do not fit into the field input length (interpreted as a maximum width) will be truncated when moving to another field or when validating the dialog instruction.

For example:

- When using UTF-8 and CLS: ASCII characters (like "e") use one byte, Latin acute characters (like "é") use 2 bytes and Chinese characters use 3 bytes. A field with an input length of 5 cells can hold:
  - 5 Latin characters ($5 \times 1W = 5W$),
  - 3 Latin characters and one Chinese character ($3 \times 1W + 1 \times 2W = 5W$),
  - 2 Chinese characters ($2 \times 2W = 4W < 5$),
  - it cannot hold 3 Chinese characters ($3 \times 2W = 6W > 5$).

### Field width definition in grid-based containers

In a grid-based container, by default the input length is defined by the width of the field item tag in the LAYOUT section. The width of a field item tag is defined by the number of cell positions used between the square brackets:

```plaintext
LAYOUT GRID
{
  [f1 ] -- width = 3 cells
  [f2 ] -- width = 6 cells
  ...
}
```

As a general rule, forms must define fields that can hold all possible values that the corresponding program variable can contain. For example, a DATE field must be defined with 10 cells, to hold date values in the format DD/MM/YYYY.

If the program variable is defined with a numeric data type like INTEGER or DECIMAL, the input length is defined by the form field width. For example, if the form item tag defines a width of 5 cells and is bound to an INTEGER variable by the input dialog, even if the integer variable can hold larger values, the user can only enter 5 digits or the negative sign and 4 digits. As result, the value range will be -9999 to 99999.

If the program variable is defined with a DATE, DATETIME or INTERVAL data type, the input length is defined by the form field width. The user can potentially enter any kind of characters. However, when the date/time field is checked by the dialog instruction, it must represent a valid date/time value.
If the program variable is defined with character data type such as CHAR, VARCHAR or STRING, by default, the input length is defined by the form field width. The SCROLL attribute can be used to bypass this limit and force the input length to be as large as the program variable. For example, when using a CHAR(20) variable with a form field defined with width of 6, the maximum input length will be a width of 20 instead of 6.

**Note:** Using the SCROLL attribute must be an exception: it is recommended to allow a size for form fields to hold all possible characters that fit in the corresponding program variable. Note also that for specific item types like TEXTEDIT, the SCROLL attribute behavior is implicit when the element is stretchable or allows scrollbars.

**Field width definition in stack-based containers**
In a stack-based layout, the input length is defined by the data type of the program variable.

In this example, the cust_id field will allow numeric input length in the range of the INTEGER data type, and the cust_name field will allow up to 50 characters:

```
-- Form file
LAYOUT
  STACK
    EDIT customer.cust_id;
    EDIT customer.cust_name;
...

-- Program
MAIN
  DEFINE cust_rec RECORD
    cust_id INTEGER,
    cust_name VARCHAR(50)
  END RECORD
...
  INPUT BY NAME cust_rec.*
...
```

If the program variable is defined with a numeric data type like INTEGER or DECIMAL or a character data type such as CHAR, VARCHAR or STRING, the input length is defined by the value range of the program variable. For numeric values, you can use the INCLUDE attribute to define the range of possible values.

If the program variable is defined with a DATE, DATETIME or INTERVAL data type, the input length is defined by the data type. For example a DATE field allows 10 characters.

**Related concepts**
- **Length semantics settings** on page 521
- **FORMAT attribute** on page 1366
  The FORMAT attribute defines the data formatting of numeric and date time fields, for input and display.
- **SAMPLE attribute** on page 1389
  The SAMPLE attribute defines the text to be used to compute the width of a form field widget.

**Field configuration and decoration**
Form fields can be customized with specific decoration and settings.

**Data formatting in fields**
Data format for input and display of numeric (DECIMAL, INTEGER) and DATE fields can be defined with the FORMAT attribute.

A default data format can be defined with environment variables (DBDATE, DBFORMAT, etc)

**Forcing the input pattern**
User input can be controlled with the PICTURE attribute, to force alpha-numeric or numeric characters at a given position. This is typically used for formatted data, such as credit card numbers or vehicle identification numbers.
Form field display size
If the default size of the form field widget is too large, provide a text example with the SAMPLE attribute, to let the front-end compute a specific size.

Define field description
Form field description can be displayed to the user with the COMMENT attribute.

Hide text for password input
In order to hide characters typed by the user, define the INVISIBLE attribute in the form field.

Force user to input value twice
When the VERIFY attribute is define for a form field, the user must enter the value twice before leaving the field.

Force uppercase or lowercase input
To force uppercase input, use the UPSHIFT attribute, and to force lowercase use the DOWNSHIFT attribute.

Default form field values
When the input dialog is not using WITHOUT DEFAULTS clause, the default value of a form field can be set with the DEFAULT attribute.

Tabbing order of form fields
To bypass the default tabbing order, use the TABINDEX attribute, as described in Defining the tabbing order on page 1729.

Value input limits, boolean correspondence and steps
Depending on the form field item type, control the value limits with VALUEMIN / VALUEMAX attributes (SPINEDIT, SLIDER), the incremental and decremental gap with STEP, and boolean correspondence with VALUECHECKED / VALUEUNCHECKED (CHECKBOX).

Form fields with button
To identify the action to be fired typically in a BUTTONEDIT field, define the ACTION attribute.

Defining the TABLE/TREE column TITLE
With form fields used in a list container like TABLE / TREE, it is possible to define the column title with the TITLE attribute.
Consider using a %"ident" localized string.

Control TABLE/TREE columns resize, visibility and sort selection
Form fields used in a TABLE / TREE list container are by default resizable, hidable and can be used for sorting. This can be denied with UNSIZABLE, UNHIDABLE and UNSORTABLE attributes.

CHECKBOX caption with TEXT
With form fields defined as CHECKBOX elements, labels can be defined by the TEXT attribute.
Consider using a %"ident" localized string.
The KEYBOARDHINT attribute
Especially for mobile devices, consider using the KEYBOARDHINT attribute, to get a specific keyboard for phone numbers, emails, URLs, etc.

Using a presentation STYLE
Additional configuration and decoration can be define for a form field with the STYLE attribute, pointing to style attributes defined in .4st files. Form field text and background colors are a typical decoration that can be achieved with styles.

COMBOBOX and RADIOGROUP items
The selection items of a COMBOBOX or RADIOGROUP form field are defined with the ITEMS attribute.
Consider using %"ident" localized strings for item labels.

Related concepts
Localized strings on page 538
Localized strings provide a means of writing applications in which the text of strings can be customized on site.
Field data type on page 1714
Depending on the type of dialog, the field data type is defined by program variables or form specification file.

The buffered and unbuffered modes
The buffered and unbuffered mode control the synchronization of program variables and form fields.

Data model / view / controller paradigm
When bound to an interactive instruction (such as dialog), program variables act as a data model to display data or to get user input. To change the values of form fields by program code, the corresponding variables must be set and displayed.
Synchronization of program variables with the form fields depends on the buffer mode used by the dialog. Use the unbuffered mode to get automatic data model / form field synchronization.

Configuring the buffer mode
By default, singular dialogs (INPUT, DISPLAY ARRAY) and procedural DIALOG blocks use the buffered mode, while parallel dialogs use the unbuffered mode by default.
The unbuffered mode can be set per (modal) dialog instruction, with the UNBUFFERED dialog attribute:

```
INPUT BY NAME p_site.* ATTRIBUTES(UNBUFFERED)
... END INPUT
```
When using a procedural DIALOG block, all sub dialogs defined locally or included with the SUBDIALOG clause inherit the buffer mode of the parent procedural dialog block:

```
DIALOG ATTRIBUTES(UNBUFFERED)
  INPUT BY NAME p_site.* -- unbuffered
  ...
  END INPUT
  DISPLAY ARRAY a_events TO sr_events.* -- unbuffered
  ...
  END DISPLAY
  SUBDIALOG d_comments -- unbuffered
END DIALOG
```
The unbuffered mode can also be set globally with the `ui.Dialog.setDefaultUnbuffered()` method, for singular and procedural dialogs:

```plaintext
CALL ui.Dialog.setDefaultUnbuffered(TRUE)
...
INPUT BY NAME rec_cust.* WITHOUT DEFAULTS -- uses unbuffered mode
...
END INPUT
```

In contrast with modal dialogs described above, when implementing parallel dialogs, all started dialogs implicitly use the unbuffered mode, and it is not possible to use the buffered mode:

```plaintext
DIALOG d_customers()
   INPUT BY NAME r_cust.*
   ...
   END INPUT
   END DIALOG
   ...
START DIALOG d_customers  -- will be unbuffered by default
   ...
```

### The buffered mode

When you use the default "buffered" mode, program variable changes are not automatically displayed to form fields; you need to execute `DISPLAY TO` or `DISPLAY BY NAME`. Additionally, if an action is triggered, the value of the current field is not validated and is not copied into the corresponding program variable. The only way to get the text of a field is to use `GET_FLD_BUF()` or `DIALOG.getFieldBuffer()`. These functions return the current text, which might not be a valid representation of a value of the field data type:

```plaintext
INPUT BY NAME p_item.*
   ON ACTION zoom
      CALL select_item()
      RETURNING p_item.code, p_item.desc
      DISPLAY BY NAME p_item.code, p_item.desc
      END IF
   ...
END INPUT
```

### The unbuffered mode

With the unbuffered mode, program variables and form fields are automatically synchronized, and the dialog instruction is sensitive to program variable changes: You don't need to display values explicitly with `DISPLAY TO` or `DISPLAY BY NAME`. When an action is triggered, the value of the current field is validated and is copied into the corresponding program variable. If you need to display new data during the dialog execution, just assign the values to the program variables; the runtime system will automatically display the values to the screen after user code of the current control or interaction block has been executed:

```plaintext
INPUT BY NAME p_site.* ATTRIBUTES(UNBUFFERED)
   ON ACTION zoom
      CALL select_item()
      RETURNING p_item.code, p_item.desc
      -- no need to display desc.
      END IF
   ...
END INPUT
```
**Actions configuration for field validation**

During data input, values entered by the user in form fields are automatically validated and copied into the program variables. Actually the value entered in form fields is first available in the form field buffer. This buffer can be queried with built-in functions or dialog class methods. With the unbuffered mode, the field buffer is used to synchronize program variables each time control returns to the runtime system - for example, when the user clicks on a button to execute an action.

With the unbuffered mode, data validation must be prevented for some actions such as cancel or close. To avoid field validation for a given action, set the `validate` action default attribute to "no", in the `.4ad` file or in the ACTION DEFAULTS section of the form file:

```
ACTION DEFAULTS
    ACTION undo (TEXT = "Undo", VALIDATE = NO)
```

**Note:** Actions such as `dialogtouched`, `cancel`, `delete`, `close`, `help` are by default defined with the `validate=no` attribute in the `$FGLDIR/lib/default.4ad` file.

If field validation is disabled for an action, the code executed in the ON ACTION block acts as if the dialog was in buffered mode. The program variable is not set. However, the input buffer of the current field is updated. When returning from the user code, the dialog will not synchronize the form fields with program variables, and the current field will display the input buffer content. Therefore, if you change the value of the program variable during an ON ACTION block where validation is disabled, you must explicitly display the values to the fields with DISPLAY TO / BY NAME.

To illustrate this case, imagine that you want to implement an undo action to allow the modifications done by the user to be reverted (before these have been saved to the database of course). You typically copy the current record into a clone variable when the dialog starts, and copy these old values back to the input record when the undo action is invoked. An undo action is a good candidate to avoid field validation, since you want to ignore current values. If you don't re-display the values, the input buffer of the current field will remain when returning from the ON ACTION block:

```
DIALOG ATTRIBUTES(UNBUFFERED)
    INPUT BY NAME p_cust.*
    BEFORE INPUT
        LET p_cust_copy.* = p_cust.*
        ON ACTION undo -- Defined with VALIDATE=NO
            LET p_cust.* = p_cust_copy.*
            DISPLAY BY NAME p_cust.*
    END INPUT
END DIALOG
```

For more details, see Data validation at action invocation on page 1762.

**Related concepts**

- Binding variables to form fields on page 1715
  Some dialogs need program variables to store form field values.

**Form field initialization**

Form field initialization can be controlled by the WITHOUT DEFAULTS dialog option.

The INPUT and INPUT ARRAY dialogs provide the WITHOUT DEFAULTS option to use program variable values when the dialog starts, or to apply the DEFAULT attribute defined in forms. The semantics of this option is slightly different in INPUT and INPUT ARRAY dialogs. Use of the WITHOUT DEFAULTS clause is always recommended in INPUT ARRAY.
The WITHOUT DEFAULTS option can be used in the binding clause or as an ATTRIBUTES option. When used in the binding clause, the option is defined statically at compile time as TRUE. When used as an ATTRIBUTES option, it can be specified with an integer expression that is evaluated when the DIALOG interactive instruction starts:

```plaintext
INPUT BY NAME p_cust.* ATTRIBUTES (WITHOUT DEFAULTS = NOT new)
... END INPUT
```

### The WITHOUT DEFAULTS clause in INPUT

In the default mode, an INPUT clears the program variables and assigns the values defined by the DEFAULT attribute in the form file (or indirectly, the default value defined in the database schema files). This mode is typically used to input and INSERT a new record in the database. The REQUIRED field attributes are checked to make sure that the user has entered all data that is mandatory. Note that REQUIRED only forces the user to enter the field, the value can be NULL unless the NOT NULL attribute is used. Therefore, if you have an AFTER FIELD or ON CHANGE control block with validation rules, you can use the REQUIRED attribute to force the user to enter the field and trigger that block.

In contrast, the WITHOUT DEFAULTS option starts the INPUT dialog with the existing values of program variables. This mode is typically used in order to UPDATE an existing database row. Existing values are considered valid, thus the REQUIRED attributes are ignored when this option is used.

The NOT NULL field attribute is always checked at dialog validation, even if the WITHOUT DEFAULTS option is set.

### The WITHOUT DEFAULTS clause in INPUT ARRAY

With an INPUT ARRAY, the WITHOUT DEFAULT option defines whether the program array is populated when the dialog begins. Once the dialog is started, existing rows are always handled as records to be updated in the database (WITHOUT DEFAULTS=TRUE), while newly created rows are handled as records to be inserted in the database (WITHOUT DEFAULTS=FALSE). In other words, column default values defined in the form specification file or the database schema files are only used for newly-created rows.

It is unusual to implement an INPUT ARRAY with no WITHOUT DEFAULTS option, because the program array would be cleared and the list would appear empty.

**Important:** The default in INPUT ARRAY used inside DIALOG is WITHOUT DEFAULTS=TRUE, but in a singular INPUT ARRAY dialog, the default is WITHOUT DEFAULTS=FALSE.

### Related concepts

- **Database schema** on page 467
  Defines database table structures with column type information to be reused in program variable definitions.

- **Form specification files** on page 1237
  Form specification files are the source files defining the layout and content of application forms.

### Input field modification flag

Each input field controlled by a dialog instruction has a modification flag.

The modification flag is used to execute form-level validation rules and trigger ON CHANGE blocks. The flag can also be queried to detect if a field was touched during the DIALOG instruction, for example with the FIELD_TOUCHED () operator or with ui.Dialog.getFieldTouched().

Both FIELD_TOUCHED() and ui.Dialog.getFieldTouched() accept a list of fields and/or the screen-record.* notation in order to check the modification flag of multiple fields in a unique function call. Use a single * star character, to reference all fields used by the dialog.

The modification flag is set to TRUE when the user enters data in a field, or when the program code executes a DISPLAY TO, DISPLAY BY NAME, CLEAR field-spec or CLEAR SCREEN ARRAY screen-array.* instruction. The modification flags can also be set by program to TRUE or reset to FALSE with the
ui.Dialog.setFieldTouched() method, to emulate user input, or to reset the modification flags after data is saved in the database.

The modification flags of all fields are automatically reset to FALSE by the interactive instruction in the following cases:

- When the dialog instruction starts.
- In a DIALOG block, when entering a group of fields controlled by an INPUT or a CONSTRUCT sub-dialog.
- When moving to (or creating) a new row in an INPUT ARRAY.
- Within a DISPLAY ARRAY, the modification flags are always TRUE for all fields.

When using a DISPLAY ARRAY, the modification flags are set to TRUE for all fields. This behavior exists because of backward compatibility. Since values cannot be modified by the user, the modification flags are not relevant in this dialog. However, you must pay attention when implementing nested dialogs, because DISPLAY ARRAY will set the modification flags of the fields driven by the parent dialog, for example when executing a DISPLAY ARRAY from an INPUT ARRAY.

Query the modification flags with the ui.Dialog.getFieldTouched() method, typically in the context of AFTER INPUT, AFTER CONSTRUCT, AFTER INSERT or AFTER ROW control blocks.

When using a list driven by an INPUT ARRAY binding, a temporary row added at the end of the list will be automatically removed if all fields have the modification flag set to FALSE.

For typical EDIT fields, the modification flag is set when leaving the field. If you want to detect data modification earlier, it is recommended that you use the dialogtouched predefined action. However, this event is only an indicator that the user started to modify a field, the value will not be available in the program variables.

**Related concepts**

Immediate detection of user changes on page 1725
This section describes the dialogtouched predefined action.

Reacting to field value changes on page 1724
This section describes the purpose of the ON CHANGE interaction block.

**Reacting to field value changes**

This section describes the purpose of the ON CHANGE interaction block.

The ON CHANGE interaction block can be used in different ways:

- With form fields allowing only entire value input such as CHECKBOX, or using an additional widget such as a calendar in a DATEEDIT: ON CHANGE can be used to detect an immediate value change, or the selection of a value in the additional widget, without leaving the field.
- With text fields like EDIT (allowing incomplete values), defined with the COMPLETER attribute to implement autocompletion: In this case the ON CHANGE trigger is used to fill the list of matching values, when the user types characters in (ON CHANGE is fired after a short delay).
- With text fields like EDIT (allowing incomplete values): ON CHANGE can be used to detect a value change, when the field is left.

A typically usage of ON CHANGE is for example with a CHECKBOX, to enable/disable other form elements depending on the value of the checkbox field:

```
INPUT BY NAME rec.* ...
...
ON CHANGE input_details -- can be TRUE or FALSE
   CALL DIALOG.setFieldActive("address1", rec.input_details)
   CALL DIALOG.setFieldActive("address2", rec.input_details)
...
END INPUT
```

**Important:** The dialogtouched predefined action can also be used to detect field changes immediately, but with this action you cannot get the data in the target variables. Use of this special action is only recommended to detect if the user has started to modify data in the current dialog.
Immediate detection of user changes
This section describes the dialogtouched predefined action.

Purpose of the dialogtouched action
The dialogtouched predefined action is a special action that can be used to detect user changes immediately without leaving the current field.

Important: The dialogtouched action must be enabled/disabled in accordance with the needs of the dialog: When this action is enabled, the ON ACTION dialogtouched block will be invoked each time the user types characters, modifies the value with copy/paste actions, or uses the widget input helper (like the calendar of a DATEEDIT). In client/server mode, this can generate more network traffic as needed. As soon as the dialogtouched action is fired, it should be disabled to avoid unnecessary network round-trips, and it should only be re-enabled when needed.

When the form item type allows value changes to be detected immediately, for example with COMBOBOX, CHECKBOX or DATEEDIT fields, the alternative to dialogtouched to detect field input changes is to use the ON CHANGE trigger.

Typical usage of dialogtouched action
Singular interactive instructions are typically ended with an accept or cancel action. For example, a singular INPUT statement allows the end user to enter a database record, and validate or cancel the changes. The INPUT statement is then re-executed, to enter or modify another record.

Unlike singular dialogs, the DIALOG instruction can be used continuously for several data operations, such as navigation, creation, or modification. Typically, the default is the navigation mode, and as soon as the user starts to modify a field, it switches to edit mode, to modify a record or create a new record. Only a user-defined close or exit action will terminate a DIALOG block. In such case, the dialog must be notified when the user starts to modify the current record. This can be achieved with the dialogtouched predefined action.

What user events fire a dialogtouched action?
The dialogtouched action works for any field controlled by the current interactive instruction, and with any type of form field.

Every time the user modifies the value of a field (without leaving the field), the ON ACTION dialogtouched block will be executed, if it is enabled.

The dialogtouched action can be triggered by typing characters in a text editor field, using copy/paste, clicking on a CHECKBOX/RADIOGROUP, moving the cursor of a SLIDER or changing a date with the calendar of a DATEEDIT.

Field value validation when dialogtouched occurs
When a dialogtouched action occurs, the current field may contain some text that does not represent a correct value corresponding to the field data type. For example, a form field bound to a DATE variable may contain only a part of a valid date string, such as "12/24/". For this reason, the target variable bound to the field cannot hold the current text displayed on the screen when the ON ACTION dialogtouched code is executed, even when using the UNBUFFERED mode.
To avoid data validation on action code execution, the `dialogtouched` action is defined with `validate="no"` attribute in the `$FGLDIR/lib/default.4ad` action defaults file. This is mandatory when using the UNBUFFERED mode; otherwise the runtime would try to copy the input buffer into the program variable when a `dialogtouched` action is invoked.

**Note:** Do not define `validate="yes"` for the `dialogtouched` action, otherwise non-string data fields will in most cases produce a conversion error, when the user enters data.

**Programming steps to handle a "save" button**

By default, the `dialogtouched` action and navigation actions are enabled.

In the `ON ACTION dialogtouched` block, detect the beginning of a record modification in a DIALOG block.

To prevent further `dialogtouched` action events, disable the action with a `DIALOG.setActionActive()` method, disable also navigation actions, and enable the `save` action.

Set a flag/status variable to track that the dialog is in edit mode.

When user input is validated and committed in the database, the `dialogtouched` and navigation actions can be enabled again, and the `save` action can be disabled.

Set a flag/status variable to indicate that the dialog is back to navigation mode.

In the action handler for the `close` or `exit` action, which can be used to close the form, check the status flag to know if the user has started to edit the fields, and show a warning box before leaving the dialog with `EXIT DIALOG`.

Code example:

```plaintext
DEFINE editing BOOLEAN

DIALOG
  ...
  ON ACTION dialogtouched
    CALL setup_dialog(DIALOG, TRUE)
  ...
  ON ACTION save
    CALL save_record()
    CALL setup_dialog(DIALOG, FALSE)
  
  ON ACTION close
    IF NOT check_close(DIALOG) THEN
      CONTINUE DIALOG
    END IF
    EXIT DIALOG
  ...
END DIALOG

FUNCTION setup_dialog(d, e)
  DEFINE d ui.Dialog, e BOOLEAN
  LET editing = e
  CALL DIALOG.setActionActive("dialogtouched", NOT editing)
  CALL DIALOG.setActionActive("save", editing)
  CALL DIALOG.setActionActive("query", NOT editing)
END FUNCTION

FUNCTION check_close(d)
  DEFINE d ui.Dialog
  IF editing THEN
    CALL setup_dialog(d, editing)
    RETURN mbox_yn("Do you want to close the form without saving changes?")
  END IF
  RETURN TRUE
```
Related concepts
The buffered and unbuffered modes on page 1720
The buffered and unbuffered mode control the synchronization of program variables and form fields.

Reacting to field value changes on page 1724
This section describes the purpose of the ON CHANGE interaction block.

Form-level validation rules
Form-level validation rules can be defined for each field controlled by a dialog.

Form-level validation can be specified at the form field level with attributes such as NOT NULL, REQUIRED and INCLUDE. These attributes are part of the business rules of the application and must be checked before saving data into the database.

Implicit validation rule checking
An INPUT or INPUT ARRAY block automatically executes form-level validation rules in the following cases:

• The NOT NULL attribute is satisfied if a value is in the field. NOT NULL is checked:
  • when the user moves to a different row in a list controlled by an INPUT ARRAY. However, if the row is temporary and none of the fields is touched, the attribute is ignored.
  • in a DIALOG block, when focus leaves the sub-dialog controlling the field.
  • in a DIALOG block, when NEXT FIELD gives the focus to a field in a different sub-dialog than the current sub-dialog.
  • when the dialog instruction is ended, for example when a procedural DIALOG is ended with ACCEPT DIALOG, or when an singular INPUT is ended with ACCEPT INPUT or with the implicit accept action.

• The REQUIRED attribute is satisfied if the field modification flag is true, if a DEFAULT value is defined, or if the WITHOUT DEFAULTS option is used. REQUIRED is checked:
  • when the user moves to a different row in a list controlled by an INPUT ARRAY. However, if the row is temporary and none of the fields is touched, the attribute is ignored.
  • in a DIALOG block, when focus leaves the sub-dialog controlling the field.
  • in a DIALOG block, when NEXT FIELD gives the focus to a field in a different sub-dialog than the current sub-dialog.
  • when the dialog instruction is ended, for example when a procedural DIALOG is ended with ACCEPT DIALOG, or when a singular INPUT is ended with ACCEPT INPUT or with the implicit accept action.

• The INCLUDE attribute is satisfied if the value is in the list defined by the attribute. INCLUDE is checked when the target program variable must be assigned. This happens:
  • when UNBUFFERED mode is used, focus is in the field, and an action is invoked.
  • when the focus leaves the field.
  • when the user moves to a different row in a list controlled by an INPUT ARRAY. However, if the row is temporary and none of the fields is touched, the attribute is ignored.
  • in a DIALOG block, when focus leaves the sub-dialog controlling the field.
  • in a DIALOG block, when NEXT FIELD gives the focus to a field in a different sub-dialog than the current sub-dialog.
  • when the dialog instruction is ended, for example when a procedural DIALOG is ended with ACCEPT DIALOG, or when a singular INPUT is ended with ACCEPT INPUT or with the implicit accept action.

Performing validation rules explicitly
Singular input dialogs (INPUT / INPUT ARRAY) create default accept / cancel actions. The form-level validation rules are typically performed when the implicit accept action is triggered.

The DIALOG procedural instruction can be used as in singular interactive instructions, with the typical OK / Cancel buttons (accept / cancel actions) to finish the instruction. The accept/cancel action handlers would respectively
execute the ACCEPT DIALOG and EXIT DIALOG instructions. This solution allows the user to input or modify one record at a time, and the program flow must reenter the DIALOG instruction to edit or create another record. Alternatively, the DIALOG instruction can let the user input / modify multiple records without leaving the dialog. In this case, you need a way to execute the form-level validation rules defined for each field, before saving the data to the database.

To validate a subset of fields controlled by the DIALOG instruction, use the `ui.Dialog.validate("field-list")` method, as shown in this example:

```plaintext
ON ACTION save
  IF DIALOG.validate("cust.*") < 0 THEN
    CONTINUE DIALOG
  END IF
  CALL customer_save()
```

This method automatically displays an error message and registers the next field in case of error. It is mandatory to execute a CONTINUE DIALOG instruction if the function returns an error.

Within singular input dialogs, form-level validation rules can also be explicitly performed with the ACCEPT INPUT instruction, or with the `DIALOG.validate("**")` API call, followed by a CONTINUE INPUT in case of error.

**Related concepts**
- Appending rows in INPUT ARRAY on page 1848
- Rows appended at the end of an editable list are temporary until they are edited.
- Form field initialization on page 1722
  - Form field initialization can be controlled by the WITHOUT DEFAULTS dialog option.
- ACCEPT DIALOG instruction on page 1640
- The buffered and unbuffered modes on page 1720
  - The buffered and unbuffered mode control the synchronization of program variables and form fields.
- Input field modification flag on page 1723
  - Each input field controlled by a dialog instruction has a modification flag.

**Form field deactivation**

The form fields bound to a dialog are by default active (they can get the focus). When needed, disable the fields that do not require user input, and reactivate them later during the dialog execution.

For example, imagine a form containing an "Industry" COMBOBOX field, with the options Healthcare, Education, Government, Manufacturing, and Other. If the user selects "Other", a secondary EDIT field is expected to be activated automatically, to let the user input the specific description of the industry. But if one of the predefined values is selected, there is no need for the additional field, so the secondary field can be left disabled.

This can be achieved by enabling/disabling fields with the `ui.Dialog.setFieldActive()` method depending on the context. The "Industry" field case described can be implemented as follows:

```plaintext
DIALOG ATTRIBUTES(UNBUFFERED)
  INPUT BY NAME rec.*
    ON CHANGE industry
      -- A value of 99 corresponds to the "Other" item
      CALL DIALOG.setFieldActive( "cust.industry", (rec.industry!=99) )
    ...
  END INPUT
BEFORE DIALOG
  CALL DIALOG.setFieldActive( "cust.industry", FALSE )
  ...
END DIALOG
```

Consider centralizing field activation / deactivation in a setup function specific to the dialog, passing the DIALOG object as parameter.
Do not disable all fields of a dialog, otherwise the dialog execution stops (at least one field must get the focus during a dialog execution).

If you disable the current field having the focus, the dialog will execute the \texttt{AFTER FIELD} block of the current field and the \texttt{BEFORE FIELD} block on the next field in the tabbing order. This can unnecessarily fire validation code implemented in \texttt{AFTER FIELD}. As a general pattern, do not disable the current field having the focus.

\textbf{Note:} Form fields can be hidden with the \texttt{ui.Form.setFieldHidden()} method of the form objects. A dialog considers hidden fields as disabled (there is no need to disable fields that are already hidden).

\textbf{Related concepts}

\texttt{COMBOBOX item type} on page 1270
Defines a line-edit with a drop-down list of values.

\texttt{EDIT item type} on page 1274
Defines a simple line-edit field.

\textbf{Identifying sub-dialogs in procedural DIALOG}

Sub-dialogs need to be identified by a name to distinguish the different contexts.

A procedural \texttt{DIALOG} block is a collection of sub-dialogs that act as controllers for different parts of a form. In order to program a procedural \texttt{DIALOG} block, there must be a unique identifier for each sub-dialog.

For example, to set the current row of a screen array with the \texttt{DIALOG.setCurrentRow()} method, you pass the name of the screen array to specify the sub-dialog to be affected. Sub-dialog identifiers are also used as a prefix to specify actions for the sub-dialog.

The following topics describe how to specify the names of the different types of \texttt{DIALOG} sub-dialogs:

- Identifying an \texttt{INPUT} sub-dialog on page 1596
- Identifying a \texttt{DISPLAY ARRAY} sub-dialog on page 1598
- Identifying an \texttt{INPUT ARRAY} sub-dialog on page 1599
- Identifying a \texttt{CONSTRUCT} sub-dialog on page 1597
- The \texttt{SUBDIALOG} clause on page 1600.

\textbf{Related concepts}

\texttt{Structure of a procedural DIALOG block} on page 1594

\texttt{The Dialog class} on page 2512
The \texttt{ui.Dialog} class provides a set of methods to configure, query and control the current interactive instruction.

\texttt{Binding action views to action handlers} on page 1762
How are action views of the forms bound to action handlers in the program code?

\textbf{Defining the tabbing order}

Control the order of tabbing through the fields with the \texttt{TABINDEX} attribute.

When a dialog is executing, the end-user can jump from field to field with the keyboard by using the Tab and Shift-Tab keys.

\textbf{Note:} One can tab out of an \texttt{INPUT ARRAY} sub-dialog with Ctrl-Tab and Shift-Ctrl-Tab accelerators (in \texttt{INPUT ARRAY}, Tab and Shift-Tab loop in the fields of the current row).

The order in which the fields can be visited with the Tab key can be controlled with a program option and the \texttt{TABINDEX} form field attribute.

The \texttt{FIELD ORDER} dialog attribute defines the way tabbing order works (it can also be defined globally with \texttt{OPTIONS FIELD ORDER mode}). Tabbing order can be based on the dialog binding list (\texttt{FIELD ORDER CONSTRAINED}, the default) or it can be based on the form tabbing order (\texttt{FIELD ORDER FORM}). It is recommended that you use the \texttt{FIELD ORDER FORM} option, to use the tabbing order specified in the form file.

The \texttt{TABINDEX} field attribute allows tabbing order in the form to be defined for each form item. By default, the form compiler assigns a tabbing index for each form item based on the position of the item in the layout.

\textbf{Important:} \texttt{TABINDEX} values must be unique in a form.
Form elements that can get the focus are:

- Simple form fields controlled by `INPUT` or `CONSTRUCT`.
- Read-only lists controlled by `DISPLAY ARRAY`.
- Editable list cells controlled by `INPUT ARRAY`.
- Simple buttons controlled by a `COMMAND` interaction block.

If you use the keyboard to tab into a form element, the focus will go to the next (or previous) element that is visible and activated. In other words, if a form item is hidden or disabled, it is removed from the tabbing list.

The tabbing position of a read-only list driven by a `DISPLAY ARRAY` binding is defined by the `TABINDEX` of the first field.

When `TABINDEX` is set to zero, the form item is excluded from the tabbing list. However, the item with `TABINDEX=0` can still get the focus with the mouse (or when you tap on it on a mobile device).

The `NEXT FIELD` instruction can also use the tabbing order, when executing `NEXT FIELD NEXT` and `NEXT FIELD PREVIOUS`.

**Note:** When moving columns around in a `TABLE` or `TREE` container, new tab indexes are assigned to columns, so that input tabbing order corresponds to the visual column order. If the order of the columns in an editable list shouldn't be changed, freeze the table columns with the `UNMOVABLECOLUMNS` attribute of the `TABLE/TREE`.

**Related concepts**

- Dialog configuration with FGLPROFILE on page 1708
- FGLPROFILE parameters can be used to configure dialog behavior.

### Which form item has the focus?

Identify what element of the current form has the focus.

Sometimes it is important to know which form element has currently the focus. This is especially important when implementing a `DIALOG` block, that can control several parts of a form. For example, when several lists are driven by multiple `DISPLAY ARRAY` sub-dialogs, you may need to know which is the current list.

To get the name of the current form item, use the `DIALOG.getCurrentItem()` method. This method is the replacement of the former `fgl_dialog_getfieldname()` built-in function. It has been extended to return identifiers for fields, lists, or actions identifiers.

```dialog
dialog attributes(unbuffered)
display array p_orders to orders.*
...
end display
display array p_items to items.*
...
end display
...
if dialog.getCurrentItem() == "items" then
  ...
end if
...
end dialog
```

It is also possible to detect when the focus enters or leaves a field or a group of fields by using control blocks such as `BEFORE INPUT/BEFORE DISPLAY` or `AFTER INPUT/AFTER DISPLAY`.

**Related concepts**

- The `DISPLAY ARRAY` sub-dialog on page 1597
- The `DISPLAY ARRAY` sub-dialog is the controller to implement the navigation in a list of records, with option data modification actions.

**Detection of focus changes** on page 1732
Describes how to detect when the focus goes from field to field or to a read-only list.

The buffered and unbuffered modes on page 1720
The buffered and unbuffered mode control the synchronization of program variables and form fields.

**Giving the focus to a form element**
How to force the focus to move or stay in a specific form element using program code.

Use the NEXT FIELD instruction to force the focus to a specific field or screen record (list). The NEXT FIELD instruction expects a form field name.

In a DIALOG block, when the specified field is the first column identifier of a sub-dialog driven by a DISPLAY ARRAY block, the read-only list gets the focus. If the field name is not known at compile time, you can alternatively use the `ui.Dialog.nextfield()` method.

```plaintext
DIALOG ATTRIBUTES(UNBUFFERED)
  INPUT BY NAME p_cust ATTRIBUTES(NAME="cust")
  ...
END DISPLAY
DISPLAY ARRAY p_orders TO orders.*
  ...
END DISPLAY
ON ACTION go_to_header
  NEXT FIELD cust_num
ON ACTION go_to_detail
  NEXT FIELD order_lineno
  ...
END DIALOG
```

When a BUTTON exists in the form layout, it can get the focus if the DIALOG block defines a COMMAND clause as action handler. Currently there is no way to give the focus to a BUTTON by program.

```plaintext
DIALOG ATTRIBUTES(UNBUFFERED)
  ...
  COMMAND "print"
    CALL print_order()
  ...
END DIALOG
```

In rare cases (especially when using folder tabs), it may be required to show a part of the form that is not controlled by the dialog, when there is no active field or button that can get the focus in that part of the form, and when the above techniques cannot work. In this case, in order to show temporarily a given part of the form that cannot get the focus, you use the `ui.Form.ensureFieldVisible` on page 2502 or `ui.Form.ensureElementVisible` on page 2501 methods.

```plaintext
DEFINE form ui.Form
  ...
DIALOG ATTRIBUTES(UNBUFFERED)
  ...
  BEFORE DIALOG
    LET form = DIALOG.getForm()
      ...
  ON ACTION show_image1
    CALL form.ensureElementVisible("image1")
      ...
END DIALOG
```

When using the FOCUSONFIELD attribute of DISPLAY ARRAY, you can set the focus to a specific cell by using the NEXT FIELD instruction on page 1497 instruction or the `ui.Dialog.nextField()` method, in conjunction with the `ui.Dialog.setCurrentRow()` method. For more details, see Field-level focus in DISPLAY ARRAY on page 1840.
Related concepts

The DISPLAY ARRAY sub-dialog on page 1597
The DISPLAY ARRAY sub-dialog is the controller to implement the navigation in a list of records, with option data modification actions.

ui.Dialog.getCurrentItem on page 2531
Returns the current item having focus.

Which form item has the focus? on page 1730
Identify what element of the current form has the focus.

Implementing dialog action handlers on page 1761
How to execute user code in ON ACTION blocks when an action is fired.

Detection of focus changes
Describes how to detect when the focus goes from field to field or to a read-only list.

Detecting focus changes in a singular INPUT or CONSTRUCT
An singular INPUT or CONSTRUCT controls several fields that can get the focus and become current. In order to execute some code when a field gets (or loses) the focus, use the following control blocks:

- **BEFORE FIELD** (a specific field (or group of fields) gets the focus)
- **AFTER FIELD** (the field (or group of fields) loses focus)

Detecting focus changes in a singular DISPLAY ARRAY
A singular DISPLAY ARRAY controls rows of a list, that can get the focus and become current. In order to execute some code when a row gets (or loses) the focus, use the following control blocks:

- **BEFORE ROW** (a new row gets the focus inside a DISPLAY ARRAY or INPUT ARRAY list)
- **AFTER ROW** (a row inside a DISPLAY ARRAY or INPUT ARRAY list loses focus)

Detecting focus changes in a singular INPUT ARRAY
An singular INPUT ARRAY controls several fields and rows of a list, that can get the focus and become current. In order to execute some code when a field or a row gets (or loses) the focus, use the following control blocks:

- **BEFORE ROW** (a new row gets the focus inside a DISPLAY ARRAY or INPUT ARRAY list)
- **BEFORE FIELD** (a specific field (or group of fields) gets the focus)
- **AFTER FIELD** (the field (or group of fields) loses focus)
- **AFTER ROW** (a row inside a DISPLAY ARRAY or INPUT ARRAY list loses focus)

Detecting focus changes in a DIALOG
A DIALOG interaction block can handle different parts of a form simultaneously. In order to execute some code when a part of the form gets (or loses) the focus, use the following control blocks:

- **BEFORE INPUT** (a field of this INPUT or INPUT ARRAY sub-dialog gets the focus and none of its fields had focus before)
- **BEFORE CONSTRUCT** (a field of this CONSTRUCT sub-dialog gets the focus and none of its fields had focus before)
- **BEFORE DISPLAY** (this DISPLAY ARRAY sub-dialog gets the focus and none of its fields had focus before)
- **BEFORE ROW** (a new row gets the focus inside a DISPLAY ARRAY or INPUT ARRAY list)
- **BEFORE FIELD** (a specific field (or group of fields) gets the focus)
- **AFTER FIELD** (the field (or group of fields) loses focus)
- **AFTER ROW** (a row inside a DISPLAY ARRAY or INPUT ARRAY list loses focus)
- **AFTER DISPLAY** (this DISPLAY ARRAY sub-dialog loses the focus = focus goes to another sub-dialog)
- **AFTER CONSTRUCT** (this CONSTRUCT sub-dialog loses the focus = focus goes to another sub-dialog)
• **AFTER INPUT** (this INPUT or INPUT ARRAY sub-dialog loses focus = focus goes to another sub-dialog)

These triggers are also executed by **NEXT FIELD**.

**Related concepts**

**DIALOG control blocks** on page 1607

*Dialog control blocks* are predefined dialog triggers where you can implement specific code to control the interactive instruction.

**Enabling autocompletion**

Autocompletion allows a list of completion proposals to be displayed while the user is typing text into a field.

**Introduction to autocompletion**

Text input fields (like **EDIT** and **BUTTONEDIT**) can be defined with an autocompletion feature, by combining the **COMPLETER** form field attribute with program code providing the list of completion proposals in a dynamic array of strings. Autocompletion is activated with the **DIALOG.setCompleterItems()** method, when the **ON CHANGE** trigger is fired for the field.

**Defining a form field for autocompletion**

In order to enable autocompletion in a text form field, you must define the **COMPLETER** attribute:

```plaintext
EDIT f1 = FORMONLY.firstname, COMPLETER;
```

The **COMPLETER** attribute can be used for **EDIT** and **BUTTONEDIT** fields.

**Providing the front-end with a list of completion proposals**

The **DIALOG.setCompleterItems()** method must be used to provide the list of completion proposals during dialog execution:

```plaintext
DEFINE items DYNAMIC ARRAY OF STRING
-- fill the array with items
LET items[1] = "Ann"
LET items[2] = "Anna"
LET items[3] = "Annabel"
CALL DIALOG.setCompleterItems(items)
```

**Important:** Consider the execution time of the code creating the completion proposal list. For example, avoid long complex SQL queries that can take more than a few milliseconds to complete.

The **setCompleterItems()** method will raise error `-8114` if the list of items contains more than 50 elements. The purpose of autocompletion is to provide a short list of completion proposals to the user. Note that this error is not trappable with exception handlers like **TRY/CATCH**, the code must avoid exceeding the limit.

**Detecting user input**

When implementing autocompletion, you must detect when the user modifies the field value, to adapt the list of items with the **setCompleterItems()** method.

In order to detect user input, define the **ON CHANGE** dialog control block, and call a custom function by passing the **DIALOG** object, and the value of the current field as parameter, to filter the completion proposal list accordingly:

```plaintext
INPUT BY NAME rec.firstname
... 
ON CHANGE firstname
    CALL fill_proposals_firstname(DIALOG, rec.firstname)
```
For text fields defined with the COMPLETER attribute, the ON CHANGE trigger will be fired without leaving the field, each time the user types characters in. The event is fired after a short delay, so as not to overload the UI exchanges between the front-end and the runtime system.

**Note:** The item list for a field implementing autocompletion is not permanent: The program must redefine the autocompletion item list with `setCompleterItems()`, on every ON CHANGE event.

**Example**

The example below implements form field with autocompletion: Each time the ON CHANGE trigger is fired, the set of completion proposals is adapted to the current field value, to match names that start with the same characters typed by the user.

**Form file (compl.per):**

```
LAYOUT GRID
{[f1 ] }
{[f2 ] }
END
END
ATTRIBUTES
EDIT f1 = FORMONLY.field1, COMPLETER;
EDIT f2 = FORMONLY.field2;
END
```

**Program file (compl.4gl):**

```
DEFINE all_names DYNAMIC ARRAY OF STRING

MAIN
  DEFINE rec RECORD
    field1 STRING,
    field2 STRING
  END RECORD
  CALL fill_names()
  OPEN FORM f FROM "compl"
  DISPLAY FORM f
  OPTIONS INPUT WRAP
  INPUT BY NAME rec.* ATTRIBUTES(UNBUFFERED)
    ON CHANGE field1
        CALL fill_proposals(DIALOG, rec.field1)
  END INPUT
END MAIN

FUNCTION fill_names()
  DEFINE i INTEGER
  LET i=0
  LET all_names[i:=i+1] = "Amanda"
  LET all_names[i:=i+1] = "Ann"
  LET all_names[i:=i+1] = "Anna"
  LET all_names[i:=i+1] = "Annabelle"
  LET all_names[i:=i+1] = "Barbara"
  LET all_names[i:=i+1] = "Barry"
  LET all_names[i:=i+1] = "Brice"
END FUNCTION

FUNCTION fill_proposals(dig, curr_val)
  DEFINE dig ui.Dialog, curr_val STRING
  DEFINE curr_set DYNAMIC ARRAY OF STRING,
    i, x INTEGER
```
LET x=0
FOR i=1 TO all_names.getLength()
    IF upshift(all_names[i]) MATCHES upshift(curr_val)||"*" THEN
        LET curr_set[x:=x+1] = all_names[i]
    END IF
END FOR
CALL dlg.setCompleterItems(curr_set)
END FUNCTION

Related concepts

Reacting to field value changes on page 1724
This section describes the purpose of the ON CHANGE interaction block.

The buffered and unbuffered modes on page 1720
The buffered and unbuffered mode control the synchronization of program variables and form fields.

Filling a COMBOBOX item list
The item list of COMBOBOX fields can be initialized at runtime.

Introduction to COMBOBOX fields

COMBOBOX fields are typically used when a field can hold a short predefined list of values. COMBOBOX fields are usually rendered with a drop-down list, from where the end user can choose a value.

Note: All items of a COMBOBOX list will be transmitted to the front-end. Therefore, the number of items that can be selected in a COMBOBOX fields should be limited to 20 to 50 items. If the selection list holds more items, consider using a BUTTONEDIT field, which opens a new window with a TABLE container.

COMBOBOX item lists can be defined in three different ways:

1. In the form definition file, as a static list of items with single values.
2. In the form definition file, as a static list of items with value/label pairs.
3. At runtime when the form is loaded, as single values or value/label pairs.

In this topic we will learn how to implement a COMBOBOX field that is filled dynamically.

For static item list definitions, see COMBOBOX item type on page 1270.

Defining the COMBOBOX initialization function

In order to fill a COMBOBOX field when the form file is loaded, use the INITIALIZER attribute to define the name of the function that will be called to fill the item list:

```
COMBOBOX f1 = FORMONLY.city, INITIALIZER = init.fill_city;
```

The INITIALIZER attribute accepts a module prefix for the initializer function. When a module is specified, it will be loaded on demand to resolve the function symbol at runtime when the form is displayed. When specifying only a function name, make sure that the initialization function is available when the form is displayed: Use for example IMPORT FGL of the module, and call one of the functions of that module before displaying the form.

Important: The initialization function name is case insensitive.

Important: The module prefix of the initialization function name is case sensitive (unlike the function name, which is case insensitive).

Defining a global COMBOBOX initialization function

If needed, it is possible to define a common function that implements the item list initialization for all comboboxes of the forms loaded by a program, by using the ui.Combobox.setDefaultInitializer() method.

Tip: Use the TAG attribute to distinguish COMBOBOX fields in all your forms.
Implementing the item list initialization function

The function defined with the INITIALIZER attribute takes a `ui.ComboBox` object as parameter.

To add items to the selection list of the COMBOBOX field, use the `addItem()` method of `ui.ComboBox`:

```plaintext
FUNCTION fill_city( cmd ui.ComboBox ) RETURNS ()
    CALL cmb.addItem(101,"Berlin")
    CALL cmb.addItem(102,"Madrid")
    CALL cmb.addItem(103,"London")
    CALL cmb.addItem(104,"Paris")
    CALL cmb.addItem(105,"Rome")
END FUNCTION
```

**Note:** If you want to define a list of items with single values, specify only the first parameter of `addItem()`.

Detecting COMBOBOX value change

In order to detect a value change in a COMBOBOX, define the `ON CHANGE` dialog control block. The `ON CHANGE` block will be immediately executed when the user selects a new item in the list. One can typically clear other fields related to the COMBOBOX field:

```plaintext
ON CHANGE city
    LET rec.address = NULL
```

NULL values in COMBOBOX fields

Pay attention to NULL value handling with COMBOBOX fields:

- By default, if the field allows nulls, the item list automatically gets a NULL item.
- It is recommended to disallow nulls with the `NOT NULL` attribute, and add a special item such as `(0,"<Undefined>")` to identify a non-specified-value.

Example

The next example shows how to implement the function to fill the item list of a COMBOBOX field with a list of cities. When the COMBOBOX field is changed, the `ON CHANGE` block is fired and the address field is cleared:

Form file (combobox.per):

```plaintext
LAYOUT
GRID
  {City : [f1]
   Address: [f2]
  }
END
END
ATTRIBUTES
COMBOBOX f1 = FORMONLY.city, INITIALIZER=combobox.fill_city;
EDIT f2 = FORMONLY.address;
END
```

Program file (combobox.4gl):

```plaintext
MAIN
    DEFINE rec RECORD
        city INTEGER,
        address VARCHAR(100)
    END RECORD

    OPEN FORM f1 FROM "combobox"
```
DISPLAY FORM f1

INPUT BY NAME rec.* ATTRIBUTES(UNBUFFERED)
  ON CHANGE city
    LET rec.address = NULL
END INPUT

END MAIN

FUNCTION fill_city(cmb ui.ComboBox) RETURNS ()
  CALL cmb.addItem(101,"Berlin")
  CALL cmb.addItem(102,"Madrid")
  CALL cmb.addItem(103,"London")
  CALL cmb.addItem(104,"Paris")
  CALL cmb.addItem(105,"Rome")
END FUNCTION

Related concepts
COMBOBOX item type on page 1270
Defines a line-edit with a drop-down list of values.

Field-anchored windows
The Window style attribute "position" can be set to "field" in order to display the window under the current field.

The Window style attribute "position"
The "position" style attribute for Window elements can take several values, to control where the window will appear on the screen. It can for example be anchored to an existing window, or centered. For more details, see the position style attribute reference topic.

In this topic, we will describe how to use the "field" value of the Window position style attribute, to display the window under the current field having the focus. This will result in a "drop-down" window, typically used to select a record in a list:

![Figure 85: Typical Window.position="field" usage](image)

Defining your custom "dropdown" Window style
The "position" style attribute for Window elements must be defined under a Style element.

We use the name "dropdown" to reference the style element in program source files.
Add the following XML elements to the default.4st file:

```xml
<Style name="Window.dropdown">
  <StyleAttribute name="position" value="field"/>
  <StyleAttribute name="windowType" value="modal"/>
  <StyleAttribute name="border" value="none"/>
  <StyleAttribute name="windowSystemMenu" value="no"/>
  <StyleAttribute name="actionPanelPosition" value="none"/>
  <StyleAttribute name="ringMenuPosition" value="none"/>
  <StyleAttribute name="statusBarType" value="none"/>
</Style>
```

**Note:** Additional Window style attributes are required to get the expected rendering. For example, we remove the border, system menu, statusbar, action panel, etc, to get a minimal window frame. Note also that the `windowType` attribute must be "modal".

The **zoom.per** form

The **zoom.per** form defines a TABLE container, with the **STYLE** attribute set to "dropdown":

```plaintext
LAYOUT
TABLE t1 (STYLE="dropdown")
{
  [c1 | c2 ]
  [c1 | c2 ]
  [c1 | c2 ]
  [c1 | c2 ]
  [c1 | c2 ]
}
END
END
ATTRIBUTES
c1 = FORMONLY.col1;
c2 = FORMONLY.col2;
END
INSTRUCTIONS
SCREEN RECORD sr(FORMONLY.*);
END
```

The **main.per** form

The **main.per** form defines a GRID container with two BUTTONEDIT fields, with an **ACTION**, that will fire the **ON ACTION** program code to open the zoom window:

```plaintext
LAYOUT
GRID
{
  [f1 ] [f2]
}
END
END
ATTRIBUTES
BUTTONEDIT f1 = FORMONLY.field1, ACTION=zoom1, IMAGE="zoom";
BUTTONEDIT f2 = FORMONLY.field2, ACTION=zoom2, IMAGE="zoom";
END
```

The **main.4gl** program

The **main.4gl** module implements a simple **INPUT** with **ON ACTION** handlers calling a function that opens the zoom window.
Note that the OPEN WINDOW instruction uses the STYLE="dropdown" attribute:

```
MAIN
  DEFINE rec RECORD
    field1 STRING,
    field2 STRING
  END RECORD
  OPEN FORM f1 FROM "main"
  DISPLAY FORM f1
  INPUT BY NAME rec.* ATTRIBUTES(UNBUFFERED)
    ON ACTION zoom1
      LET rec.field1 = open_zoom(rec.field1)
    ON ACTION zoom2
      LET rec.field2 = open_zoom(rec.field2)
  END INPUT
END MAIN

FUNCTION open_zoom(cv)
  DEFINE cv STRING
  DEFINE arr DYNAMIC ARRAY OF RECORD
    col1 INT,
    col2 STRING
  END RECORD
  DEFINE i INT
  FOR i=1 TO 100
    LET arr[i].col1 = i
    LET arr[i].col2 = SFMT("Item %1",i)
  END FOR
  OPEN WINDOW wz WITH FORM "zoom" ATTRIBUTES(STYLE="dropdown")
  LET int_flag = FALSE
  DISPLAY ARRAY arr TO sr.*
  AFTER DISPLAY
    IF NOT int_flag THEN
      LET cv = arr[arr.curr()].col2
    END IF
  END DISPLAY
END FUNCTION
```

Related concepts

Windows and forms on page 1131
The section describes the concept of windows and forms in the language.

Presentation styles on page 1165
Use presentation styles to specify decoration attributes for window and form elements.

Dialog actions

Describes how to program action handling when the end user triggers an action on the front-end.

**Action handling basics**

This topic describes the basics of action views, action events, and action handlers.

In the user interface of the application, action views can produce action events, that will execute user code in the corresponding action handler defined in the current interactive instruction of the program.

Actions views are for example BUTTON form items.

Action handlers are **ON ACTION** or **COMMAND** dialog blocks that execute user code, in the current interactive dialog.
Action views are bound to action handlers by name.

If no action view is explicitly defined in the current form, the front-end will create a "default action view" for the action. This is typically a button that appears in a specific area, located and decorated following the front-end platform standards.

Actions can be configured with action attributes. These can be defined explicitly at the action view level (button in form), as dialog-specific action configuration (ON ACTION name ATTRIBUTES(...)), or with action defaults.

Special actions are supported, such as the interrupt action if the user cancels a running application procedure.

Related concepts
Defining action views in forms on page 1759
How to define action views that will fire action events.
Implementing dialog action handlers on page 1761
How to execute user code in ON ACTION blocks when an action is fired.
Binding action views to action handlers on page 1762
How are action views of the forms bound to action handlers in the program code?
Default action views on page 1743
A default action view is created to render an action handler when no explicit action view exists for it.
ACTION DEFAULTS section on page 1296
The ACTION DEFAULTS section defines local action view default attributes for the form elements.

Predefined actions
Genero predefines some action names for common operations of interactive instructions.

Predefined actions are different from user-defined action, in the sense that the name of a predefined action is reserved, and the action may have an ON ACTION handler, while user-defined actions have a specific name, and must be implemented with an ON ACTION handler.

There are two types of predefined actions:

- Automatic actions: actions that are automatically created and handled by the program dialog, like accept, cancel, insert.

- Special actions: actions with a special usage, that can be invoked asynchronously or automatically by the front-end, like interrupt, dialogtouched.

Default decoration attributes and keyboard shortcuts are defined with action defaults, like for user-defined actions.

Overwriting predefined actions with ON ACTION

If you define your own ON ACTION handler with the name of a predefined action, the default action processing is bypassed and the program code is executed instead.

This code example defines an ON ACTION clause with the accept predefined action name:

```plaintext
INPUT BY NAME customer.*
    ON ACTION accept
     ...
END INPUT
```

In this case, the default behavior of the automatic accept action is not performed; the user code is executed instead.

Predefined actions enabled depending on the context

Some predefined actions (such as insert, append and delete in INPUT ARRAY) are enabled and disabled automatically by the dialog depending on the context (for example, when a static array used by the INPUT ARRAY is full, the insert and append actions get disabled).

Even when overwriting such actions with your own action handler, the runtime system will continue to enable and disable the actions automatically.
Overwriting predefined actions is not recommended.

**Related concepts**

Syntax of INPUT ARRAY instruction on page 1532

The INPUT ARRAY supports data entry by users into a screen array and stores the entered data in an array of records.

**Binding action views to predefined actions**

As for user-defined actions, if you design forms with action views using predefined action names, they will automatically attach themselves to the actions of the interactive instructions.

It is also possible to define default images, texts, comments and accelerator keys in the action defaults resource file for the predefined actions.

**Related concepts**

Configuring actions on page 1744

Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with action attributes.

**List of predefined actions**

**Important:** The predefined actions described in these tables are created depending on the usage context described in the Context descriptions on page 1742 notes.

**Table 381: Automatic actions (automatically created by dialogs, can be overwritten by ON ACTION)**

<table>
<thead>
<tr>
<th>Action Name</th>
<th>Description</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>accept</td>
<td>Validates the current interactive instruction</td>
<td>(1)</td>
</tr>
<tr>
<td>cancel</td>
<td>Cancels the current interactive instruction</td>
<td>(1)</td>
</tr>
<tr>
<td>close</td>
<td>Triggers a cancel key in the current interactive instruction (by default)</td>
<td>(7)</td>
</tr>
<tr>
<td>insert</td>
<td>Inserts a new row before current row</td>
<td>(2)</td>
</tr>
<tr>
<td>append</td>
<td>Appends a new row at the end of the list</td>
<td>(2)</td>
</tr>
<tr>
<td>delete</td>
<td>Deletes the current row</td>
<td>(2)</td>
</tr>
<tr>
<td>find</td>
<td>Opens the fglfind dialog window to let the user enter a search value, and seeks to the row matching the value</td>
<td>(4)</td>
</tr>
<tr>
<td>findnext</td>
<td>Seeks the next row matching the value entered during the fglfind dialog</td>
<td>(4)</td>
</tr>
<tr>
<td>nextrow</td>
<td>Moves to the next row</td>
<td>(4)</td>
</tr>
<tr>
<td>prevrow</td>
<td>Moves to the previous row</td>
<td>(4)</td>
</tr>
<tr>
<td>firstrow</td>
<td>Moves to the first row</td>
<td>(4)</td>
</tr>
<tr>
<td>lastrow</td>
<td>Moves to the last row</td>
<td>(4)</td>
</tr>
<tr>
<td>help</td>
<td>Shows the help topic defined by the HELP clause</td>
<td>(1)</td>
</tr>
<tr>
<td>editcopy</td>
<td>Copy selected rows (or current row if MRS is off) to the clipboard</td>
<td>(8)</td>
</tr>
<tr>
<td>expandall</td>
<td>A tree node is completely expanded</td>
<td>(8)</td>
</tr>
<tr>
<td>collapseall</td>
<td>A tree node is completely collapsed</td>
<td>(8)</td>
</tr>
</tbody>
</table>
## Table 382: Special actions (specific behavior, requires ON ACTION)

<table>
<thead>
<tr>
<th>Special Action Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>interrupt</td>
<td>Sends an interruption request to the program when processing. For more details, see User interruption handling on page 1711. <strong>Note:</strong> This special action does not require an ON ACTION block.</td>
</tr>
<tr>
<td>browser_back</td>
<td>Sent when the user hits the back button in a web browser (web front-end only).</td>
</tr>
<tr>
<td>browser_forward</td>
<td>Sent when the user hits the forward button in a web browser (web front-end only).</td>
</tr>
<tr>
<td>dialogtouched</td>
<td>Sent by the front-end each time the user modifies the value of a field. For more details, see Immediate detection of user changes on page 1725.</td>
</tr>
<tr>
<td>enterbackground</td>
<td>On Mobile devices, this action is fired when the app goes to background mode.</td>
</tr>
<tr>
<td>enterforeground</td>
<td>On Mobile devices, this action is fired when the app goes to foreground mode. For background and foreground action see Background/foreground modes on page 3573.</td>
</tr>
<tr>
<td>windowresized</td>
<td>On Mobile devices, this action is sent when changing the orientation of the device. On other front-ends, it is sent when the current active window is resized. For more details, see Adapting to viewport changes on page 1417.</td>
</tr>
<tr>
<td>notificationpushed</td>
<td>On Mobile devices, this action is fired when receiving a push notification message. See getRemoteNotifications on page 2703.</td>
</tr>
<tr>
<td>cordovacallback</td>
<td>On Mobile devices, this action is fired when receiving Cordova plugin results. See callWithoutWaiting on page 2722.</td>
</tr>
</tbody>
</table>

### Context descriptions

1. CONSTRUCT, INPUT, PROMPT, INPUT ARRAY and DISPLAY ARRAY.
2. INPUT ARRAY only.
3. CONSTRUCT, INPUT and INPUT ARRAY.
4. INPUT ARRAY and DISPLAY ARRAY.
5. Only possible when no interactive instruction is active.
6. Possible in any kind of interactive instruction (MENU included).
7. DIALOG, CONSTRUCT, INPUT, PROMPT, INPUT ARRAY and DISPLAY ARRAY.
8. DISPLAY ARRAY only.

### Related concepts

**Dialog actions** on page 1739

Describes how to program action handling when the end user triggers an action on the front-end.

### List of local actions (GDC only)

**Important:** This feature is deprecated, its use is discouraged although not prohibited.

### List of local actions

**Important:** Local actions are a deprecated feature. Local actions are only supported by the GDC front-end for backward compatibility. Avoid binding action views with the action names listed in the following table, and avoid changing the action defaults attributes (such as accelerators) for these actions.
### Table 383: Local actions (handled by the front-end)

<table>
<thead>
<tr>
<th>Local Action Name</th>
<th>Description</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>editcopy</td>
<td>Copies the current selected text to the clipboard</td>
<td>(4)</td>
</tr>
<tr>
<td>editcut</td>
<td>Copies the current selected text to the clipboard and removes the text from the current input widget</td>
<td>(4)</td>
</tr>
<tr>
<td>editpaste</td>
<td>Pastes the clipboard content to the current input widget</td>
<td>(4)</td>
</tr>
<tr>
<td>nextpage</td>
<td>Moves to the next page in the list</td>
<td>(2)</td>
</tr>
<tr>
<td>prevpage</td>
<td>Moves to the previous page in the list</td>
<td>(2)</td>
</tr>
<tr>
<td>nextfield</td>
<td>Moves to the next field in the form</td>
<td>(1)</td>
</tr>
<tr>
<td>prevfield</td>
<td>Moves to the previous field in the form</td>
<td>(1)</td>
</tr>
<tr>
<td>nexttab</td>
<td>Moves to the next page in the folder</td>
<td>(3)</td>
</tr>
<tr>
<td>prevtab</td>
<td>Moves to the previous page in the folder</td>
<td>(3)</td>
</tr>
</tbody>
</table>

**Context descriptions**

1. CONSTRUCT, INPUT and INPUT ARRAY.
2. INPUT ARRAY and DISPLAY ARRAY.
3. Possible in any kind of interactive instruction (MENU included).
4. DIALOG, CONSTRUCT, INPUT, PROMPT, INPUT ARRAY and DISPLAY ARRAY.

**Automatic and local actions using the same name**

Some predefined actions exist as both automatic actions and as local actions (like editcopy). The automatic actions are created from the dialog context. If an automatic action has to be defined and if a local action exists with the same name, the automatic action takes precedence over the local action.

For example, if the dialog context requires an editcopy runtime action, the local editcopy action will not be handled by the front-end. Identical action names are used for automatic and local action to bind with the same action view. For example, the same toolbar button created with the editcopy name will trigger the automatic action or the local action, depending on the context.

**Related concepts**

Dialog actions on page 1739

Describes how to program action handling when the end user triggers an action on the front-end.

**Default action views**

A default action view is created to render an action handler when no explicit action view exists for it.

If no explicit action view is defined, such as a toolbar button, a topmenu item or a simple button in the form layout, the front-end creates a default action view for each COMMAND or ON ACTION action handler, or implicit action such as insert/delete in INPUT ARRAY, in the current interactive instruction.

The rendering of default action views depends on the platform. On a desktop front-end, the default action views appear as buttons in the action frame in the right-hand side of the current window. On a mobile device, the default action views will follow the mobile user interface standards, which can be vendor specific. For more details about default action views on mobile, see Action views on mobile devices on page 1777.
When creating action handlers with \texttt{ON KEY} (or \texttt{COMMAND KEY} without a command name in a \texttt{MENU}), the default action view is invisible. If you define a \texttt{text} attribute in the action defaults, the default action view is made visible.

Control the default action view visibility by using the \texttt{DEFAULTVIEW} action attribute.

If one or more action views are defined explicitly for a given action, the front-end considers that the default view is not needed. Typically, if you define in the form a \texttt{BUTTONEDIT} field, a \texttt{BUTTON}, or a \texttt{TOOLBAR} item that triggers the action, you do not need an additional button in the action frame.

The presentation of default action views can be controlled with presentations style attributes for the \texttt{Window AUI} tree nodes.

\textbf{Related concepts}

\texttt{Action handling basics} on page 1739

This topic describes the basics of action views, action events, and action handlers.

\texttt{Configuring actions} on page 1744

Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with \textit{action attributes}.

\textbf{Related reference}

\texttt{Window style attributes} on page 1220

Window presentation style attributes apply to a window element.

\textbf{Configuring actions}

Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with \textit{action attributes}.

\textit{Action attributes} define attributes for actions, including decoration such as text, icon, comment, as well as keyboard accelerator (Ctrl-?, function keys), and also semantics such as current field validation control when an action is fired.

The action attributes can be defined in different ways:

1. Common action attributes can be centralized in a global action defaults file with the .4ad extension.
2. Form-specific action attributes can be defined in the \texttt{ACTION DEFAULTS} section of a form definition file.
3. Dialog-specific action attributes can be defined in programs with the \texttt{ATTRIBUTES()} clause of \texttt{ON ACTION} handlers.
4. Form-item specific action view attributes (decoration only) can be defined directly at the item level (labels, icons, comments).
5. Default action views can be configured dynamically with \texttt{ui.Dialog.setAction*()} methods.

Action attributes do not only define action view decoration; it is possible to define the semantics of an action, for example by using the VALIDATE action default attribute. Functional attributes take effect for a given action when the dialog implementing the action handler becomes active.

Action attributes are particularly important for rendering the default action view (when there is no explicit action view defined in the form). This is typically the case when no form is associated with the dialog.

Action attributes can be defined with action defaults. Common action defaults are defined in a global action defaults (.4ad) file, while form specific actions are defined within the \texttt{ACTION DEFAULTS} section of form files.

If a dialog is not attached to a specific form such as an independent \texttt{MENU}, define the action attributes with the \texttt{ATTRIBUTES} clause of \texttt{ON ACTION} handlers, to render the default view and configure the action semantics.

Attributes defined by \texttt{ON ACTION action-name ATTRIBUTES()} will only be applied to the default action view: The elements in the forms do not get decoration attributes defined by dialog action handlers.

The final decoration and functional attribute values are set in this order of precedence:

1. The attribute defined in the action view element definition itself (local form element decoration) - For default action views, the attributes are defined with \texttt{ui.Dialog.setAction*()} methods.
2. The attribute defined in the \texttt{ATTRIBUTES} clause of an \texttt{ON ACTION} handler.
3. The attribute defined for the action in the \texttt{ACTION DEFAULTS} section of the current form.
4. The attribute defined for the action in the global action defaults file (.4ad).
Note: The global action defaults can be loaded at runtime with `ui.Interface.loadActionDefaults` on page 2481, and the form-specific `ACTION DEFAULTS` section can be loaded with `ui.Form.loadActionDefaults` on page 2504. These solutions are typically used in a migration process, to get action views decoration without modifying existing `.per` forms.

The syntax for defining action attributes depends on the context where the action attributes are defined:

- In the `.4ad` file, the syntax follows XML standards, as defined in Action default attributes reference (.4ad) on page 1163.
- In the `.per` files, the syntax follows the form specification file attributes, as defined in `ACTION DEFAULTS` section.
- In the `.4gl` files (in dialog action handlers), the syntax follows the language syntax, as defined in ON ACTION block on page 1478.

Example

Consider the following parts of code related to the same action definition, namely "print":

1. A BUTTON item defined in the form specification file:

   ```
   ATTRIBUTES
   BUTTON b1: print, TEXT="Print item";
   END
   ```

2. A dialog instruction with code defining the ON ACTION handler with an ATTRIBUTES clause:

   ```
   DIALOG ...
   ...
   ON ACTION print
   ATTRIBUTES( ROWBOUND, IMAGE = "printer_2" )
   ...
   ```

3. The form `ACTION DEFAULTS` section defining:

   ```
   form.per:
   ACTION DEFAULTS
   ACTION print (IMAGE="printer_1",
   COMMENT="Print the order",
   ACCELERATOR=Control-P,
   CONTEXTMENU=NO)
   END
   ```

4. A global `.4ad` action defaults file defining:

   ```
   <ActionDefaultList>
   <ActionDefault name="print" text="Print" image="smiley" />
   </ActionDefaultList>
   ```

When the dialog executes, the "print" action will get the following functional attributes:

- `acceleratorName = "control-p"` - from the form `ACTION DEFAULTS` section
- `rowBound = "yes"` - from the dialog ON ACTION handler
- `contextMenu = "no"` - from the form `ACTION DEFAULTS` section

The form button (the action view) will get the following decoration attribute values:

- `text = "Print item"` - from the BUTTON form item
- `image = "printer_2"` - from the dialog ON ACTION handler
- `comment = "Print the order"` - from the form `ACTION DEFAULTS` section

Related concepts

Action defaults files on page 1162
Action defaults files allow to centralize action configuration parameters such as text, icon, accelerators and behavior options in XML format.

**Action attributes context usage**

Action attributes are used to configure functional and decoration properties of actions. The table below lists the possible action attributes and indicates in what context they can be defined.

**Table 385: Action attributes definitions**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Form action view</th>
<th>Dialog action handler</th>
<th>Form action defaults section</th>
<th>Global action defaults file (.4ad)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCELERATOR</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>See ACCELERATOR action attribute on page 1750.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACCELERATOR2</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>See ACCELERATOR2 action attribute on page 1751.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACCELERATOR3</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>See ACCELERATOR3 action attribute on page 1752.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACCELERATOR4</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>See ACCELERATOR4 action attribute on page 1752.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMMENT</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>See COMMENT action attribute on page 1753.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONTEXTMENU</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>See CONTEXTMENU action attribute on page 1754.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEFAULTVIEW</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>See DEFAULTVIEW action attribute on page 1755.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DISCLOSUREINDICATOR</td>
<td>No</td>
<td>Yes (only for MENU)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>See DISCLOSUREINDICATOR action attribute on page 1756.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMAGE</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>See IMAGE action attribute on page 1756.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Using attributes of action defaults

Purpose of action defaults

*Action defaults* allow you to define default attributes for common action. These defaults can be overwritten with form item attributes, or with dialog action handler attributes (only for default action views).

Centralize action attributes with action defaults, to avoid specifying them in all the source files that define the same action view and action handler. For example, you can specify the default text, image and keyboard accelerator for elements like push **BUTTON**, **TOOLBAR** items, **TOPMENU** options.

Common action defaults are typically defined in a global action defaults (.4ad) file, while form specific actions are configured with form action defaults in the **ACTION_DEFAULTS** section of the .per form specification file.

Global action defaults file

Global action defaults are defined in an XML file with the .4ad extension. By default, the runtime system searches for a file named default.4ad in several directories as described in the FGLRESOURCEPATH reference topic. If no file was found, standard action default settings are loaded from the $FGLDIR/lib/default.4ad file.

Important: Global action defaults must be defined in a unique file; you cannot combine several 4ad files.

It is possible to use localized strings in action default attributes such as **TEXT** and **COMMENT**, by using **LStr** XML elements:

```xml
<ActionDefaultList>
  <ActionDefault name="yes" text="Yes">
    <LStr text="common.yes"/>
  </ActionDefault>
  ...
</ActionDefaultList>
```

Note: A global action defaults file can be loaded dynamically by program with the `ui.Interface.loadActionDefaults()` method. Use this solution if you need different global action defaults depending on the program. However, it is recommended that you consider using a single global default actions file, to get the same decoration and keyboard accelerators in all your programs.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Form action view</th>
<th>Dialog action handler</th>
<th>Form action defaults section</th>
<th>Global action defaults file (.4ad)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROWBOUND</td>
<td>No</td>
<td>Yes (only for list dialogs)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>See <strong>ROWBOUND</strong> action attribute on page 1757.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEXT</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>See <strong>TEXT</strong> action attribute on page 1758.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VALIDATE</td>
<td>No</td>
<td>Yes (only for input dialogs)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>See <strong>VALIDATE</strong> action attribute on page 1759.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Form specific action defaults

Action defaults can be defined at the form level in the ACTION DEFAULTS section. When action defaults are defined in the form file, action views get the attributes defined locally for this form:

```
ACTION DEFAULTS
    ACTION print (TEXT="Print",
                  IMAGE="printer",
                  COMMENT="Print the current record",
                  ACCELERATOR=CONTROL-P)
END
```

It is possible to use localized strings in action default attributes such as TEXT and COMMENT:

```
ACTION print (TEXT=%"common.print")
```

Note: Form specific action defaults files can be loaded dynamically by program with the `ui.Form.loadActionDefaults()` method. Use this solution if you cannot change your .per form definition files to define ACTION DEFAULTS section. The loadActionDefaults() method of a form object is typically used in a generic form initializer function.

Action defaults are applied only once

Decoration attributes (like TEXT, IMAGE) of an action view will automatically be set with the value defined in the action defaults to all new action views of a newly-created form, if there is no explicit value specified in the element definition for that attribute. Decoration action default attributes are applied only once, to newly-created form elements. Dynamic changes are not reapplied to action views. For example, if you first load a toolbar, then you load a global action defaults file, the attributes of the toolbar items will not be updated with the last loaded action defaults. If you dynamically create action views (like TopMenu or ToolBar), the action defaults are not applied, so you must set all decoration attributes by hand.

Action defaults and sub-dialog actions

The action default attributes to be applied are selected following the name of the action. In some situations, the action view can be bound to an action handler by specifying a sub-dialog and/or field name prefix. For those views, the action defaults defined with the corresponding action name will be used to set the attributes with the default values. In other words, the prefix will be ignored. For example, if an action view is defined with the name `custlist.append`, it will get the action defaults defined for the `append` action.

Functional attributes

Functional attributes (like VALIDATE, ACCELERATOR) can only be defined in action defaults, or in ON ACTION dialog action handlers with the ATTRIBUTES clause. Functional attributes take effect for a given action when the dialog becomes active.

Dialog action handler attributes

Action attributes can be specified at the dialog instruction level for default action views. These action attributes will overwrite the attributes defined in action defaults.

To define dialog-level action attributes for an action, add the ATTRIBUTES() clause to ON ACTION, with a comma-separated list of action default attributes:

```
ON ACTION print
    ATTRIBUTES (TEXT = "Print",
                 COMMENT = "Print the current record",
                 IMAGE = "printer",
                 VALIDATE = NO)
```
It is possible to use localized strings in action attributes such as TEXT and COMMENT:

```plaintext
ON ACTION print
    ATTRIBUTES (TEXT = "%common.print.label",
                COMMENT = "%common.print.comment",
                ...
```

Dialog-level action attributes are typically used when the dialog is not related to a specific form, for example with independent MENU dialogs.

If the current form defines explicit action views (buttons in layout, toolbar buttons, topmenu items) with the same name as the ON ACTION handler defining action attributes with the ATTRIBUTES() clause, the explicit action views will not get the action attributes defined by the ON ACTION.

**Related concepts**

Localized strings on page 538

Localized strings provide a means of writing applications in which the text of strings can be customized on site.

### Configuring default action views dynamically

Attributes of default action views can be changed dynamically during a dialog execution with `ui.Dialog.setAction*()` methods.

For example, to change the text and icon of the default action view bound to the "print" action:

```plaintext
CALL DIALOG.setActionText("print", "Print order")
CALL DIALOG.setActionImage("print", "printer")
```

The corresponding default action view will be decorated with the new text and icon.

**Note:** The attributes set with `ui.DIALOG.setAction*()` methods are volatile and last only for the duration of the current dialog.

**Related concepts**

Localized strings on page 538

Localized strings provide a means of writing applications in which the text of strings can be customized on site.

### Text attribute shows default action view

When creating actions with ON KEY (or COMMAND KEY without a command name in a MENU), the default action view (i.e. button in action frame) is invisible. However, if you define a text action attribute for the corresponding key action, the default action view is made visible.

You can also control the visibility of the default action view with the DEFAULTVIEW action attribute.

Note that it is also possible to set key labels with form attributes (KEY) or with function calls (FGL_SETKEYLABEL()), this feature is supported for backward compatibility. Use action default text attributes in new developments.

**Related concepts**

Setting action key labels on page 1775

Labels can be defined to decorate buttons controlled by ON KEY / COMMAND KEY action handlers.

### Defining keyboard accelerators for actions

Keyboard accelerators keys are attributes defining the keyboard shortcuts for actions.

Keyboard accelerators can be defined at several levels (global action defaults, form file action defaults, dialog instructions).

For example, in a .per form file ACTION DEFAULTS section:

```plaintext
ACTION DEFAULTS
```
ACTION print (TEXT="Print", ACCELERATOR=Control-P)
END

Note: Up to four accelerator keys can be defined for the same action in action defaults. However, as a general pattern, define only one accelerator per action.

The runtime system sets default accelerators for predefined actions, when no accelerators are explicitly defined for the action. The default accelerator of predefined actions depends on the TUI/GUI user interface mode. For example, in TUI mode, the accept action gets the Escape key, but in GUI mode, it gets the Return and Enter keys (from FGLDIR/lib/default.4ad file). Similarly, in TUI mode, in INPUT ARRAY, the insert and delete predefined actions will respectively get the F1 and F2 accelerators (these can be changed with OPTIONS INSERT KEY / DELETE KEY); In GUI mode, INPUT ARRAY predefined actions get the default accelerators from the FGLDIR/lib/default.4ad file: insert gets F3, append gets Control-F3 and delete gets F4.

When a user-defined action is configured with an accelerator that would normally be used for a predefined action, the runtime system does not set that accelerator for the predefined action. For example (in GUI mode), if you define an ON ACTION quit with an action default using the accelerator "Escape", the "cancel" predefined action will not get the "Escape" default accelerator. In this case, user settings take precedence over defaults.

Note: Text edition and navigation keys such as [Home] and [End] are usually local to the widget. Depending on the context, such common keys might be eaten by the graphical widget and will not invoke the action configured with the corresponding accelerator. For example, even if the "firstrow" action defines the Home accelerator, when using an INPUT ARRAY, the [Home] key will jump to the beginning of the edit field, not the first row of the list.

When using the ON ACTION clause in a dialog instruction, action accelerators are used in both GUI and TUI mode. However, for backward compatibility, this is not done in TUI mode when using the ON KEY clause.

The traditional ON KEY clause in a dialog like INPUT implicitly defines the acceleratorName attribute for the action, and the corresponding action default accelerator will be ignored. For example, when you define an ON KEY (F10) block, the first accelerator will be "F10", even if an action default defines an accelerator "F5" for the action "F10". However, you can set other accelerators with the acceleratorName2, acceleratorName3 and acceleratorName4 attributes in action defaults.

Important: In TUI mode, actions created with ON KEY do not get accelerators of action defaults; only actions defined with ON ACTION will get accelerators of action defaults.

In a MENU dialog, the behavior is a bit different, see the COMMAND "option" and COMMAND KEY(keyname) clauses of MENU.

If you want to force an action to have no accelerator, you can use none as the accelerator name.

Related concepts
Predefined actions on page 1740
Genero predefines some action names for common operations of interactive instructions.

Keyboard accelerator names on page 1773
Reference for keyboard accelerator names to be used in ACCELERATOR* attributes, and in ON KEY / COMMAND KEY clauses in source dialog code.

Action attributes list
ACCELERATOR action attribute
The ACCELERATOR is an action attribute defining the primary accelerator key for an action.

Syntax 1 (Dialog action handlers):

```
ACCELERATOR = "key"
```

Syntax 2 (ACTION DEFAULTS section in form files):

```
ACCELERATOR = key
```
Syntax 3 (Global .4ad action defaults file)

```plaintext
acceleratorName2 = "key"
```

1. `key` defines the accelerator key.

Usage

The ACCELERATOR attribute defines the keyboard combination that can be pressed by the user to send an action to the program.

Note that in dialog-specific action attributes, the ACCELERATOR must be specified as a string expression.

This attribute applies to the actions defined by the current dialog in the current window. It can be specified as action default attribute in a global .4ad file, in the ACTION DEFAULTS section of form files, or as dialog action attribute.

Example

```plaintext
-- As action handler attribute
ON ACTION print ATTRIBUTES(ACCELERATOR="control-p")

-- As action default
ACTION DEFAULTS
  ACTION print (ACCELERATOR=control-p)
END

-- In a global action defaults file
<ActionDefault name="print" acceleratorName="control-p" ... />
```

Related concepts

**ACCELERATOR attribute** on page 1352
The ACCELERATOR is an action attribute defining the primary accelerator key for an action.

**Keyboard accelerator names** on page 1773
Reference for keyboard accelerator names to be used in ACCELERATOR* attributes, and in ON KEY / COMMAND KEY clauses in source dialog code.

**Defining keyboard accelerators for actions** on page 1749
**Configuring actions** on page 1744
Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with action attributes.

**ACCELERATOR2 action attribute**
The ACCELERATOR2 is an action attribute defining the secondary accelerator key for an action.

**Syntax 1 (Dialog action handlers):**

Note: There is no syntax to define accelerator #2 in dialog action handlers.

**Syntax 2 (ACTION DEFAULTS section in form files):**

```plaintext
ACCELERATOR2 = key
```

**Syntax 3 (Global .4ad action defaults file):**

```plaintext
acceleratorName2 = "key"
```

1. `key` defines the accelerator key.
Usage

The ACCELERATOR2 attribute defines the keyboard combination that can be pressed by the user to send an action to the program.

Important: This attribute is provided for specific cases, consider using only one accelerator per action.

Related concepts

ACCELERATOR2 attribute on page 1352
The ACCELERATOR2 is an action attribute defining the secondary accelerator key for an action.

Keyboard accelerator names on page 1773
Reference for keyboard accelerator names to be used in ACCELERATOR* attributes, and in ON KEY / COMMAND KEY clauses in source dialog code.

Defining keyboard accelerators for actions on page 1749
Configuring actions on page 1744
Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with action attributes.

ACCELERATOR3 action attribute
The ACCELERATOR3 is an action attribute defining the third accelerator key for an action.

Syntax 1 (Dialog action handlers):

Note: There is no syntax to defined accelerator #3 in dialog action handlers.

Syntax 2 (ACTION DEFAULTS section in form files):

ACCELERATOR3 = key

Syntax 3 (Global .4ad action defaults file):

acceleratorName3 = "key"

1. key defines the accelerator key.

Usage

The ACCELERATOR3 attribute defines the keyboard combination that can be pressed by the user to send an action to the program.

Important: This attribute is provided for specific cases, consider using only one accelerator per action.

Related concepts

ACCELERATOR3 attribute on page 1353
The ACCELERATOR3 is an action attribute defining the third accelerator key for an action.

Keyboard accelerator names on page 1773
Reference for keyboard accelerator names to be used in ACCELERATOR* attributes, and in ON KEY / COMMAND KEY clauses in source dialog code.

Defining keyboard accelerators for actions on page 1749
Configuring actions on page 1744
Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with action attributes.

ACCELERATOR4 action attribute
The ACCELERATOR4 is an action attribute defining the fourth accelerator key for an action.

Syntax 1 (Dialog action handlers):

Note: There is no syntax to defined accelerator #4 in dialog action handlers.
Syntax 2 (ACTION DEFAULTS section in form files):

ACCELERATOR4 = key

Syntax 3 (Global .4ad action defaults file):

acceleratorName4 = "key"

1. key defines the accelerator key.

Usage

The ACCELERATOR4 attribute defines the keyboard combination that can be pressed by the user to send an action to the program.

Important: This attribute is provided for specific cases, consider using only one accelerator per action.

Related concepts

ACCELERATOR4 attribute on page 1353

The ACCELERATOR4 is an action attribute defining the fourth accelerator key for an action.

Keyboard accelerator names on page 1773

Reference for keyboard accelerator names to be used in ACCELERATOR* attributes, and in ON KEY / COMMAND KEY clauses in source dialog code.

Defining keyboard accelerators for actions on page 1749

Configuring actions on page 1744

Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with action attributes.

COMMENT action attribute

The COMMENT attribute defines hint for the user about the action.

Syntax 1 (Dialog action handlers and form action defaults):

COMMENT = "\%"string"

1. string is the comment to display, with the % prefix for a localized string.

Syntax 2 (Global .4ad action defaults file):

comment = "string"

1. string is the comment to display. Use <LStr/> child element for a localized string.

Usage

Use the COMMENT attribute to define a description for the action. This text will typically be displayed as a hint for the corresponding action view.

Consider using localized strings with the %"string-id" syntax, if you plan to internationalize your application.

This action attribute can be specified as action default attribute in a global .4ad file, in the ACTION DEFAULTS section of form files, as dialog action attribute, or as action view attribute.

Example

-- As action handler attribute
ON ACTION print ATTRIBUTES(COMMENT="Prints current record")

-- As action default
ACTION DEFAULTS
ACTION print (COMMENT="Print current order information")
END

-- In a form buttom, using a localized string id
BUTTON b1: print, COMMENT=%"actions.print.comment";

-- In a global action defaults file with a localized string id
<ActionDefault name="zoom" comment="Opens a zoom window" ...
  <LStr comment="actions.zoom.comment" />
</ActionDefault>

Related concepts

**COMMENT attribute** on page 1359
The **COMMENT** attribute defines a hint for the user about the form element.

**Localized strings** on page 538
Localized strings provide a means of writing applications in which the text of strings can be customized on site.

**CONTEXTMENU** action attribute
The **CONTEXTMENU** attribute defines whether a context menu option must be displayed for an action.

**Syntax 1 (Dialog action handlers and form action defaults):**

```
CONTEXTMENU = ↓ AUTO ↓ YES ↓ NO ↓
```

**Syntax 2 (Global .4ad action defaults file):**

```
contextMenu = ↓ "yes" ↓ "no" ↓ "auto" ↓
```

**Usage**

**CONTEXTMENU** is an action attribute defining whether the context menu option must be displayed for an action.

**Important:** Actions to be displayed in a context menu must have a **TEXT** attribute. If the **TEXT** attribute is not defined or is empty, the action will not be shown in the context menu.

Possible values for **CONTEXTMENU** are:

1. **NO** indicates that no context menu option must be displayed for this action.
2. **YES** indicates that a context menu option must always be displayed for this action, if the action is visible.
3. **AUTO** means that the context menu option is displayed if no explicit action view is used for that action and the action is visible.

The default is **YES**.

**Note:** With some front-ends, the **CONTEXTMENU=AUTO** attribute will be interpreted as **CONTEXTMENU=YES**. Consider using **CONTEXTMENU=NO** or leave the default (**CONTEXTMENU=YES**).

This attribute applies to the actions defined by the current dialog in the current window. It can be specified as action default attribute in a global .4ad file, in the **ACTION DEFAULTS** section of form files, or as dialog action attribute.

**Example**

```
-- As action handler attribute
ON ACTION zoom ATTRIBUTES(CONTEXTMENU=YES)

-- As action default
ACTION DEFAULTS
  ACTION zoom (CONTEXTMENU=YES)
END
```
Related concepts

**CONTEXTMENU attribute** on page 1359

The `CONTEXTMENU` attribute defines whether a context menu option must be displayed for an action.

**ui.Dialog.setActionHidden** on page 2542

Showing or hiding a default action view.

**DEFAULTVIEW action attribute**

The `DEFAULTVIEW` attribute defines if a default view (a button) must be displayed for a given action.

Syntax 1 (Dialog action handlers and form action defaults):

```plaintext
DEFAULTVIEW = AUTO | YES | NO
```

Syntax 2 (Global .4ad action defaults file):

```plaintext
defaultView = "yes" | "no" | "auto"
```

Usage

`DEFAULTVIEW` is an action attribute defining whether the default action view (a button) must be displayed for an action.

Possible values for `DEFAULTVIEW` are:

- **NO** indicates that no default action view must be displayed for this action.
- **YES** indicates that a default action view must always be displayed for this action, if the action is visible.
- **AUTO** means that a default action view is displayed if no explicit action view is used for that action and the action is visible.

The default is **AUTO**.

This attribute applies to the actions defined by the current dialog in the current window. It can be specified as action default attribute in a global .4ad file, in the `ACTION DEFAULTS` section of form files, or as dialog action attribute.

**Note:** The `DEFAULTVIEW` attribute does not apply to default action views displayed by a `MENU` instruction using the `STYLE="dialog"` or `"popup"` attribute: `DEFAULTVIEW` is related to the action panel of a window, pop-up and dialog menus have no relation to the action panel.

Example

```plaintext
-- As action handler attribute
ON ACTION zoom ATTRIBUTES(DEFAULTVIEW=YES)

-- As action default
ACTION DEFAULTS
  ACTION zoom (DEFAULTVIEW=YES)
END

-- In a global action defaults file
<ActionDefault name="zoom" defaultView="yes" ... />
```

Related concepts

**DEFAULTVIEW attribute** on page 1362
The `DEFAULTVIEW` attribute defines if a default view (a button) must be displayed for a given action.

**Default action views on page 1743**

A default action view is created to render an action handler when no explicit action view exists for it.

**ui.Dialog.setActionHidden on page 2542**

Showing or hiding a default action view.

**DISCLOSUREINDICATOR action attribute**

The `DISCLOSUREINDICATOR` attribute a drill-down decoration to an action.

**Syntax**

(only in `MENU` action handlers)

```plaintext
DISCLOSUREINDICATOR
```

**Usage**

`DISCLOSUREINDICATOR` is an action attribute defining whether a disclosure indicator must be shown for the default view (a button) of an action.

**Important:** This feature is only for the GMI/iOS platform.

A disclosure indicator gives a visual hint to the user, to show that the selection of the action will drill down in the application screens.

The `DISCLOSUREINDICATOR` attribute is typically used in a `MENU` instruction, for options that open a sub-menu.

The rendering of a disclosure indicator depends on the front-end platform standards. On iOS devices, buttons will show a typical > icon on the right.

This attribute can only be specified in a `MENU` dialog, as action attribute in the `ATTRIBUTES()` clause of `ON ACTION` handlers, and applies to the actions defined by the current dialog in the current window.

Note that form buttons can get a `DISCLOSUREINDICATOR` attribute, as an action view decoration.

**Example**

```plaintext
MENU ...
...
ON ACTION details ATTRIBUTES(DISCLOSUREINDICATOR)
    CALL show_customer_details(cust_rec.cust_no)
...
```

**Related concepts**

**DISCLOSUREINDICATOR attribute** on page 1363

The `DISCLOSUREINDICATOR` attribute adds a drill-down decoration to the form item.

**Default action views on page 1743**

A default action view is created to render an action handler when no explicit action view exists for it.

**IMAGE action attribute**

The `IMAGE` attribute defines the image resource to be displayed for the action.

**Syntax 1 (Dialog action handlers and form action defaults):**

```plaintext
IMAGE = "resource"
```
Syntax 2 (Global .4ad action defaults file):

```
image = "resource"
```

1. `resource` defines the file name, path or URL to the image source.

**Usage:**

The `IMAGE` attribute is used to define the image resource for action views such as `BUTTON`, `BUTTONEDIT`, or a `TOOLBAR` button.

For more details about image resource specification, see Providing the image resource on page 1149.

This action attribute can be specified as action default attribute in a global .4ad file, in the ACTION_DEFAULTS section of form files, as dialog action attribute, or as action view attribute.

**Example**

```
-- As action handler attribute
ON ACTION print ATTRIBUTES(IMAGE="printer")

-- As action default
ACTION DEFAULTS
  ACTION print (IMAGE="printer")
END

-- In a form buttonedit or button
BUTTONEDIT f001 = FORMONLY.field01, IMAGE = "zoom"
BUTTON b01: open_file, IMAGE = "buttons/fileopen"
BUTTON b02: accept, IMAGE = "http://myserver/images/accept.png"
```

**Related concepts**

- **IMAGE attribute** on page 1370
  The `IMAGE` attribute defines the image resource to be displayed for the form item.

- **Using images** on page 1147
  Describes how to use pictures in the forms of your application.

- **ROWBOUND action attribute**
  The `ROWBOUND` attribute defines if the action is related to the row context of a record list.

**Syntax**

(only in action handlers of record list dialog)

```
ROWBOUND
```

**Usage**

The `ROWBOUND` attribute is typically used in a DISPLAY ARRAY or INPUT ARRAY dialog action handler, when the action depends on the row context. The actions marked with this attribute will be automatically enabled/disabled based on current row existence, and rendered in a special way depending on the front-end platform standards.

**Important:** This feature is only for mobile platforms.

The `ROWBOUND` attribute was mainly introduced for mobile applications, when using a `TABLE` container to get a list view. Actions marked with this attribute will be rendered in a native manner on the mobile device.

If a default action view is displayed for the action, it is automatically hidden when no current row context is available.

This attribute can only be specified in a list handling dialog, as action attribute in the ATTRIBUTES () clause of ON ACTION handlers, and applies to the actions defined by the current dialog in the current window.
Default actions such as the delete action when using an ON DELETE modification trigger automatically get the ROWBOUND attribute, to be available only when at least one row exists in the list. Therefore, the ROWBOUND attribute cannot be specified for such DISPLAY ARRAY modification triggers.

**Example**

```plaintext
DISPLAY ARRAY ...
  ... 
  ON ACTION print ATTRIBUTES(ROWBOUND)
     CALL print_customer_info(arr_curr())
  ...
```

**Related concepts**

- **Default action views** on page 1743
  A default action view is created to render an action handler when no explicit action view exists for it.

- **TEXT action attribute**
  The TEXT attribute defines the label associated to the action.

**Syntax 1 (Dialog action handlers and form action defaults):**

```plaintext
TEXT = \[ string \]
```

1. *string* defines the label for the action, with the % prefix for a localized string.

**Syntax 2 (Global .4ad action defaults file):**

```plaintext
text = "string"
```

1. *string* is the text to display. Use `<LStr/>` child element for a localized string.

**Usage**

The TEXT attribute is used to define the label associated to an action, for example for a CHECKBOX form field or a BUTTON action view.

Consider using localized strings with the %"string-id" syntax, if you plan to internationalize your application.

This action attribute can be specified as action default attribute in a global .4ad file, in the ACTION DEFAULTS section of form files, as dialog action attribute, or as action view attribute.

**Example**

```plaintext
-- As action handler attribute
ON ACTION print ATTRIBUTES(TEXT="Print")

-- As form action default
ACTION DEFAULTS
  ACTION print (TEXT="Print")
END

-- As a CHECKBOX label
CHECKBOX cb01 = FORMONLY.checkbox01,
  TEXT="OK" ...
;

-- As a BUTTON label, using a localized string id
BUTTON b1: print, TEXT=%"actions.print.label";

-- In a global action defaults file with a localized string id
<ActionDefault name="zoom" text="Zoom" ... >
  <LStr text="actions.zoom.label" />
```
Related concepts

**TEXT attribute** on page 1397
The TEXT attribute defines the label associated with a form item.

**Localized strings** on page 538
*Localized strings* provide a means of writing applications in which the text of strings can be customized on site.

**VALIDATE action attribute**
The VALIDATE action attribute defines the data validation level for a given action.

### Syntax 1: (Dialog action handlers and form action defaults)

```plaintext
VALIDATE = NO
```

### Syntax 2 (Global .4ad action defaults file)

```plaintext
validate = "no"
```

### Usage

When the VALIDATE action attribute is set to NO, it indicates that no data validation must occur for this action. However, current input buffer contains the text modified by the user before triggering the action.

This attribute applies to the actions defined by the current dialog in the current window. It can be specified as action default attribute in a global .4ad file, in the ACTION DEFAULTS section of form files, or as dialog action attribute.

### Example

```plaintext
-- As action handler attribute
ON ACTION undo ATTRIBUTES(VALIDATE=NO)

-- As action default
ACTION DEFAULTS
  ACTION undo (VALIDATE=NO)
END

-- In a global action defaults file
<ActionDefault name="undo" validate="no" ...
```

Related concepts

**VALIDATE attribute** on page 1403
The VALIDATE action attribute defines the data validation level for a given action.

**Data validation at action invocation** on page 1762
The validate action default attribute controls field validation when an action is fired.

### Defining action views in forms

How to define action views that will fire action events.

Actions views are form items that can be activated to fire an action event. The action event triggers user code in an ON ACTION block.

We distinguish action views defined explicitly in form files from **default action views**. A default action view will automatically appear when an action handler is implemented in the current dialog (if no explicit action view with the same name exists in the form). Default action view creation can be controlled with the DEFAULTVIEW action attribute.

To fire user code, action views are bound to **action handlers** by name.
Action view decoration attributes (IMAGE for icons, TEXT for label, COMMENT for hint) can be centralized in action defaults.

Action views can be items of form elements dedicated to action execution, such as TOOLBAR items (toolbar buttons) or TOPMENU options:

```plaintext
TOOLBAR
    ITEM accept
    ITEM cancel
    ...
END
```

Action views can be typical BUTTON items defined in the form LAYOUT:

```plaintext
LAYOUT
GRID
{[b1     ]
    ...
}
...
ATTRIBUTES
BUTTON b1 : print, IMAGE="printer";
...
```

Action views can be sub-elements of other elements, as when defining a BUTTONEDIT with an ACTION attribute:

```plaintext
LAYOUT
GRID
{[f1              ]
    ...
}
...
ATTRIBUTES
BUTTONEDIT f1 = customer.cust_city, ACTION=choose_city, IMAGE="zoom";
...
```

Action views can also be simple IMAGE items, when the ACTION attribute is specified:

```plaintext
LAYOUT
GRID
{[i1              ]
    ...
}
...
ATTRIBUTES
IMAGE i1: image1, ACTION=show_details, IMAGE="mylogo";
...
```

Note that IMAGE fields can be defined as TABLE columns and define the ACTION attribute to trigger user code:

```plaintext
LAYOUT
GRID
{<TABLE t1          >
    [c1   | c2     | c3  ]
    [c1   | c2     | c3  ]
    [c1   | c2     | c3  ]
    ...
}<
```
For more details about image column actions see Image columns firing actions on page 1861.

The row selection in a TABLE (or TREE) will be considered an action view when defining the DOUBLECLICK attribute:

```
DISPLAY ARRAY arr TO sr.*
  ATTRIBUTES(UNBUFFERED, DOUBLECLICK=select)
END DISPLAY
```

Action views can also be graphical elements that are standard action triggers on the front-end platform, such as the [x] cross button of desktop windows, that will automatically bind to a “close” action, or the FAB button of Android, which can be configured to trigger a specific action.

**Related concepts**

Form specification files on page 1237
Form specification files are the source files defining the layout and content of application forms.

**Implementing dialog action handlers**

How to execute user code in ON ACTION blocks when an action is fired.

Actions handlers are typically defined in dialog instructions with the ON ACTION interaction block. You must specify the name of the action after the ON ACTION keywords:

```
INPUT BY NAME ...
  ...
  ON ACTION print
    -- user code
  ...
```

Action handlers can also be defined with the COMMAND syntax in MENU and DIALOG instructions:

```
MENU ...
  ...
  COMMAND "Print" "Print the current record"
    -- user code
  ...
```

ON ACTION blocks provide better abstraction than COMMAND blocks by using simple action identifiers and leaving the decoration in the form files or action defaults files.

The ON ACTION block defines an action handler with a simple action name.

Built-in class methods such as ui.Dialog.setActionActive() take an action name as parameter. For more details, see Identifying actions in ui.Dialog methods on page 2555.

The COMMAND block defines an action handler with an action name, but it also defines decoration attributes, such as the label and comment. Keyboard accelerators and help topic numbers can also be defined.

**Note:** Action views controlled by ON ACTION handlers cannot get the focus. When using the COMMAND action handler, action views such as a BUTTON defined in the form layout can get the focus and are part of the tabbing item list.

Action handlers are bound to action views by name.

**Related concepts**

Action handling basics on page 1739
This topic describes the basics of action views, action events, and action handlers.

**COMMAND [KEY] block** on page 1626

**Binding action views to action handlers**

How are action views of the forms bound to action handlers in the program code?

Action views (such as buttons) are bound to action handlers by the `name` attribute. Action handlers are defined in interactive instructions with an `ON ACTION` clause or `COMMAND / ON KEY` clauses.

For example, in the `ATTRIBUTES` section of the form, a button may be defined as follows:

```
BUTTON b1: show_help, TEXT="Show Help";
```

The corresponding action handler (code) in the program will use the "show_help" action name:

```
ON ACTION show_help
  CALL ShowHelp();
```

Other type of action views can for example be toolbar items:

```
TOOLBAR tb
  ITEM show_help ( TEXT="Show Help" )
  ...;
```

Or `BUTTONEDIT` buttons (using the `ACTION` attribute to define the action name):

```
BUTTONEDIT f1 = customer.cust_city, ACTION = open_city_list;
```

The `COMMAND / ON KEY` clauses are typically used to write text mode programs. Such clauses define the name of the action and the decoration label. It is recommended that you use `ON ACTION` clauses instead, because they identify user actions with an abstract name. However, if required, you can use a `COMMAND` clause in a non-menu dialog to include the corresponding action view in the focusable form items.

In the `ON ACTION action-name` clause, the name of the action must be a valid identifier, preferably written in lowercase letters. In the abstract user interface tree (where the action views are defined), action names are case sensitive (as they are standard DOM attribute values). However, identifiers are not case sensitive in the language. The `fglcomp` compiler always converts the action identifiers of `ON ACTION` clauses to lowercase:

```
ON ACTION PrintRecord   -- will be compiled as "printrecord"
```

To avoid confusion, always use lowercase names for action names (for example, `print_record` instead of `PrintRecord`).

**Related concepts**

- **Sub-dialog actions in procedural DIALOG blocks** on page 1765
This topic describes how action are differentiated with handlers defined in a procedural DIALOG block.

- **Field-specific actions (INFIELD clause)** on page 1767
Using the `INFIELD` clause of `ON ACTION` provides automatic action activation when a field gets the focus.

- **Defining action views in forms** on page 1759
How to define action views that will fire action events.

- **Implementing dialog action handlers** on page 1761
How to execute user code in `ON ACTION` blocks when an action is fired.

**Data validation at action invocation**

The `validate` action default attribute controls field validation when an action is fired.

When using the `UNBUFFERED` mode of interactive instructions such as `INPUT` or `DIALOG`, if the user triggers an action, the current field data is checked and loaded in the target variable bound to the form field. For example, if
the user types a wrong date (or only a part of a date) in a field using a DATE variable and then clicks on a button to invoke an action, the runtime system will display an invalid input error and will not execute the ON ACTION block corresponding to the button.

To prevent data validation for some actions, use the VALIDATE action attribute.

**Note:** The validate action attribute can be set in the global action default file, or at the form level, with the VALIDATE attribute in a line of the ACTION DEFAULTS section.

This validate attribute instructs the runtime not to copy the input buffer text into the program variable (requiring input buffer text to match the target data type).

```plaintext
ACTION DEFAULTS
  ...
  ACTION zoom ( ... VALIDATE = NO ... )
  ...
END
```

This is especially needed in DIALOG instructions; in singular dialogs like INPUT, predefined actions like cancel do not validate the current field value when UNBUFFERED mode is used.

**Related concepts**

- [The buffered and unbuffered modes](#) on page 1720

  The buffered and unbuffered mode control the synchronization of program variables and form fields.

- [ACTION DEFAULTS section](#) on page 1296

  The ACTION DEFAULTS section defines local action view default attributes for the form elements.

**Enabling and disabling actions**

By default, dialog actions are enabled. However, it is recommended that an action be disabled when not allowed in the current context.

Dialog actions are enabled to let the user invoke the action handler (ON ACTION/COMMAND) by clicking on the corresponding action view (button) or by pressing its accelerator key. In most situations, actions remain active during the whole dialog execution. However, to follow GUI standards, actions must be disabled when not allowed in the current context. For example, a print action is disabled if no record is currently shown in the form. After a database query, when the form is filled with a given record, the print action can be activated.

Depending on the front-end ergonomics, the visual result of disabling an action can be different. On desktop front-ends, the action views (buttons) are typically grayed, indicating that the action is there but cannot be triggered. On other front-ends such as some mobile devices, the action view might be hidden, for layout reasons (there is not much space on a mobile device screen).

During a dialog instruction, enable or disable an action with the `setActionActive()` method of the `ui.Dialog` built-in class. This method takes the name of the action (in lowercase letters) and a boolean expression (0 or FALSE, 1 or TRUE) as arguments.

```plaintext
BEFORE INPUT
  CALL DIALOG.setActionActive( "zoom", FALSE )
```

Consider centralizing action activation / deactivation in a setup function specific to the dialog, passing the DIALOG object as the parameter. Centralizing the action activation defines the rules in a single location:

```plaintext
FUNCTION cust_dialog_setup(d)
  DEFINE d ui.Dialog
  DEFINE can_modify BOOLEAN
  LET can_modify = (cust_rec.is_new OR user_info.is_admin)
  CALL d.setActionActive("update", can_modify)
  CALL d.setActionActive("delete", can_modify)
  ...
END FUNCTION
```
Some predefined dialog actions such as insert / append / delete of INPUT ARRAY are automatically enabled/disabled based on the context. For example, if the maximum number of rows (MAXCOUNT) is reached in an INPUT ARRAY, insert and append actions are disabled.

When the action activation depends on the focus being in a specific field, consider using the INFIELD clause of ON ACTION to automatically disable an action if the focus leaves the specified field.

Inside a DIALOG block, actions can be defined at different levels, and may need to be identified with the sub-dialog prefix, when you invoke the ui.Dialog.setActionActive() method outside of the context of the sub-dialog. In the following example, the check_row action must be prefixed by the s_ord sub-dialog name, because setActionActive() is called from the INPUT BY NAME sub-dialog context, to disable an action from the DISPLAY ARRAY sub-dialog:

```dialog
DIALOG ATTRIBUTES(UNBUFFERED)
    DISPLAY ARRAY a_ord TO s_ord.*
    -- sub-dialog-level action
    ON ACTION check_row
        ...
    END DISPLAY
    ...
    INPUT BY NAME rec.* ...
    ON CHANGE consolidation
        -- Must use sub-dialog name to identify the check_row action:
        CALL DIALOG.setActionActive( "s_ord.check_row", FALSE )
    ...
END DIALOG
```

Related concepts
Identifying actions in ui.Dialog methods on page 2555

Hiding and showing default action views
If needed, default action views can be hidden or shown.

When an action is rendered with a default action view (for example, by a button on the action frame of a desktop front-end, or in the top action panel on a mobile front-end), it is sometimes required to hide the action button when the operation is not possible and there is not much space on the screen.

Important: Hiding an action will only make the default action view invisible, if there is a keyboard accelerator associated to the action, it can still fire the action. Consider disabling the action completely with setActionActive().

During a dialog instruction, show or hide an action with the setActionHidden() method of the ui.Dialog built-in class. This method takes the name of the action (in lowercase letters) and a boolean expression (FALSE or TRUE) as arguments:

```dialog
BEFORE INPUT
    CALL DIALOG.setActionHidden( "zoom", TRUE )
```

Consider centralizing action visibility control in a setup function specific to the dialog, passing the DIALOG object as the parameter. Centralizing the action activation defines the rules in a single location:

```dialog
FUNCTION cust_dialog_setup(d)
    DEFINE d ui.Dialog
    DEFINE can_modify BOOLEAN
    LET can_modify = (cust_rec.is_new OR user_info.is_admin)
    CALL d.setActionActive("update", can_modify)
    CALL d.setActionHidden("update", IIF(can_modify,0,1))
    CALL d.setActionActive("delete", can_modify)
    CALL d.setActionHidden("delete", IIF(can_modify,0,1))
    ...
```
Pay attention to multi-level action definitions inside a DIALOG block: Inside a DIALOG block, actions must be hidden/shown with the `ui.Dialog.setActionHidden()` method by specifying a simple action name:

```java
DIALOG ATTRIBUTES(UNBUFFERED)
... 
  BEFORE DIALOG
    CALL DIALOG.setActionHidden( "print", TRUE )
... 
  ON ACTION query
    -- query the database and fill the record
    CALL DIALOG.setActionHidden( "print", (cust_id IS NULL) )
... 
END DIALOG
```

Sub-dialog actions in procedural DIALOG blocks
This topic describes how action are differentiated with handlers defined in a procedural DIALOG block.

We distinguish dialog actions from sub-dialog actions. When the ON ACTION handler is defined at the same level as a BEFORE DIALOG control block, it is a dialog action, and the action name is a simple identifier as in singular interactive instructions:

```
action-name
```

When the ON ACTION handler is defined inside a sub-dialog, or if the action is an implicit action such as `insert` in INPUT ARRAY, it is a sub-dialog action. A sub-dialog action is qualified with the name of the sub-dialog:

```
sub-dialog-name.action-name
```

The INPUT ARRAY and DISPLAY ARRAY sub-dialogs are implicitly identified with the screen-record name defined in the form. For INPUT and CONSTRUCT sub-dialogs, the sub-dialog identifier can be specified with the NAME attribute.

This example defines two 'check' actions in different sub-dialog contexts, and a 'close' action at the dialog level:

```java
DIALOG
  INPUT BY NAME ... ATTRIBUTES (NAME = "cust")
  ON ACTION check -- sub-dialog action "cust.check"
... 
END INPUT
  DISPLAY ARRAY arr_orders TO sr_ord.*
... 
  ON ACTION check -- sub-dialog action "sr_ord.check"
... 
END DISPLAY
  BEFORE DIALOG
... 
  ON ACTION close -- dialog action "close"
... 
END DIALOG
```

By using the sub-dialog identifier in form definition files, you can bind action views to specific sub-dialog actions. Action views bound to sub-dialog actions with qualified sub-dialog action names will always be active, even if the focus is not in the sub-dialog of the action. You typically use fully-qualified sub-dialog action names for buttons in the form body or in topmenu options. However, it does not make much sense to use this technique for toolbar buttons, where buttons must be enabled/disabled based on the context.
If you bind an action view with a simple action name (without the sub-dialog prefix), the action view will be attached to any sub-dialog action with the matching name. This is especially useful for common actions such as the implicit insert / append / delete actions created by `INPUT ARRAY`, when the dialog handles multiple editable lists. Bind toolbar buttons to these actions without the sub-dialog prefix; the buttons will apply to the current list that has the focus. The action views bound to sub-dialog actions without the sub-dialog qualifier will automatically be enabled or disabled when entering or leaving the group of fields controlled by the sub-dialog (typical navigation buttons in the toolbar will be disabled if the focus is not in a list).

If a sub-dialog action is invoked when the focus is not in the sub-dialog of the action, the focus will automatically be given to the first field of the sub-dialog, before executing the user code defined in the `ON ACTION` clause. This will trigger the same validation rules and control blocks as if the user had selected the first field of the sub-dialog by hand.

When using `DIALOG.setActionActive()` (or any method that takes an action name as parameter), you can specify the action name with or without a sub-dialog identifier. If you qualify the action with the sub-dialog identifier, the sub-dialog action is clearly identified. If you don't specify a sub-dialog prefix, the action will be identified based on the focus context - when the focus is in the sub-dialog of the action, non-qualified action names identify the local sub-dialog action; otherwise, they identify a dialog action if one exists with the same name. Disabling an action by the program with `setActionActive()`, will take precedence over the built-in activation rules (this means that if the action is disabled by the program, the action will not be activated when entering the sub-dialog).

For action views bound to sub-dialog actions with qualifiers, the action defaults defined with the corresponding action name will be used to set the attributes with the default values. In other words, the prefix will be ignored. For example, if an action view is defined with the name "custlist.append", it will get the action defaults defined for the "append" action.

**Related concepts**

Configuring actions on page 1744
Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with *action attributes*.

Topmenus on page 1444
Topmenus define typical pull-down menus that appear at the top of application forms.

Toolbars on page 1436
Toolbars define a bar of buttons that appears at the top of application forms.

**Field-specific actions (INFIELD clause)**

Using the INFIELD clause of ON ACTION provides automatic action activation when a field gets the focus.

The ON ACTION interaction block of INPUT, CONSTRUCT and INPUT ARRAY (as singular dialogs or sub-dialogs in DIALOG instruction), can be specified with the INFIELD field-name clause. With this clause, the action will only be active when the focus is on one of the fields. The same action name can be used for several fields.

```plaintext
INPUT ARRAY custarr WITHOUT DEFAULTS FROM sr_cust.*
  ON ACTION zoom INFIELD cust_city
    LET custarr[arr_curr()].cust_city = zoom_city()
  ON ACTION zoom INFIELD cust_state
    LET custarr[arr_curr()].cust_state = zoom_state()
END INPUT
```

Actions defined with the INFIELD field-name clause can be identified with the field name as prefix:

`field-name.action-name`

You can bind form action views with or without the field name prefix:

```plaintext
-- Form file:
EDIT f1 = customer.cust_name;
BUTTON b1 : cust_name.clear, ... ;    -- field-qualified action
BUTTON b2 : clear, ... ;              -- unqualified action

-- Program code:
ON ACTION clear INFIELD cust_name
...
```

- Without the field name prefix, the action view is enabled and disabled automatically depending on the current field: The action view is enabled when the corresponding INFIELD field has the focus.
- When binding the action view with the fully-qualified name including the field name prefix, the action view will always be active, and the focus will jump to the first corresponding field, if the action is fired.

**Note:**

When using ON ACTION action-name INFIELD field-name for one or several BUTTONEDIT fields, the runtime system implicitly handles the BUTTONEDIT action as a field-qualified action, even if the ACTION attribute is defined without the field name:

```plaintext
BUTTONEDIT f1 = customer.cust_city, ACTION = zoom;
```

Is equivalent to:

```plaintext
BUTTONEDIT f1 = customer.cust_city, ACTION = cust_city.zoom;
```

Actions defined in sub-dialogs of the DIALOG instruction get the name of the sub-dialog as prefix. If ON ACTION action-name INFIELD field-name is used in a sub-dialog, the action object name is prefixed with the name of the sub-dialog, followed by the name of the field. The fully-qualified action name will be:

`sub-dialog-name.field-name.action-name`

When the field-specific action is invoked (for example by a button of the toolbar bound with the fully-qualified action name) and if the field does not have the focus, the runtime system first selects that field before executing the code of the ON ACTION INFIELD block. The field selection forces data validation and AFTER FIELD of the current field, followed by BEFORE FIELD of the target field associated to the action.
It's still possible to enable and disable field-specific action objects by the program using the `DIALOG.setActionActive()` method. When specifying a fully-qualified action name with the field name prefix, that field-specific action will be enabled or disabled. When disabled by the `setActionActive()` method, the corresponding action views will always be disabled, even if the field has the focus. If you do not specify a fully-qualified name in the method call, and if several actions are defined with the same action name in different sub-dialogs and/or using the `INFIELD` clause, the method will identify the action according to the current focus context. For example, if you define `ON ACTION zoom INFIELD cust_city` and `ON ACTION zoom INFIELD cust_addr`, when the focus is in `cust_city`, a call to `DIALOG.setActionActive("zoom", FALSE)` will disable the action specific to the `cust_city` field.

Fields can be enabled or disabled dynamically with the `DIALOG.setFieldActive()` method. If an `ON ACTION` `INFIELD` is declared on a field and if you enable/disable the field dynamically, then the field-specific action (and corresponding action views in the form) will be enabled or disabled accordingly.

For action views bound to field actions with qualifiers, the action defaults defined with the corresponding action name will be used to set the attributes with the default values. In other words, the prefix will be ignored. For example, if an action view is defined with the name "`cust_addr.check`", it will get the action defaults defined for the "`check`" action.

**Related concepts**

**Configuring actions** on page 1744
Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with *action attributes*.

**Topmenus** on page 1444
Topmenus define typical pull-down menus that appear at the top of application forms.

**Toolbars** on page 1436
Toolbars define a bar of buttons that appears at the top of application forms.

**Multilevel action conflicts**

Actions can be defined at two levels in a singular dialog, and three levels in the context of a `DIALOG` block:

1. Dialog level
2. Sub-dialog level (procedural `DIALOG` only)
3. Field level (`ON ACTION` with `INFIELD` clause)

It is not good practice to use the same action name at different levels of a dialog: This makes action view bindings and action handling (enabling / disabling) very complex, because there are many possible combinations. Therefore, when using the same action name at different dialog levels, the `fglcomp` compiler will raise a warning -8409. However, it is legal to use the same action name for a given level of action handlers in sub-dialogs or for field-actions. For example, using the "`zoom`" action name for multiple `ON ACTION` `INFIELD` handlers is a common practice.

When binding action views with full qualified names, the `ON ACTION` handler is clearly identified, and the corresponding user code will be executed. However, when you do not specify the complete prefix of a sub-dialog or field action, the runtime system searches for the best `ON ACTION` handler to be executed, according to the current focus context.

Take for example a `DIALOG` instruction defining three `ON ACTION` `print` handlers at the dialog, sub-dialog and field level:

```dialog
DIALOG
  INPUT BY NAME ... ATTRIBUTES (NAME = "cust")
  ...
  ON ACTION print INFIELD cust_name -- field-level action (1)
  ...
  ON ACTION print -- sub-dialog-level action (2)
  ...
END INPUT
  ...
ON ACTION print -- dialog-level action (3)
```
The action views of the form will behave as follows:

- Action views bound with the name "print" will always be active, and invoke the ON ACTION print handler corresponding to the current focus context:
  - (1) is invoked if the focus is in the cust_name field.
  - (2) is invoked if the focus is in the cust sub-dialog, but not in cust_name field.
  - (3) is invoked if the focus is in another sub-dialog as cust sub-dialog.
- Action views bound with the name "cust.print" will always be active, even if the focus is not the cust sub-dialog, and invoke the ON ACTION print handler depending on the focus context:
  - (1) is invoked if the focus is in the cust_name field.
  - (2) is invoked if the focus is in the cust sub-dialog, but not in cust_name field.
- Action views bound with the name "cust.cust_name.print" will always be active, and invoke the ON ACTION print INFIELD cust_name handler after giving the focus to the cust_name field.

If the first field of a sub-dialog defines an ON ACTION INFIELD with the same action name as a sub-dialog action, and the focus is not in that sub-dialog when the user selects an action view bound with the name sub-dialog-name.action-name, the runtime system gives the focus to the first field of the sub-dialog. This field becomes the current field, and the runtime system executes the field-specific action handler instead of the sub-dialog action handler.

To avoid mistakes and complex combinations, it is recommended that you use specific action names for each dialog level.

**Action views with GBC**

With GBC, default action views and toolbar action views can be displayed in the GBC chrome bar to save space on small screens.

The *browser chrome* includes the control widgets of the web browser window (menus, toolbars, scroll bars and URL address bar), surrounding the HTML content.

The *GBC chrome* is the control bar of the GBC front-end, which is by default displayed with a blue background on the top of the Genero application forms.

On iOS and Android™ mobile devices, when using the GBC front-end (for example with Universal Rendering), the default action views of the action panel and ring menu panel, as well as the toolbar action views can be displayed in the GBC chrome.

On mobile devices, this will follow the Material Design specification.
Figure 86: Google Nexus 5 display with default action views in the GBC chrome
By default, on mobile devices, GBC will implicitly display these action views in the GBC chrome. This rendering corresponds to the following style attribute settings:

```xml
<Style name="Window">
  <StyleAttribute name="actionPanelPosition" value="chrome" />
  <StyleAttribute name="ringMenuPosition" value="chrome" />
  <StyleAttribute name="toolBarPosition" value="chrome" />
</Style>
```

The actions will be rendered in the following order in the GBC chrome bar:

1. Toolbar action views
2. Default action views of the action panel or ring menu panel
3. Common GBC chrome actions (Application information, Settings, Bookmarks, Close window)

When there is not enough room in the GBC chrome, the action views will be rendered in a vertical drop down menu that can be opened from a three-dots button on the right of the GBC chrome bar. This drop down menu will replace the default GBC drop down menu that shows up on small webviews.

**Note:** When using the tabbed container layout option, the "chrome" style attribute value will be ignored, and the desktop behavior will be enforced. See `tabbedContainer` style attribute.

On mobile, to get the same default rendering as on a desktop browser, use following settings:

```xml
<Style name="Window">
  <StyleAttribute name="actionPanelPosition" value="right" />
  <StyleAttribute name="ringMenuPosition" value="right" />
  <StyleAttribute name="toolBarPosition" value="top" />
</Style>
```

**Note:** When using tabbed containers with the `tabbedContainer="yes"` style attribute, the "chrome" value is ignored for the `actionPanelPosition`, `ringMenuPosition` and `toolBarPosition` style attributes. If `toolBarPosition` is set to "chrome", it will fallback to the default "top" value. If `actionPanelPosition` or `ringMenuPosition` is set to "chrome", it will fallback to the default "right" value.

**Related concepts**
Toolbars on mobile devices on page 1439

Toolbars can be used to control action view rendering on mobile devices.

**Action display in the context menu**
The `CONTEXTMENU` action default attribute allows you to control action visibility in the context menu.

Some front-ends can display a context menu, with all the active actions that are possible in the current form. Displaying all actions might not be adapted to your needs. To control if an action must be displayed in the context menu, set the `CONTEXTMENU` attribute in action defaults. Values for `CONTEXTMENU` can be YES, NO and AUTO.

```
ACTION_DEFAULTS
... 
ACTION insert ( ... CONTEXTMENU = YES ... )
ACTION append ( ... CONTEXTMENU = YES ... )
ACTION delete ( ... CONTEXTMENU = YES ... )
... 
ACTION validate_order ( ... CONTEXTMENU = NO ... )
... 
END
```

**Related concepts**
Configuring actions on page 1744
Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with action attributes.

**Implementing the close action**
The close action is a predefined action dedicated to close graphical windows (for example, with the X cross button).

**Purpose of the close action**
In graphical applications, windows can be closed by the user, for example by pressing Alt+F4 or by clicking the cross button in the upper-left corner of the window.

A predefined action named "close" is dedicated to this specific event.

When the end user closes a graphical window, the program gets a close action. It is then possible to execute user code in an ON ACTION close action handler, or leave the default behavior.

**Note:** The default action view (that is the button in the action frame) of the close action is hidden.

**The close action in multiple dialogs**
When executing a DIALOG instruction, the close action executes the ON ACTION close block, if defined. Otherwise, the close action is mapped to the cancel action if an ON ACTION cancel handler is defined.

If neither ON ACTION close, nor ON ACTION cancel are defined, nothing will happen if the user tries to close the window with the X cross button or an ALT+F4 keystroke.

The INT_FLAG register will not be set in the context of DIALOG.

**The close action in singular dialogs**
When an ON ACTION close handler is defined in an INPUT, INPUT ARRAY, CONSTRUCT, DISPLAY ARRAY or PROMPT interactive instruction, the handler code will be executed if the close action is fired.

If no explicit ON ACTION close handler is defined, the close action acts the same as the cancel predefined action. So by default when the user clicks the X cross button in a window, the interactive instruction stops and the INT_FLAG is set to 1.

If there is an explicit ON ACTION cancel block defined, INT_FLAG is set to 1 and the user code under ON ACTION cancel will be executed.

If the CANCEL=FALSE option is set, no cancel and no close action will be created, and you must write an ON ACTION close handler to proceed with the close action. In this case, the INT_FLAG register will not be set when the close action is invoked.

**The close action in MENU dialogs**
When an ON ACTION close handler is defined in a MENU statement, the handler code will be executed if the close action is fired.

If no explicit ON ACTION close action handler is defined, the code of the COMMAND KEY (INTERRUPT) or ON ACTION cancel will be executed, if defined.

If neither COMMAND KEY (INTERRUPT) nor ON ACTION cancel is defined:

- If the MENU uses the default rendering (as buttons in current window action frame) or with the attribute STYLE="dialog", nothing happens and the program stays in the MENU instruction.
- If the MENU is defined with the attribute STYLE="popup", there is no X cross button to click. However, the Escape key, or a click outside the popup menu will terminate the MENU dialog.

Regarding the INT_FLAG register, its value is unknown after a MENU instruction.

**The close action on mobile devices**
When displaying on Android™ and iOS mobile device, the 'close' action can be bound to the "Back button".
For more details, see The "Back" button on Android devices on page 1778 and The "Back" button on iOS devices on page 1778.

**Example**

You typically implement a close action handler to open a confirmation dialog box as in the following example:

```plaintext
INPUT BY NAME cust_rec.*
  ...
  ON ACTION close
     IF msg_box_yn("Are you sure you want to close this window?") == "y" THEN
       EXIT INPUT
     END IF
  ...
END INPUT
```

**Related concepts**

*Predefined actions* on page 1740

Genero predefines some action names for common operations of interactive instructions.

**Keyboard accelerator names**

Reference for keyboard accelerator names to be used in `ACCELERATOR*` attributes, and in `ON KEY / COMMAND KEY` clauses in source dialog code.

**ON KEY Virtual keys**

Virtual keys are the key names that can be used in program instructions such as `ON KEY` and `COMMAND KEY`.

An `ON KEY` block defines one to four different action objects that will be identified by the key name in lowercase (`ON KEY(F5,F6) = creates Action f5 + Action f6`). Each action object will get an `acceleratorName` attribute assigned. In GUI mode, **action defaults** are applied for `ON KEY` actions by using the name of the key. You can define secondary accelerator keys, as well as default decoration attributes like button text and image, by using the key name as action identifier. The action name is always in lowercase letters.

Check carefully the `ON KEY CONTROL-?` statements because they may result in having duplicate accelerators for multiple actions due to the accelerators defined by action defaults. Additionally, it is recommended to avoid using `ON KEY` statements with `ESC`, `TAB`, `UP`, `DOWN`, `LEFT`, `RIGHT`, `HELP`, `NEXT`, `PREVIOUS`, `INSERT`, `CONTROL-M`, `CONTROL-X`, `CONTROL-V`, `CONTROL-C`, and `CONTROL-A` in GUI programs, because they are very likely to clash with default accelerators defined in the action defaults.

By default, `ON KEY` actions are not decorated with a default button in the action frame (**default action view**). You can show the default button by configuring a `text` attribute with the action defaults.

**Table 386: Names of keys to be referenced in programs**

<table>
<thead>
<tr>
<th>Key Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCEPT</td>
<td>The validation key.</td>
</tr>
<tr>
<td>INTERRUPT</td>
<td>The interruption key.</td>
</tr>
<tr>
<td>ESC or ESCAPE</td>
<td>The ESC key (not recommended, use ACCEPT instead).</td>
</tr>
<tr>
<td>TAB</td>
<td>The TAB key (not recommended).</td>
</tr>
<tr>
<td>Control-char</td>
<td>A control key where <code>char</code> can be any character except A, D, H, I, J, K, L, M, R, or X.</td>
</tr>
<tr>
<td>F1 through F255</td>
<td>A function key.</td>
</tr>
<tr>
<td>DELETE</td>
<td>The key used to delete a new row in an array.</td>
</tr>
<tr>
<td>INSERT</td>
<td>The key used to insert a new row in an array.</td>
</tr>
<tr>
<td>HELP</td>
<td>The help key.</td>
</tr>
</tbody>
</table>
### Table 387: Keyboard key names

<table>
<thead>
<tr>
<th>Key Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>Special name indicating the runtime system must not set any accelerator for the action.</td>
</tr>
<tr>
<td>0–9</td>
<td>Decimal digit keys from [0] to [9]</td>
</tr>
<tr>
<td>A–Z</td>
<td>Letters keys from [A] to [Z]</td>
</tr>
<tr>
<td>F1–F35</td>
<td>The functions keys like [F10]</td>
</tr>
<tr>
<td>BackSpace</td>
<td>The [BACKSPACE] key (do not confuse with [DELETE] key)</td>
</tr>
<tr>
<td>Delete</td>
<td>The [DELETE] key (navigation keyboard group)</td>
</tr>
<tr>
<td>Down</td>
<td>The [DOWN] key (arrow keyboard group)</td>
</tr>
<tr>
<td>End</td>
<td>The [END] key (navigation keyboard group)</td>
</tr>
<tr>
<td>Enter</td>
<td>The [ENTER] key (numeric keypad, see Note)</td>
</tr>
<tr>
<td>Escape</td>
<td>The [ESCAPE] key</td>
</tr>
<tr>
<td>Home</td>
<td>The [HOME] key (navigation keyboard group)</td>
</tr>
<tr>
<td>Insert</td>
<td>The [INSERT] key (navigation keyboard group)</td>
</tr>
<tr>
<td>Left</td>
<td>The [LEFT] key (arrow keyboard group)</td>
</tr>
<tr>
<td>Minus</td>
<td>The [-] minus sign key (Hyphen)</td>
</tr>
<tr>
<td>Next</td>
<td>The [PAGE UP] key (navigation keyboard group)</td>
</tr>
<tr>
<td>Prior</td>
<td>The [PAGE DOWN] key (navigation keyboard group)</td>
</tr>
<tr>
<td>Return</td>
<td>The [RETURN] key ( alphanumeric keypad, see Note)</td>
</tr>
<tr>
<td>Right</td>
<td>The [RIGHT] key (arrow keyboard group)</td>
</tr>
<tr>
<td>Space</td>
<td>The [SPACEBAR] key</td>
</tr>
</tbody>
</table>
**User interface**

<table>
<thead>
<tr>
<th>Key Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tab</td>
<td>The [TAB] key</td>
</tr>
<tr>
<td>Up</td>
<td>The [UP] key (arrow keyboard group)</td>
</tr>
</tbody>
</table>

**Note:** The "Enter" accelerator key name represents the [ENTER] key available on the numeric keypad of standard keyboards, while "Return" represents the [RETURN] key of the alphanumeric keyboard. By default, the "accept" validation action is configured to accept both "Enter" and "Return" keys.

**Related concepts**

- **Configuring actions** on page 1744
  Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with *action attributes*.

- **Predefined actions** on page 1740
  Genero predefines some action names for common operations of interactive instructions.

**Accelerator key modifiers**

Key modifiers define keyboard control key combinations in accelerator key names.

The accelerator key names such as "F10" can be combined with key modifiers, by using a minus sign (hyphen) as separator:

```
key-modifier - | key-modifier - | key-modifier - | key-name
```

**Table 388: Names of key modifiers**

<table>
<thead>
<tr>
<th>Key Modifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
<td>The left or right [Ctrl] key.</td>
</tr>
<tr>
<td>Shift</td>
<td>The left or right [Shift] key.</td>
</tr>
<tr>
<td>Alt</td>
<td>The left or right [Alt] key.</td>
</tr>
</tbody>
</table>

For example:

- Control-P
- Shift-Alt-F12
- Control-Shift-Alt-Z

**Related concepts**

- **Accelerator key names** on page 1774
  Accelerators keys are attributes defining the keyboard shortcuts for actions.

**Setting action key labels**

Labels can be defined to decorate buttons controlled by *ON KEY / COMMAND KEY* action handlers.

**Syntax**

Key label configuration can take place at different levels.

- FGLPROFILE definitions
  ```
  key.key-name.text = "label"
  ```

- Program-level key labels
  ```
  CALL fgl_setkeylabel( "key-name", "label" )
  ```
• Form level key labels (in KEYS section)

```
KEYS key-name = [¶]"label"
[...]
[END]
```

• Dialog level key labels

```
CALL fgl_dialog_setkeylabel( "key-name", "label" )
```

• Form field level key labels (in field definition)

```
KEY key-name = [¶]"label"
```

1. `key-name` is the name of the key.
2. `label` is the text to be displayed in the default action view (button).

**Usage**

In GUI mode, ON KEY and COMMAND KEY action handlers in dialogs can be shown as form buttons when a label text is defined for the key. By defining a label for a key, the runtime system will automatically show a default button for the key action.

**Important:** Key label configuration is provided for backward compatibility. Consider using action configuration in new programs. However, key labels can be used to easily improve the graphical rendering of your application, without touching legacy code using ON KEY / COMMAND KEY in dialogs.

In the example, the function key F10 is used to show a detail window in this interactive dialog:

```
INPUT BY NAME myrecord.*
  ON KEY (F10)
    CALL ShowDetail()  
END INPUT
```

By default, if you do not specify a label for F10, no default action button is displayed for a function key or control key. Furthermore, if the text provided for the key label is empty or null, the default action button will not be displayed.

In order to get a default action view button for F10, define for example the KEY attribute in the form file, for the corresponding fields where this action can be fired:

```
ATTRIBUTES
... 
  f07 = customer.cust_city, KEY F10 = "City list";  
  f08 = customer.cust_state, KEY F10 = "State list";  
...
```

**Table 389: Key names recognized by the runtime system**

<table>
<thead>
<tr>
<th>Key Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1 to f255</td>
<td>Function keys.</td>
</tr>
<tr>
<td>control-a to control-z</td>
<td>Control keys.</td>
</tr>
<tr>
<td>accept</td>
<td>Predefined dialog validation action.</td>
</tr>
<tr>
<td>interrupt</td>
<td>Predefined dialog cancellation action. The action name is <code>cancel</code>, not <code>interrupt</code>.</td>
</tr>
<tr>
<td>insert</td>
<td>Predefined INPUT ARRAY dialog row insertion action.</td>
</tr>
</tbody>
</table>
### Key Name | Description
--- | ---
append | Predefined INPUT ARRAY dialog row addition action.
delete | Predefined INPUT ARRAY dialog row deletion action.
help | Predefined help action.

Key labels can be defined at different levels. The order of precedence for key label definition is the following:

1. The label defined with the **KEY** attribute of the form field.
2. The label defined for the current dialog, using the **FGL_DIALOG_SETKEYLABEL** function.
3. The label defined in the **KEYS** section of the form specification file.
4. The label defined as default for a program, using the **FGL_SETKEYLABEL** function.
5. The label defined in the **FGLPROFILE** configuration file (`key.key-name.text` entries).

In Genero, you typically define action labels with action attributes. However, if key labels are defined, they will overwrite the text defined in action attributes for the corresponding key action. In BDS 3.xx versions, default key labels are defined in `$FGLDIR/etc/fglprofile`. These defaults have been commented out in Genero to have action attribute text applied (In Genero, by default, `fgl_getkeylabel()` returns NULL for all keys). If you want to get the same default key labels as in BDS 3.xx, uncomment the key.* lines in `$FGLDIR/etc/fglprofile`.

You can query the label defined at the program level with the **FGL_GETKEYLABEL** function and, for the current interactive instruction, with the **FGL_DIALOG_GETKEYLABEL** function.

**Related concepts**
- **Configuring actions** on page 1744
- Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with *action attributes*.
- **ON KEY block** on page 1465
- **COMMAND KEY() block** on page 1477
- **The FGLPROFILE file(s)** on page 255
- **FGLPROFILE** environment variable defines Genero BDL configuration files

**Action views on mobile devices**
Action views are rendered following mobile specific standards.

**Overview of action views on mobile devices**
This is a short overview about action view rendering with the front-ends for mobile devices.

**Action views on mobile in Native Rendering and Universal Rendering mode**
Action view rendering will depend on the **Native Rendering mode** versus **Universal Rendering mode**.

- When using the Native Rendering mode, action views are rendered by GMA and GMI frond-ends with native platform widgets, as described in the next topics.
- When using the Universal Rendering mode, action views are rendered with GBC front-end rules.

**Action view rendering on mobile**
The top and/or bottom part of the app screen can be used to place action views (UI button controls).

Key functions of these areas:

- Make important actions prominent and accessible in a predictable way (such as **New** or **Search**).
- Support consistent navigation and view switching within apps.
- Reduce clutter by providing an action overflow for less-used actions.
- Provide a dedicated space for giving your app an identity with text and/or an image.

How actions are rendered on the mobile device depends on:

- The order of the **ON ACTION** statements in the current dialog of the running app.
• The type of platform (Android™/iOS).
• The type (phone/tablet) and orientation of the device.
• The usage of Native Rendering or Universal Rendering mode.

Actions are mapped to the Android™ or iOS platform in a specific way, following the platform standard. Actions can be programmatically enabled and disabled, and hidden and shown.

The text, image and other properties of the action can be controlled with action attributes.

**GUI elements to trigger actions on mobile devices**

In the Native Rendering mode, each mobile platform provides its own standard to display action triggers.

GMA and GMI follow respectively the Android™ and iOS standards:

• iOS navigation controller and toolbar pane on page 1779
• Android action bar on page 1785
• Android floating action button on page 1787

**Decorating action views on mobile**

Actions are typically decorated using the IMAGE or the TEXT action attribute. If these attributes are not defined or if the specified image resource is not available, the mobile front-end uses a default decoration in the Native Rendering mode.

For some actions, the front-end always uses the platform-specific decoration. For example, on iOS devices, the "refresh" action always renders as a typical circular arrow icon.

Well-known actions use a default icon or text corresponding to the mobile platform GUI guidelines. As these follow the mobile OS standards, do not define your own text or icons for common actions such as "accept" or "cancel".

For a complete list of predefined action decorations, see:

• iOS default action views decoration on page 1784
• Android default action views decoration on page 1788

**The "Back" button on Android™ devices**

In the Native Rendering mode, the physical back button on an Android™ device is considered a default action view for the "close", "cancel", or "accept" action in the current dialog:

• If a close action is defined and active, it is assigned to the back button.
• If no active close action exists in the current dialog, but the cancel action is defined and active, it is assigned to the back button.
• If neither close nor cancel actions are defined / active, but the accept action is defined and active, it is assigned to the back button.

If accept or cancel cannot be assigned to the back button, a default action view appears in the action panel. For example, if all three actions (close, cancel and accept) exist and are active, the action panel shows a check mark for the accept action and a cross icon for the cancel action, while the back button fires the close action.

**The "Back" button on iOS devices**

In the Native Rendering mode, the physical back button on an iOS device is considered a default action view for the "close", "cancel", or "accept" action in the current dialog:

• If a cancel action is defined and active, it is assigned to the back button.
• If no active cancel action exists in the current dialog, but a close action is defined and active, it is assigned to the back button.
• If neither close nor cancel actions are defined / active, but accept action is defined and active, it is assigned to the back button.
• Exception with single DISPLAY ARRAY dialog: The accept action binds by default to the back button if it is not bound as DOUBLECLICK action. This is intentional to match the IOS behavior for simple selection lists.

**Default action views rendering with MENU**

On mobile devices, the rendering of the MENU dialog depends on whether or not the current window has a form.

If a MENU is active and the current Window has no form, then the MENU is shown as a list of actions.

If a MENU is active and has a FORM, then the menu actions are rendered like all other dialog actions.

**Default and toolbar action views with GBC (Universal Rendering)**

When using the GBC on mobile with Universal Rendering, the default action views of the action panel, ring menu panel and the toolbar action views are by default displayed in the GBC chrome bar, to adapt to mobile device GUI standards.

For more details, see Action views with GBC on page 1769.

**iOS navigation controller and toolbar pane**

On iOS devices, apps can display a navigation controller and a toolbar pane.

The iOS navigation controller is made of a navigation bar, an optional menu icon and a common action pane (the current window/form title is displayed in the middle of these UI controls).

The iOS toolbar pane displays at the bottom of the screen to show application options.
Figure 87: iOS app interface
Navigation bar (1)  The left side navigation bar provides a linear path through various screens. The accept, cancel or close action is rendered as the [<] back button, respectively in the order of precedence given here. If there is a previous form or window, then this button shows the title of the previous page. If there is not a form to return to, the "back" navigation button is not shown. See also The "Back" button on iOS devices on page 1778.

Menu icon (2)  The three-bars icon is the menu icon. This icon appears when the current form defines a TOPMENU. The menu options are displayed in a listview, when the user taps on the menu icon.

Common action pane (3)  The right-hand side is the common action pane. Default action views are displayed here, in the same order as the ON ACTION blocks of the current dialog.

Toolbar pane (4)  When default action views are displayed, if there is not enough room in the common action pane (3), the remaining actions are displayed in the toolbar pane at the bottom of the screen. If there is not enough space to display all action views in the toolbar pane, an overflow icon appears on the right. Tap on the overflow icon to show the remaining action views.

Use a TOOLBAR in your form, to have full control on the toolbar pane: An action displayed as a TOOLBAR item in the toolbar pane will no longer display as default action view in the common action pane (3).

In the following screenshot, the iOS device is oriented in landscape mode. The app is the same, yet since there is enough space in the navigation bar, all default action views display in the common action pane.
To customize the application, define the colors of the iOS navigation bar and toolbar with the following Window-class style attributes:

- `iosTintColor`, for items in (1), (2), (3) and (4) (and for other form items)
- `iosNavigationBarTintColor`, for (1), (2) and (3).
- `iosToolBarTintColor`, for (4).

For example, by setting the following style attributes, the navigation bar will render as shown in the screenshot:

```xml
<Style name="Window">
  <StyleAttribute name="iosTintColor" value="darkRed" />
  <StyleAttribute name="iosNavigationBarTintColor" value="orange" />
  <StyleAttribute name="iosToolBarTintColor" value="orange" />
  <StyleAttribute name="iosTabBarTintColor" value="orange" />
</Style>
```
Figure 89: iOS (7) colored navigation bar
**iOS default action views decoration**

Common default action views get a decoration implicitly, following iOS standards.

On iOS devices, the decoration for well known actions can be a symbol or text. When text is used, it is internationalized. For example, the "accept" action translates to "Done" when the mobile language is English, "Fertig" in German, and "OK" in French.

For the default action views of the common actions, the decoration will by default follow the iOS standards, even if an attribute is explicitly specified for the action. For example, if you implement an ON ACTION save action handler with ATTRIBUTES(TEXT="Write", IMAGE="disk"), the action view renders with the "Save" text on an iOS device configured for the English language.

To bypass the iOS standard decoration and render default action views with text and images defined by the corresponding action attributes, use the `iosRenderSystemActions` presentation style attribute with the value "no".

**Table 390: Default rendering for common actions on iOS**

<table>
<thead>
<tr>
<th>Action name</th>
<th>iOS default rendering</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>accept</td>
<td>Internationalized text (English: Done)</td>
<td>N/A</td>
</tr>
<tr>
<td>cancel</td>
<td>Internationalized text (English: Cancel)</td>
<td>N/A</td>
</tr>
<tr>
<td>refresh</td>
<td>Typical circular symbol</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>Typical plus sign symbol</td>
<td></td>
</tr>
<tr>
<td>append</td>
<td>Typical plus sign symbol</td>
<td></td>
</tr>
<tr>
<td>delete</td>
<td>Typical trash symbol</td>
<td></td>
</tr>
<tr>
<td>find</td>
<td>Typical magnifier symbol</td>
<td></td>
</tr>
<tr>
<td>Action name</td>
<td>iOS default rendering</td>
<td>Symbol</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>search</td>
<td>Typical magnifier symbol</td>
<td>![Search Symbol]</td>
</tr>
<tr>
<td>edit</td>
<td>Internationalized text (English: Edit)</td>
<td>N/A</td>
</tr>
<tr>
<td>save</td>
<td>Internationalized text (English: Save)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Android™ action bar**

On Android devices, apps show an action bar.

The Android™ action bar displays in the top of the screen, with several elements having a specific purpose:

1. **The app icon (1)**
   - The app icon and the title of the current form display in the upper left corner.
   - The application title is defined by the TEXT attribute of the main window displayed by the application.
   - The icon that appears is either the icon set for the app in the packaging, or it is the image specified by the ui.Interface.setImage method. The application icons must be included in the deployment package (.apk) and follow the Android™ standards (several icon sizes are required).

2. **The view control (2)**
   - If your app implements different views controlled by a top-level navigator, this segment allows users to switch between views. For more details, see Navigator pane on
page 1904. In an application handling multiple views in parallel, the view control item displays as a text button.

**Figure 90: Android View Control**

The right-hand side of the action bar shows the actions. The action buttons (3) show the most important actions of your app. Actions that do not fit in the action bar are moved to the action overflow, and an overflow icon appears on the right. Tap on the overflow icon to display the list of remaining action views. If the device has a physical Menu button, the overflow actions are accessible by pressing the physical Menu button and not from an action overflow icon.

Actions display in the order of the `on action` statements of the current dialog. If a `toolbar` is defined, the actions defined in the toolbar take priority and list prior to other actions, in the order they are defined in the toolbar.

If an image is available, it is displayed, otherwise the action text is shown. Depending on the space available (space used by the app icon, screen size, orientation, and so on), the number of actions and the device type,
Android displays either the icon or the icon and the text of the action.

**Android floating action button**

On Android devices, apps using material design show a Floating Action Button (FAB).

The Android floating action button displays on the bottom right of the screen, and can be tapped to fire a specific action.

**Figure 91: The floating action button (1)**

The material design guidelines include the concept of promoted actions, that can be triggered with the floating action button.

Define the list of actions that can be fired from the FAB button with FAB configuration style attributes:

```xml
<Style name="Window">
  <StyleAttribute name="materialFABActionList" value="accept,select,detail" />
</Style>
```

The order of the actions define which action is triggered when the FAB button is tapped, and several matching actions are active. With the above example, if the "accept" action is disabled, and the "select" and "detail" actions are active, a tap on the FAB button fires the "select" action.

The icon of the FAB button is defined by the IMAGE attribute of the corresponding action. If no IMAGE attribute is defined for the action, a default icon is selected from the built-in icons, based on the name of the action. See Android
**default action views decoration** on page 1788 for more details about action names for default Android™ built-in icon mapping.

**Android™ default action views decoration**

Common default action views get a decoration implicitly based on Android standards.

On Android™ devices, when the **IMAGE** and the **TEXT** action attributes are not defined for an action, the default action view gets an implicit decoration.

The default icon is selected from the name of the action. The symbol is chosen from the built-in images (Android™ material design icons), if it has the same name as the action. If no icon corresponds, the default action view will get no icon.

The text defaults to the name of the action, converted to uppercase. The text displays only if the Android™ system considers that the screen is large enough to display the texts. Typically, text is shown on tablets, but not on smartphones with small/medium screens.

For example, when implementing a **ON ACTION refresh** handler, GMA will implicitly use the default icon with the name "refresh" (the typical circular refresh symbol), and, if there is enough room, display the text "REFRESH" on the right of the icon.

Position and rendering of default action views can be controlled with Android™ specific style attributes. For more details, see Action/MenuAction style attributes on page 1192.

Note also that some actions can be rendering as the Floating Action Button of material design, as described in Android floating action button on page 1787.

Default actions views displayed in the top control bar and in the overflow button will get text but no icons, while the FAB material design button will get an icon but no text.

The table shows the default icons selected for common Genero BDL action names.

**Note:** This table does not list all possible built-in icons. More images are available from the Android™ material design icon library, and the GMA will select the icon from the action name. For example, an action with the name "audio" will get the Android™ music symbol icon:
Table 391: Default icons for common actions on Android

<table>
<thead>
<tr>
<th>Action name</th>
<th>Icon</th>
</tr>
</thead>
</table>


<table>
<thead>
<tr>
<th>Action name</th>
<th>Icon</th>
</tr>
</thead>
</table>


<table>
<thead>
<tr>
<th>Action name</th>
<th>Icon</th>
</tr>
</thead>
<tbody>
<tr>
<td>append</td>
<td></td>
</tr>
<tr>
<td>Action name</td>
<td>Icon</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Action name</td>
<td>Icon</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Action name</td>
<td>Icon</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Action name</th>
<th>Icon</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Action name</td>
<td>Icon</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>cancel</td>
<td></td>
</tr>
<tr>
<td>Action name</td>
<td>Icon</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>cut</td>
<td></td>
</tr>
<tr>
<td>Action name</td>
<td>Icon</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>delete</td>
<td></td>
</tr>
<tr>
<td>Action name</td>
<td>Icon</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Action name</th>
<th>Icon</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Action name</td>
<td>Icon</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Action name</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Icon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action name</td>
</tr>
<tr>
<td>-------------</td>
</tr>
<tr>
<td>editcut</td>
</tr>
<tr>
<td>Action name</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>1805</td>
</tr>
<tr>
<td>Action name</td>
</tr>
<tr>
<td>-------------</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Action name</th>
<th>Icon</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Action name</th>
<th>Icon</th>
</tr>
</thead>
</table>

The table above illustrates the mapping between action names and their corresponding icons. The icons are used to visually represent the actions, enhancing the user interface experience by providing quick visual cues for users.
<table>
<thead>
<tr>
<th>Action name</th>
<th>Icon</th>
</tr>
</thead>
</table>


<table>
<thead>
<tr>
<th>Action name</th>
<th>Icon</th>
</tr>
</thead>
<tbody>
<tr>
<td>findnext</td>
<td></td>
</tr>
<tr>
<td>Action name</td>
<td>Icon</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Action name</th>
<th>Icon</th>
</tr>
</thead>
</table>


<table>
<thead>
<tr>
<th>Action name</th>
<th>Icon</th>
</tr>
</thead>
<tbody>
<tr>
<td>back</td>
<td></td>
</tr>
<tr>
<td>forward</td>
<td></td>
</tr>
<tr>
<td>stop</td>
<td></td>
</tr>
<tr>
<td>Action name</td>
<td>Icon</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Action name</th>
<th>Icon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action name</td>
<td>Icon</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Action name</th>
<th>Icon</th>
</tr>
</thead>
</table>


<table>
<thead>
<tr>
<th>Action name</th>
<th>Icon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action name</td>
<td>Icon</td>
</tr>
<tr>
<td>------------</td>
<td>------</td>
</tr>
<tr>
<td>lastrow</td>
<td></td>
</tr>
<tr>
<td>Action name</td>
<td>Icon</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Action name</td>
<td>Icon</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>nextrow</td>
<td></td>
</tr>
<tr>
<td>Action name</td>
<td>Icon</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Action name</th>
<th>Icon</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Action name</td>
<td>Icon</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Action name</td>
<td>Icon</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Action name</th>
<th>Icon</th>
</tr>
</thead>
</table>


<table>
<thead>
<tr>
<th>Action name</th>
<th>Icon</th>
</tr>
</thead>
<tbody>
<tr>
<td>rewind</td>
<td></td>
</tr>
<tr>
<td>Action name</td>
<td>Icon</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
List dialogs

Describes how to program dialogs controlling list containers.

Understanding list dialogs

List dialogs are dialogs controlling a list of records rendered in a list container such as TABLE, TREE or SCROLLGRID.

Genero provides the DISPLAY ARRAY and INPUT ARRAY list dialogs to control a list of records that are defined in a program array.

Note: The DISPLAY ARRAY and INPUT ARRAY dialogs can use dynamic or static arrays. Static arrays are supported for backward compatibility. Consider using dynamic arrays in new implementations.

A DISPLAY ARRAY handles by default a read-only list. However, you can implement modification triggers, to let the end user append, modify and delete rows.

An INPUT ARRAY dialog allows immediate data modification: The rows are editable by default.

The topics in this chapter are common to all list dialogs.

The form must define a list container to display the records, it can be one of the following:

- TABLE
- TREE
- SCROLLGRID

Eventually, the list dialog can control a plain grid with basic form fields. In this case, only one record will be displayed at a time, but list navigation is still available.

Related concepts

Record list (DISPLAY ARRAY) on page 1501

The DISPLAY ARRAY instruction provides record list navigation in an application form, with optional record modification actions.

Editable record list (INPUT ARRAY) on page 1531

The INPUT ARRAY instruction provides always-editable record list handling in an application form.

Array binding in list controllers

Program array elements are bound to screen arrays elements in the definition of the DISPLAY ARRAY or INPUT ARRAY list dialog.

Array elements are bound to screen array fields by position

A screen array groups a set of fields to define list container columns in a .per form file.

When using a program array in DISPLAY ARRAY or INPUT ARRAY, the elements of the array are bound by position to the fields of the associated screen array.

In the form file:

```plaintext
... 
INSTRUCTIONS
SCREEN RECORD sa ( 
    FORMONLY.col_pkey, 
    FORMONLY.col_name, 
    FORMONLY.col_details 
); 
END
```

In the program file (note the name of the record elements can be different from screen array element names):

```plaintext
DEFINE arr DYNAMIC ARRAY OF RECORD 
pkey INTEGER,
```

...
```sql
name VARCHAR(50),
details VARCHAR(50)
END RECORD
...
DISPLAY ARRAY arr TO sa.*
...
END DISPLAY
```

**Tip:** The order of the screen array elements must match the order of program array elements used by the list dialog. However, in the list container of the LAYOUT section, the order of the columns (field item tags) does not need to match the order of the fields in the screen array. To get the tabbing order defined by the form, you will however have to use `OPTIONS FIELD ORDER FORM` in the program.

### Using PHANTOM fields to get the same columns as the database table

In most cases, the record list data comes from a database table. A program array can be easily defined with the same structure as its corresponding database table, by using a schema file and a `DEFINE LIKE` instruction:

```sql
SCHEMA stores
DEFINE arr_cust DYNAMIC ARRAY OF RECORD LIKE customer.*
```

However, you might not want to display all columns of the database table in the list container. To hide some columns, use `PHANTOM field definition` in the form. The screen array will hold all columns of the table and program array. It is then possible to bind the program array to the screen array: The number of elements in the program array and in the screen array will match.

```sql
...\nATTRIBUTES
EDIT c1 = FORMONLY.col_pkey;  -- Column in TABLE container
EDIT c1 = FORMONLY.col_name;  -- Column in TABLE container
PHANTOM FORMONLY.col_details; -- Not used in LAYOUT (only in screen array)
END
...```

### Array sub-records can be bound to flat screen arrays

If additional fields are required to hold data that is not stored in the database table, it is possible to define the program array with a sub-record matching the database table structure, and define volatile data fields beside this record.

```sql
SCHEMA stores
DEFINE arr DYNAMIC ARRAY OF RECORD
checked CHAR(1),
sql_data RECORD LIKE items.*,
comment STRING
END RECORD
MAIN
...\nINPUT ARRAY arr FROM sa.* ...
...\nEND MAIN
```

Here the "checked" and "comment" members are not part of the database table, while the "sql_data" element is defined LIKE the "items" table of the "stores" schema. All members defined in this dynamic array can be bound to a flat screen array.

### Complete example using additional fields and phantom fields

In this example, the program array is defined with more elements than the corresponding database table, and the form definition file uses phantom fields, to hide some database columns.
Form file:

LAYOUT
GRID
{
   <TABLE t1>
   [c1 |c2 |c3 ]
   [c1 |c2 |c3 ]
   [c1 |c2 |c3 ]
   [c1 |c2 |c3 ]
   < }
END
END

ATTRIBUTES
EDIT c1 = FORMONLY.pkey, TITLE="PKey", NOENTRY;
EDIT c2 = FORMONLY.name, TITLE="Name";
CHECKBOX c3 = FORMONLY.checked, TITLE="Checked";
PHANTOM FORMONLY.details; -- Not used in layout
PHANTOM FORMONLY.comment; -- Not used in layout
END

INSTRUCTIONS
SCREEN RECORD sa(
   FORMONLY.checked, -- Note order is different to the layout
   FORMONLY.pkey,
   FORMONLY.name,
   FORMONLY.details,
   FORMONLY.comment
);
END

Program file:

TYPE t_type RECORD
   checked CHAR(1),
   sql_data RECORD -- Could be RECORD LIKE items.*
      pkey INTEGER,
      name VARCHAR(50),
      details VARCHAR(200)
END RECORD,
comment STRING
END RECORD

DEFINE arr DYNAMIC ARRAY OF t_type

MAIN
   OPTIONS INPUT WRAP, FIELD ORDER FORM
   CALL create_db()
   CALL fill_array()
   OPEN FORM f1 FROM "form1"
   DISPLAY FORM f1
   INPUT ARRAY arr FROM sa.* ATTRIBUTES(WITHOUT DEFAULTS)
      BEFORE ROW
         MESSAGE arr[arr_curr()].sql_data.details
   END INPUT
END MAIN

FUNCTION create_db()
   DEFINE rec t_type
   CONNECT TO ":memory:+driver='dbmsqt'"
   CREATE TABLE items {
      pkey INTEGER PRIMARY KEY,
      name VARCHAR(50),
      }
details VARCHAR(200)
)
FOR rec.sql_data.pkey=100 TO 150
LET rec.sql_data.name = SFMT("Item %1",rec.sql_data.pkey)
LET rec.sql_data.details = SFMT("Details for %1",rec.sql_data.pkey)
INSERT INTO items VALUES ( rec.sql_data.* )
END FOR
END FUNCTION

FUNCTION fill_array()
DEFINE rec t_type,
x INTEGER
DECLARE c1 CURSOR FOR
SELECT 'N', items.*, '' FROM items ORDER BY pkey
CALL arr.clear()
FOREACH c1 INTO rec.*
LET x = x+1
LET arr[x].* = rec.*
END FOREACH
FREE c1
END FUNCTION

Related concepts
Binding tables to arrays in dialogs on page 1857
Program arrays act as data model that are bound to form tables, when implementing list dialogs.
Variable binding in DISPLAY ARRAY on page 1504
Variable binding in INPUT ARRAY on page 1535
Populating a DISPLAY ARRAY on page 1841
The program array must be filled with rows to populate the DISPLAY ARRAY dialog.

Controlling the number of rows
Methods are provided to set and get the total number of rows in a read-only or editable list of records.

Set the number of rows when using a static array
When using a static array in DISPLAY ARRAY or INPUT ARRAY, you must specify the actual number of rows with the SET_COUNT() built-in function or with the COUNT dialog attribute. Both of them are only taken into account when the interactive instruction starts.

DEFINE arr ARRAY[100] OF ...
... (fill the array with x rows)
CALL set_count(x)
DISPLAY ARRAY arr TO sa.*
...
END DISPLAY

When using multiple list subdialogs in a DIALOG block, the SET_COUNT() built-in function is unusable, as it defines the total number of rows for all lists. The only way to define the number of rows when using a static array in multiple dialogs is to use the COUNT attribute.
Consider using dynamic arrays instead of static arrays.

Set the number of rows when using a dynamic array
When using a dynamic array in DISPLAY ARRAY or INPUT ARRAY, the total number of rows is automatically defined by the array variable (array.getLength()).

DEFINE arr DYNAMIC ARRAY OF ...
... (fill the array with x rows)
DISPLAY ARRAY arr TO sa.*
However, special consideration has to be taken when using the paged mode of DISPLAY ARRAY. In this mode, the dynamic array only holds a page of the complete row set shown to the user. In paged mode, you must specify the total number of rows with the `ui.Dialog.setArrayLength()` method.

**Get the number of rows in a list**

To get the current number of rows in a DISPLAY ARRAY or INPUT ARRAY, use either the `ui.Dialog.getArrayLength()` or the `ARR_COUNT()` function.

The `getArrayLength()` method can be used inside or outside the context of the list dialog, as it takes the screen array as parameter to identify the list dialog. For example, when implementing a DIALOG block with two DISPLAY ARRAY subdialogs, you can query the number of rows of a list in the code block of another list controller:

```plaintext
DIALOG ...
    DISPLAY ARRAY arr1 TO sa1.*
    ON ACTION check
        IF DIALOG.getArrayLength("sa2") > 1 THEN
            ...
        END IF
    END DISPLAY
    DISPLAY ARRAY arr2 TO sa2.*
    END DISPLAY
END DIALOG
```

The `ARR_COUNT()` function must be used in the context of the DISPLAY ARRAY or INPUT ARRAY dialog, or just after executing such dialog. For example, it can be used just after an INPUT ARRAY dialog, to get the number of rows left in the list:

```plaintext
INPUT ARRAY arr FROM sa.*
...
END INPUT
IF NOT int_flag THEN
    FOR i=1 TO arr_count()
        ...
    END FOR
END IF
```

The `ARR_COUNT()` function returns the number of rows for the last executed dialog, until a new list dialog is started.

**Related concepts**

**ARRAY** on page 419
An array defines a vector variable with a list of elements.

**Populating a DISPLAY ARRAY** on page 1841
The program array must be filled with rows to populate the DISPLAY ARRAY dialog.

**Handling the current row**
Query and control the current row in a read-only or editable list of records.

**Get the current row**
To query the current row of a list, use either the `ui.Dialog.getCurrentRow()` method or the `arr_curr()` built-in function, depending on the context.

The `getCurrentRow()` method can be used for a DISPLAY ARRAY or INPUT ARRAY dialog. The method takes the name of the screen array as the argument to identify the list.
For example, when implementing a DIALOG block with two DISPLAY ARRAY subdialogs, you can query the current row of a list in the code block of the other list controller:

```plaintext
DIALOG ...
    DISPLAY ARRAY arr1 TO sa1.*
    ON ACTION check
        IF arr2[DIALOG.getCurrentRow("sa2").value > 0 THEN
            ...
        END IF
    END DISPLAY
    DISPLAY ARRAY arr2 TO sa2.*
END DISPLAY
END DIALOG
```

The `arr_curr()` function must be used in the context of the current DISPLAY ARRAY or INPUT ARRAY dialog, or just after executing such a dialog.

For example, when implementing modification triggers in a DISPLAY ARRAY dialog, the current row and the current screen line can be queried respectively with the `arr_curr()` and `scr_line()` functions:

```plaintext
DISPLAY ARRAY arr TO sa.*
ON UPDATE
    INPUT arr[arr_curr()].* WITHOUT DEFAULTS FROM sa[scr_line()].* ;
END DISPLAY
```

After the dialog execution, `arr_curr()` returns the current row index for the last executed dialog, until a new list dialog is started.

The row index returned by the `arr_curr()` function is constant in the context of a dialog block, even when removing rows from the array by program. A typical mistake is to reuse the `arr_curr()` index to get data from the new current row, after deleting the last row of the array.

In the next code example, the reuse of `arr_curr()` without checking for the new number of rows will automatically create a new program array element when accessing the element in the MESSAGE instruction:

```plaintext
ON ACTION dialog_delete_row
    CALL DIALOG.deleteRow("sr",arr_curr())
    MESSAGE "Current item:", arr[arr_curr()].name
```

The above code works until you reach the last row: When last row is deleted, the MESSAGE instruction is automatically creating a new array element at the same index returned by `arr_curr()`.

Unlike `arr_curr()`, the DIALOG.getCurrentRow() method is synchronized with the actual number of rows in the array, as long as methods like DIALOG.deleteRow() are used:

```plaintext
ON ACTION dialog_delete_row
    CALL DIALOG.deleteRow("sr",DIALOG.getCurrentRow("sr"))
    MESSAGE "Current item:", arr[DIALOG.getCurrentRow("sr").name
```

However, the code must also test if there are still rows in the list after deleting a row:

```plaintext
ON ACTION dialog_delete_row
    LET x = DIALOG.getCurrentRow("sr")
    CALL DIALOG.deleteRow("sr",x)
    LET x = DIALOG.getCurrentRow("sr")
    IF x > 0 THEN
        MESSAGE "Current item:", arr[x].name
    ELSE
        MESSAGE "No more rows in the list"
    END IF
```
Set the current row

To set the current row in a list controlled by a DISPLAY ARRAY or INPUT ARRAY, use the `ui.Dialog.setCurrentRow()` method. This method takes the name of the screen array and the new row index as parameters:

```plaintext
DISPLAY ARRAY p_items TO sa.*
    ... ON ACTION next_empty
        LET row = findEmptyRow(p_items)
        CALL DIALOG.setCurrentRow("sa", row)
    ...
END DISPLAY
```

Calling the `DIALOG.setCurrentRow()` method will not execute control blocks such as BEFORE ROW and AFTER ROW, and will not set the focus. If you want to set the focus to the list, you must use the NEXT FIELD instruction. This works with DISPLAY ARRAY as well as INPUT ARRAY.

**Tip:** Use this method with care. Let the dialog handle normal navigation automatically, and jump to a specific row only in the context of an ON ACTION block.

The `fgl_set_arr_curr()` function can also be used. This function must be called in the context of the current list having the focus.

**Note:** `fgl_set_arr_curr()` triggers control blocks such as BEFORE ROW, while `DIALOG.setCurrentRow()` does not trigger any control blocks.

In a DISPLAY ARRAY using paged mode with COUNT=-1, before calling `DIALOG.setCurrentRow(screen-array, row-index)`, call `DIALOG.setArrayLength(screen-array, count)` where `count >= row-index`. Otherwise, the `setCurrentRow()` call will have no effect, if the dialog has not yet seen `row-index` rows through ON FILL BUFFER.

Converting visual index to/from program array index

When the end user sorts rows in a table, the program array index (`arr_curr()`) may differ from the visual row index (the row position as seen by the user).

The `ui.Dialog` class provides methods to convert between these contexts:

The `ui.Dialog.arrayToVisualIndex` on page 2527 method converts a program array index to a visual index. It can be used, for example, to display a typical list position message (Row: `current-row / total-rows`). The current row (`arr_curr()`) is a program array index that must be converted to a visual index. Note that you need to display such messages in the BEFORE ROW trigger and ON SORT trigger:

```plaintext
FUNCTION disp_row(d,n)
    DEFINE d ui.DIALOG, n STRING
    MESSAGE SFMT("Row: %1/%2",
        d.arrayToVisualIndex(n,d.getCurrentRow(n)),
        d.getArrayLength(n))
END FUNCTION
...
    DISPLAY ARRAY arr TO sr.*
    ...
    BEFORE ROW
        CALL disp_row(DIALOG,"sr")
    ON SORT
        CALL disp_row(DIALOG,"sr")
    ...
END DISPLAY
```
The `ui.Dialog.visualToArrayIndex` on page 2553 method converts a visual index to a program array index. It can be used for example to ask the user for a row position (visual index), and make that row current by using `DIALOG.setCurrentRow()` after converting to the program array index:

```plaintext
DEFINE i INTEGER
...
DISPLAY ARRAY arr TO sr.*
...
ON ACTION move_to
  PROMPT "Enter row index:" FOR i
  CALL DIALOG.setCurrentRow("sr", DIALOG.visualToArrayIndex("sr", i))
...
END DISPLAY
```

Related concepts

- **Record list (DISPLAY ARRAY)** on page 1501
  The `DISPLAY ARRAY` instruction provides record list navigation in an application form, with optional record modification actions.

- **Editable record list (INPUT ARRAY)** on page 1531
  The `INPUT ARRAY` instruction provides always-editable record list handling in an application form.

- **The DISPLAY ARRAY sub-dialog** on page 1597
  The `DISPLAY ARRAY` sub-dialog is the controller to implement the navigation in a list of records, with option data modification actions.

- **The INPUT ARRAY sub-dialog** on page 1598
  The `INPUT ARRAY` sub-dialog is the controller to implement the navigation and edition in a list of records.

- **List ordering** on page 1862
  List controllers implement a built-in sort. This feature can be disabled if not required.

**Defining the action for a row choice**

The row choice in the `DISPLAY ARRAY/INPUT ARRAY` dialog can be associated with a dedicated action.

**Action fired by default by current row selection**

When using a `DISPLAY ARRAY` dialog to control a list view like a `TABLE`, the physical event to choose a row has the following results, depending on the type of front-end:

- On the desktop (GDC) or web (GBC) front-end, by default, a mouse double-click changes the current row, and fires the "accept" action, if available. If the default accept action is fired, the dialog will end, except if the accept action has been disabled or was overwritten by an `ON ACTION accept` handler. This default behavior applies to most record lists of desktop applications, where the main purpose is to let the user choose a row from the list.
- On a mobile device (GMA/GMI), there is no concept of double-click as with a mouse. By default, a tap on a row changes the current row only. This corresponds to a single mouse click on a desktop front-end, and therefore does not fire the "accept" action by default. If a tap must fire an action, define that action with the `DOUBLECLICK` attribute in the `DISPLAY ARRAY` dialog.

When using an `INPUT ARRAY` dialog, no row choice action ("accept") is possible by default. However, when setting the `DOUBLECLICK` attribute, a double click can be detected during `INPUT ARRAY`.

**Defining the name of the row choice action**

The action to be fired when a row choice occurs can be defined in the form file with the `DOUBLECLICK` attribute of the `TABLE`, `TREE` or `SCROLLGRID` containers, or with the `DOUBLECLICK` attribute of the `DISPLAY ARRAY` dialog:

```plaintext
DISPLAY ARRAY arr TO sr.*
  ATTRIBUTES(UNBUFFERED, DOUBLECLICK=select)
  ...
```

```plaintext
```
The DISPLAY ARRAY attributes DOUBLECLICK, DETAILACTION and ACCESSORYTYPE were introduced for mobile apps, to be used together at dialog configuration level for list configuration.

**Note:** The DOUBLECLICK attribute in DISPLAY ARRAY takes precedence over the DOUBLECLICK attribute in the list container of a form file.

When defining a DOUBLECLICK action in the list container of the form file, or in a dialog attribute, you declare an explicit action view, and no default action view will be displayed for this action (you can force it with DEFAULTVIEW=YES).

**Using the DOUBLECLICK attribute with INPUT ARRAY**

With an INPUT ARRAY, field editing is implicit: A double click with the mouse is typically used to select text in an editable field. Therefore, unlike DISPLAY ARRAY, no accept action is fired by default when the user double-clicks on a cell during an INPUT ARRAY.

However, if the TABLE, TREE or SCROLLGRID list container defines the DOUBLECLICK attribute, the corresponding action is fired when a double click occurs, and a row choice action can be implemented in and INPUT ARRAY. This configuration is typically used when the list container has columns defined with non-editable fields like LABEL or EDIT using NOENTRY.

**Physical event triggering the row choice action**

For desktop and web UI applications, the physical event that fires the row choice action is the mouse double-click, which can be changed with the rowActionTrigger presentation style attribute for the Table, ScrollGrid and Tree classes.

For example, in web applications, the row choice is typically done by a simple mouse click. Define the simple-click physical event to fire the row choice action in your .4st style file as follows:

```
...  
   <Style name="Table">  
     <StyleAttribute name="rowActionTrigger" value="singleClick" />  
   </Style>
...  
```

**Row choice action handler in programs**

To handle row choice actions in the program code, define the DOUBLECLICK attribute for the DISPLAY ARRAY dialog, and the corresponding action handler block ON ACTION action-name:

```
DISPLAY ARRAY arr TO sr.*  
   ATTRIBUTES (UNBUFFERED, DOUBLECLICK=select)  
   ON ACTION select  
     MESSAGE "myselect:", arr_curr()  
END DISPLAY
```

**Note:** If the DOUBLECLICK attribute is defined, it will only configure the action for the corresponding physical event. By default, the "accept" action is still available, and the [Ok] button or the [Return] key will still fire the accept action and leave the dialog. To avoid the default accept action, add ACCEPT=FALSE to the DISPLAY ARRAY attribute list.

**Execution order of row change control blocks**

During a DISPLAY ARRAY, if the selected row is not the current row, the AFTER ROW and BEFORE ROW control blocks execute before the ON ACTION block, in the following order:

1. AFTER ROW (for the previous current row)
2. BEFORE ROW (for the new current row)
3. ON ACTION double-click-action
Related concepts

**DISPLAY ARRAY instruction configuration** on page 1505

**Field-level focus in DISPLAY ARRAY**
The DISPLAY ARRAY dialog supports cell-level focus with the FOCUSONFIELD.

**Enabling focusable cells in DISPLAY ARRAY**
When using a DISPLAY ARRAY dialog to control a list view, you can enable cell-level focus handling with the FOCUSONFIELD attribute.

**Important:** This feature is not supported on mobile platforms.

With a graphical front-end, this feature allows cell mouse clicks or tabbing between cells, instead of having the whole current row highlighted.

This feature can also be used in text mode (FGLGUI=0), to move in DISPLAY ARRAY fields with the keyboard using arrow keys or tab.

To enable field-level focus handling in a DISPLAY ARRAY, add the FOCUSONFIELD attribute in the dialog definition:

```plaintext
DISPLAY ARRAY arr TO sr.* ATTRIBUTES(FOCUSONFIELD)
```

**Detecting cell focus changes**
When the FOCUSONFIELD attribute is defined, BEFORE FIELD and AFTER FIELD blocks can be used, to detect field focus changes:

```plaintext
DISPLAY ARRAY arr TO sr.* ATTRIBUTES(FOCUSONFIELD)

  BEFORE FIELD cust_id
    MESSAGE "focus in cust_id, row = ", arr_curr()
  AFTER FIELD cust_id
    MESSAGE "focus left cust_id, row = ", arr_curr()
  BEFORE FIELD cust_name
    MESSAGE "focus in cust_name, row = ", arr_curr()

... END DISPLAY
```

Note that if defined, the AFTER ROW control block will execute after the AFTER FIELD block and the BEFORE ROW control block will execute before the BEFORE FIELD block. The code blocks execute in the following order:

1. AFTER FIELD (for the field that loses the focus)
2. AFTER ROW (for the previous current row)
3. BEFORE ROW (for the new current row)
4. BEFORE FIELD (for the field that gets the focus in the new row)

**What is the current cell?**
The current cell of DISPLAY ARRAY with FOCUSONFIELD attribute can be found by using the `ui.Dialog.getCurrentItem()` method:

```plaintext
DISPLAY ARRAY arr TO sr.* ATTRIBUTES(FOCUSONFIELD)

  ... ON ACTION show_current_cell
    MESSAGE "Current cell = ", DIALOG.getCurrentItem()
```
Setting the current cell

To set the focus to a specific cell with program code, use the NEXT FIELD instruction, or the ui.Dialog.nextField() method, in conjunction with the ui.Dialog.setCurrentRow() method:

```plaintext
DISPLAY ARRAY arr TO sr.* ATTRIBUTES(FOCUSONFIELD)
  
  ON ACTION top_left ATTRIBUTES(TEXT = "TOP LEFT")
    CALL DIALOG.setCurrentRow("sr", 1)
    NEXT FIELD first_field
  
  ON ACTION bottom_right ATTRIBUTES(TEXT = "BOTTOM RIGHT")
    CALL DIALOG.setCurrentRow("sr", sr.getLength())
    NEXT FIELD last_field
  
END DISPLAY
```

Related concepts
DISPLAY ARRAY control blocks on page 1508

Actions bound to the current row

Actions can be configured with the ROWBOUND attribute depending on whether there is a current row.

When using a DISPLAY ARRAY or INPUT ARRAY dialog to control a table view, actions can get the ROWBOUND attribute in order to make the action only available when there is a current row in the list.

Important: This feature is only for mobile platforms.

The ROWBOUND attribute must only be used with TABLE and TREE containers (it does not make sense for SCROLLGRID and static lists in GRID containers).

This attribute is generally used in mobile applications, when a list view requires actions to be decorated in a row-specific way. For example, on Android™ devices, the actions with the ROWBOUND attribute will be available by selecting the three-dot button on the right of each list view cell.

In the following example, the DISPLAY ARRAY dialog implements three actions:

- The "refresh" action is not "rowbound", and will always be available (active/visible), even if the list is empty.
- The "check" action is rowbound, and will only be available if there is a (current) row in the list.
- The "delete" action created by the ON DELETE modification trigger is implicitly "rowbounded".

```plaintext
DISPLAY ARRAY a_orders TO sr.* ATTRIBUTES(UNBUFFERED)
  
  ON ACTION refresh -- not rowbound
    CALL fetch_orders()
  
  ON ACTION check ATTRIBUTES(ROWBOUND)
    CALL check_order(arr_curr())
  
  ON DELETE -- implicitly rowbound
    CALL delete_order(arr_curr())
  
END DISPLAY
```

Related concepts
Default action views on page 1743
A default action view is created to render an action handler when no explicit action view exists for it.

Using tables on mobile devices on page 1867
Table views render in a specific way on mobile devices, in order to take advantage of mobile device ergonomics.

Populating a DISPLAY ARRAY

The program array must be filled with rows to populate the DISPLAY ARRAY dialog.

With DISPLAY ARRAY, either full list mode or paged mode is used to fill the form array. Consider using full list mode for short/medium result sets, and use paged mode for very large result sets.
Full list mode of DISPLAY ARRAY

In order to handle short/medium result sets, use the full list mode of DISPLAY ARRAY.

Understanding the full list mode

In full list mode, DISPLAY ARRAY uses a complete copy of the result set to be displayed in the form array. The full list mode is typically used for a short or medium row set (10 - 100 rows).

In full list mode, the DISPLAY ARRAY instruction uses a static or dynamic program array defined with a record structure corresponding to (or to a part of) a screen-array in the current form.

The program array is filled with data rows before DISPLAY ARRAY is executed, typically with a FOREACH loop when rows come from the database.

Figure 92: Full list mode in DISPLAY ARRAY diagram

Consider using a dynamic array instead of a static array: By using a dynamic array the program will only use the required memory resources, and the dialog will automatically detect the number of rows from the dynamic array (array.getLength())

Full list mode example

The following example implements a DISPLAY ARRAY in its simpler form: a dynamic array is filled with database rows and contains the whole result set to be displayed in the table:

```gdl
MAIN
    DEFINE arr DYNAMIC ARRAY OF RECORD
        id INTEGER,
        fname CHAR(30),
        lname CHAR(30)
    END RECORD
    DEFINE i INTEGER

DATABASE stores

OPEN FORM f1 FROM "custlist"
DISPLAY FORM f1

DECLARE c1 CURSOR FOR
    SELECT customer_num, fname, lname FROM customer
LET i=1
```
FOREACH cl INTO arr[i].*
   LET i = i+1
END FOREACH
CALL arr.deleteElement(i)

DISPLAY ARRAY arr TO sa.* ATTRIBUTES(UNBUFFERED)
   BEFORE ROW
      MESSAGE "Moved to row ", arr.curr()
END DISPLAY

END MAIN

Related concepts

**ARRAY** on page 419
An array defines a vector variable with a list of elements.

**Dynamic arrays** on page 422
**Multiple row selection** on page 1851
Multiple row selection allows the end user to select several rows within a list of records.

**Tree views** on page 1889
Describes how to implement tree views.

**The buffered and unbuffered modes** on page 1720
The buffered and unbuffered mode control the synchronization of program variables and form fields.

**Paged mode of DISPLAY ARRAY** on page 1843
In order to handle very large result sets, use the paged mode of DISPLAY ARRAY.

**Understanding the paged mode**

The *paged mode* of DISPLAY ARRAY allows the program to display a very large number of rows, without copying all database rows into the program array.

This mode uses the **ON FILL BUFFER** data block to let the program populate the array with the current visible page of rows. This is a subset of the database query result set (SELECT), typically controlled by a scrollable cursor.

![Figure 93: Paged mode diagram](image-url)
**Note:** DISPLAY ARRAY has following constraints when using the paged mode:

- By default, **row sorting** is not allowed. Implement an **ON SORT** trigger to handle list sorting.
- **Multi-range selection** is not supported, if the paged mode uses an undefined number of rows (COUNT=-1).
- To fill a **tree view** dynamically, use the **ON EXPAND / ON COLLAPSE** data blocks.

**Paged mode programming details**

In paged mode, the dynamic array holds a page of rows, not all rows of the result set. The data rows are provided through the **ON FILL BUFFER** block, by filling a dynamic array with the rows for the current visible page.

The **ON FILL BUFFER** clause is used to fill a page of rows in the dynamic array, from a row offset and the number of rows required in the page. The row offset is defined by the `FGL_DIALOG_GETBUFFERSTART()` built-in function, and the number of rows to provide is defined by the `FGL_DIALOG_GETBUFFERLENGTH()` built-in function.

**Note:** The **ON FILL BUFFER** is only triggered when all the user code is executed and the dialog gets the control back. This means that the fill clause is not immediately fired when calling `DIALOG.setArrayLength()`.

An initial **ON FILL BUFFER** block must be executed before the user gets the control:

- In a single DISPLAY ARRAY, the **ON FILL BUFFER** will be triggered before the **BEFORE DISPLAY** block.
- In a DISPLAY ARRAY of a DIALOG/END DIALOG instruction, **ON FILL BUFFER** will be triggered before the **BEFORE DIALOG** block.

If the total number of rows is known before the dialog is started, specify it with the **COUNT** dialog attribute of DISPLAY ARRAY:

- The total number of rows can be changed during dialog execution with the `ui.Dialog.setArrayLength()` method.
- In singular DISPLAY ARRAY instructions, define the total number of rows of a paged mode with the **SET_COUNT()** built-in function or the **COUNT** attribute. Note that **SET_COUNT()** or **COUNT** are only taken into account when the dialog starts.
- If the total number of rows changes during the execution of the dialog, the only way to specify the number of rows is `DIALOG.setArrayLength()`.

If the total number of rows is not known before starting the DISPLAY ARRAY dialog, set **COUNT=-1** in DISPLAY ARRAY attributes. The dialog will then query for rows with **ON FILL BUFFER** until the end of the result set is reached. The end of the result set is detected:

- When the number of rows provided in **ON FILL BUFFER** is less than the number of rows required by the dialog.
- When you reset the total number of rows to a value higher than -1 with the `ui.Dialog.setArrayLength()` method.

**Note:** The dialog cannot support **multi-row selection** when the total number of rows is undefined.

The `DIALOG.setCurrentRow()` method can be used to move to a specific row in a paged mode DISPLAY ARRAY.

**Note:** Before calling `DIALOG.setCurrentRow(screen-array, row-index)`, be sure to provide the actual number of rows with `DIALOG.setArrayLength(screen-array, count)` where `count >= row-index`. Otherwise, the `setCurrentRow()` call will have no effect, if the dialog has not yet seen `row-index` rows through **ON FILL BUFFER**.

If you use a tree view with a paged mode DISPLAY ARRAY, the program will raise an error at runtime: For tree views, the dialog needs the complete set of open nodes with parent/child relations to handle the tree view display. With the paged mode, only a short window of the dataset is known by the dialog.

A typical paged DISPLAY ARRAY implementation consists of a scroll cursor providing the list of records to be displayed. Scroll cursors use a static result set. If you want to display fresh data, you can implement an advanced paged mode by using a scroll cursor that provides the primary keys of the referenced result set, plus a prepared cursor to fetch rows on demand in the **ON FILL BUFFER** clause. In this case you may need to check whether a row still exists when fetching a record with the second cursor.
Paged mode basic example

The following example shows a DISPLAY ARRAY implementation using a scroll cursor to fill pages of records in ON FILL BUFFER, specifying an undefined number of rows (COUNT=-1).

```plaintext
MAIN
  DEFINE arr DYNAMIC ARRAY OF RECORD
  id INTEGER,
    fname CHAR(30),
    lname CHAR(30)
  END RECORD
DEFINE cnt, ofs, len, row, i INTEGER
DATABASE stores
OPEN FORM f1 FROM "custlist"
DISPLAY FORM f1
DECLARE c1 SCROLL CURSOR FOR
  SELECT customer_num, fname, lname FROM customer
OPEN c1
DISPLAY ARRAY arr TO sa.* ATTRIBUTES(COUNT=-1)
ON FILL BUFFER
  CALL arr.clear()
  LET ofs = fgl_dialog_getBufferStart()
  LET len = fgl_dialog_getBufferLength()
  LET row = ofs
  FOR i=1 TO len
    FETCH ABSOLUTE row c1 INTO arr[i].*
    IF SQLCA.SQLCODE!=0 THEN
      CALL DIALOG.setArrayLength("sa",row-1)
      EXIT FOR
    END IF
    LET row = row + 1
  END FOR
ON ACTION ten_first_rows_only
  CALL DIALOG.setArrayLength("sa", 10)
END DISPLAY
END MAIN
```

Paged mode with sorting feature

To implement row sorting in a DISPLAY ARRAY using paged mode, use the ON SORT trigger to detect a sort request, get the sort information with the ui.Dialog.getSortKey on page 2534 / ui.Dialog.isSortReverse on page 2537 methods, and re-execute the SQL query to sort rows accordingly with an ORDER BY clause. The ON SORT trigger will be fired before the ON FILL BUFFER trigger:

```plaintext
MAIN
  DATABASE test1
  OPEN FORM f1 FROM "custlist"
  DISPLAY FORM f1
  CALL show_list()
END MAIN

FUNCTION execute_sql(order_by)
  DEFINE order_by STRING
  DEFINE sql STRING
  IF order_by IS NULL THEN
    LET order_by = "ORDER BY fname"
  END IF
```
LET sql = "SELECT customer_num, fname, lname FROM customer ", order_by
DECLARE c1 SCROLL CURSOR FROM sql
OPEN c1
END FUNCTION

FUNCTION show_list()
DEFINE arr DYNAMIC ARRAY OF RECORD
  id INTEGER,
  fname VARCHAR(30),
  lname VARCHAR(30)
END RECORD
DEFINE ofs, len, row, i INTEGER,
  key STRING, rev BOOLEAN
CALL execute_sql(NULL)
DISPLAY ARRAY arr TO sa.* ATTRIBUTES(COUNT=-1)
ON SORT
  LET key = DIALOG.getSortKey("sa")
  LET rev = DIALOG.isSortReverse("sa")
  IF key IS NULL THEN
    CALL execute_sql( NULL )
  ELSE
    -- Assuming that form field names match table column names
    CALL execute_sql( "ORDER BY " || key || IIF(rev," DESC"," ") )
  END IF
ON FILL BUFFER
  CALL arr.clear()
  LET ofs = fgl_dialog_getBufferStart()
  LET len = fgl_dialog_getBufferLength()
  LET row = ofs
  FOR i=1 TO len
    FETCH ABSOLUTE row c1 INTO arr[i].*
    IF SQLCA.SQLCODE!=0 THEN
      CALL DIALOG.setArrayLength("sa",row-1)
      EXIT FOR
    END IF
    LET row = row + 1
  END FOR
END DISPLAY
END FUNCTION

Note that with the above example, the current row remains at the same position: When the table is sorted, the set of rows provided in the ON FILL BUFFER may not include the database row that was the current row before the sort.

To track the current row, store the primary key value of the current row before re-executing the query. After query execution, scan the cursor result set and perform a DIALOG.setCurrentRow() when the primary key of the current row is found. The current row might be outside the row set provided in ON FILL BUFFER. In order to make setCurrentRow() work properly, you have to count the total number of rows before the DISPLAY ARRAY:

... DEFINE cnt, ofs, len, row, i INTEGER,
  key STRING, rev BOOLEAN,
  row_count, curr_id, last_id INTEGER
...

SELECT COUNT(*) INTO row_count FROM customer

CALL execute_sql(NULL)
DISPLAY ARRAY arr TO sa.* ATTRIBUTES(COUNT=row_count)
ON SORT
  LET row = DIALOG.getCurrentRow("sa")
FETCH ABSOLUTE row c1 INTO last_id
LET key = DIALOG.getSortKey("sa")
LET rev = DIALOG.isSortReverse("sa")
IF key IS NULL THEN
   CALL execute_sql( NULL )
ELSE
   -- Assuming that form field names match table column names
   CALL execute_sql( "ORDER BY " || key || IIF(rev," DESC"," " ) )
END IF
LET row=1
WHILE TRUE
   FETCH c1 INTO curr_id
   IF SQLCA.SQLCODE==100 THEN
      ERROR "Last current row disappeared from result set!"
      EXIT PROGRAM 1
   END IF
   IF curr_id == last_id THEN
      CALL DIALOG.setCurrentRow("sa",row)
      EXIT WHILE
   END IF
   LET row = row+1
END WHILE

Related concepts

ARRAY on page 419
An array defines a vector variable with a list of elements.

Dynamic arrays on page 422

Full list mode of DISPLAY ARRAY on page 1842
In order to handle short/medium result sets, use the full list mode of DISPLAY ARRAY.

Scrollable cursors on page 623
How scrollable cursors can be supported on different databases.

INPUT ARRAY row modifications

Controlling row creation and deletion in an editable record list.

The INPUT ARRAY instruction handles record list editing. This controller allows the user to directly edit existing rows and to create or remove rows with implicit actions.

The following implicit actions are created by default by the INPUT ARRAY dialog:

- `insert`: creates a new row before the current row. If there are no rows in the list, the action adds a new one.
- `append`: creates a new row after the last row of the list.
- `delete`: deletes the current row.

To prevent INPUT ARRAY from creating the implicit "insert", "append" and "delete" actions, set respectively the `INSERT ROW`, `APPEND ROW`, or `DELETE ROW` control attributes to FALSE. To prevent row addition, also set the `AUTO APPEND` attribute to FALSE.

```sql
... 
INPUT ARRAY p_items FROM sa.*
   -- Allow only row append and delete implicit actions.
   ATTRIBUTES(AUTO APPEND=FALSE,
               INSERT ROW=FALSE)
... 
END INPUT
...```

Specific control blocks are available to take control when a row is created or deleted:
• BEFORE INSERT and AFTER INSERT control blocks can be used to control row creation. Cancel a row creation with CANCEL INSERT in BEFORE INSERT or AFTER INSERT blocks.
• BEFORE DELETE and AFTER DELETE control blocks can be used to control row deletion. Cancel row deletion with the CANCEL DELETE instruction in BEFORE DELETE.

Dynamic arrays and the `ui.Dialog` class provide methods such as `array.deleteElement()` or `ui.Dialog.appendRow()` to modify the list. When using these methods, the predefined triggers such as BEFORE DELETE or BEFORE INSERT are not executed. While it is safe to use these methods within a DISPLAY ARRAY, you must take care when using an INPUT ARRAY. For example, it is not recommended to call such methods in triggers like BEFORE ROW, AFTER INSERT, BEFORE DELETE.

Users can append temporary rows by moving to the end of the list, or when executing the append action. Appending temporary rows is different from inserting a row; an appended row is considered temporary until the user modifies a field while an inserted row remains in the list even if the user does not modify a field.

By default, when the last row is removed by a delete action, the INPUT ARRAY instruction will automatically create a new temporary row at the same position. The visual effect of this behavior can be misinterpreted - if no data were entered in the last row, you cannot see any difference. However, the last row is actually deleted and a new row is created, and the BEFORE DELETE / AFTER DELETE / AFTER ROW / BEFORE ROW / BEFORE INSERT control block sequence is executed. In order to avoid the creation of a new temporary row when the last row is deleted, set AUTO APPEND = FALSE attribute.

The insert, append or delete actions will be automatically disabled depending on the context. If the INPUT ARRAY is using a static array that becomes full, or if the MAXCOUNT attribute is reached, both insert and append actions will be disabled. The delete action is automatically disabled when AUTO APPEND = FALSE and there are no more rows in the array.

Related concepts
The INPUT ARRAY sub-dialog on page 1598
The INPUT ARRAY sub-dialog is the controller to implement the navigation and edition in a list of records.

INPUT ARRAY ATTRIBUTES clause on page 1603
INPUT ARRAY specific attributes can be defined in the ATTRIBUTE clause of the sub-dialog header.

The DIALOG control class on page 1709
This topic explains the purpose of the `ui.DIALOG` class.

Appending rows in INPUT ARRAY
Rows appended at the end of an editable list are temporary until they are edited.

In an INPUT ARRAY, a new row can be created at the end of the list. This new row is called “temporary” because it will be automatically removed if the user leaves the row without entering data. If data is entered by the user or by program (setting the touched flag), the temporary row becomes permanent.

A temporary row is promoted to a permanent row under certain conditions described in this topic. We distinguish also explicit temporary row creation from automatic temporary row creation.

Note: Temporary row creation is different from adding new rows with the `DIALOG.appendRow()` method; when appending a row by program, the row is considered permanent and remains in the list even if the user did not enter data in fields.

Explicit temporary row creation
Explicit temporary row creation takes place when the user decides to append a new row explicitly with the append action. If the list is empty, an insert action will have the same effect as an append action (a temporary row is created at position 1).

Automatic temporary row creation
By default, automatic temporary row creation takes place when AUTO APPEND is TRUE (default) and one of the following occurs:
• The user tries to move below the last row, with a Down keystroke or with the mouse.
• The user presses the Tab key when in the last field of the last row.
• The last row of the list is deleted by the user.
• The list has the focus and the last row of the list is deleted by program with `DIALOG.deleteRow()` or `DIALOG.deleteAllRows()`.
• When the `INPUT ARRAY` is in a `DIALOG` block, the list has no rows and gets the focus (A new temporary row is created to let the user enter data immediately)

**When is a temporary row removed?**

The temporary row will be automatically removed if none of the fields has been touched (the modification flag is not set), and when leaving the list (INPUT ARRAY) or by moving upward in the list.

The row is "touched" when the user enters data in a field, or when the program simulates a user input with a `DISPLAY TO / BY NAME` instruction or with the `DIALOG.setFieldTouched()` method.

**Note:** When the modification flag is set by program, NOENTRY fields are ignored. However, fields dynamically disabled by `DIALOG.setFieldActive()` are taken into account.

**Deny row append**

Temporary row creation is useful because, in most cases, `INPUT ARRAY` is used to edit existing rows and append new rows at the end of the list. However, you might want to prevent row addition or at least avoid the automatic temporary row creation when the last row is deleted or when an empty list gets the focus.

To avoid explicit temporary row creation, prevent `INPUT ARRAY` from defining the implicit append action by setting the `APPEND ROW` attribute to `FALSE` in the `ATTRIBUTE` clause:

```plaintext
... INPUT ARRAY p_items FROM sa.* ATTRIBUTES(APPEND ROW=FALSE)
   ... END INPUT
...
```

When `APPEND ROW` or `INSERT ROW` attributes are set to `FALSE`, automatic temporary row can still occur when the user deletes the last row of the list or if the list is empty when the `INPUT ARRAY` is entered. To avoid automatic temporary row creation when only one of `APPEND ROW=FALSE` or `INSERT ROW=FALSE`, use `AUTO APPEND=FALSE`:

```plaintext
... INPUT ARRAY p_items FROM sa.* ATTRIBUTES(INSERT ROW=FALSE, AUTO APPEND=FALSE)
   ... END INPUT
...
```

If both `APPEND ROW` and `INSERT ROW` attributes are set to `FALSE`, the dialog will prevent explicit temporary row creation and also automatic temporary row creation, as if `AUTO APPEND = FALSE` was used.

**Row creation control blocks for temporary rows**

In order to control row creation, use the `BEFORE INSERT` and `AFTER INSERT` control blocks. The `BEFORE INSERT` trigger is invoked after a new row was inserted or appended, just before the user gets control to enter data in fields. Regarding temporary rows, the `AFTER INSERT` block is invoked if data has been entered and you leave the new row (for example, when the focus moves to another row or leaves the current list), or if the dialog is validated, for example with `ACCEPT DIALOG` in case of `DIALOG` (or `ACCEPT INPUT` in case of singular `INPUT ARRAY`). No `AFTER INSERT` block is invoked if the user did not enter data. The temporary row is automatically deleted.

In the `BEFORE INSERT` control block, you can tell if a row is a temporary appended one by comparing the current row (`DIALOG.getCurrentRow()` or `ARR_CURR()`) with the total number of rows.
DIALOG.getArrayLength() or ARR_COUNT()). If the current row index equals the row count, you are in a temporary row.

**AFTER ROW and temporary rows**

When a temporary row is automatically removed, the AFTER_ROW block will be executed for the temporary row, but ui.Dialog.getCurrentRow() / ARR_CURR() will be one row greater than DIALOG.getArrayLength() / ARR_COUNT(). In this case, ignore the AFTER_ROW event.

**Related concepts**

- **INPUT ARRAY ATTRIBUTES clause** on page 1603
- **INPUT ARRAY** specific attributes can be defined in the ATTRIBUTE clause of the sub-dialog header.
- **Predefined actions** on page 1740
  - Genero predefines some action names for common operations of interactive instructions.
- **The Dialog class** on page 2512
  - The ui.Dialog class provides a set of methods to configure, query and control the current interactive instruction.

**DISPLAY ARRAY modification triggers**

Using dedicated interaction blocks to allow the user to modify a read-only record list.

The DISPLAY_ARRAY block implements by default a read-only list of records. The end user can navigate in the list, but cannot modify the rows.

The traditional way to implement an editable list of record is to use INPUT_ARRAY. However, INPUT_ARRAY uses ergonomics that may not correspond to the end user expectations. Basically, a list controlled by an INPUT_ARRAY is always in "edit mode": the focus is in a field and the user can modify the current field. When moving up or down in the list, the edit cursor jumps to the upper or lower cell.

Other GUI applications use a different pattern, with read-only lists that can switch to edit mode when a specific action is fired. To implement such ergonomics, use the **ON_INSERT, ON_APPEND, ON_UPDATE, ON_DELETE** modification triggers to control row insertion, appending, modification and deletion in a DISPLAY_ARRAY block.

**Related concepts**

- **Example 3: DISPLAY ARRAY using modification triggers** on page 1529

**Cell color attributes**

List controllers can display every cell in a specific color.

When using the DISPLAY_ARRAY or INPUT_ARRAY, you can assign specific colors to cells of a TABLE or TREE rows with the DIALOG.setArrayAttributes() or DIALOG.setCellAttributes() method.

Call the method in the dialog initialization clause, for example, in BEFORE DISPLAY for a singular DISPLAY_ARRAY dialog.

The method takes an array as parameter. This array must have one of the following structures:

- **A DYNAMIC ARRAY OF RECORD** (with the same structure as the data array, but using the STRING data type)
- **A DYNAMIC ARRAY WITH DIMENSION 2 OF STRING** (to define attributes in dynamic dialog when the row structure is defined at runtime)
- **A DYNAMIC ARRAY OF STRING** (to define attributes for complete lines instead of individual cells)

Cell attributes are defined by using the TTY attributes such as "red reverse" for example. See method reference for all possible values.

**Note:** If cell attributes are changed during the dialog execution, use the UNBUFFERED mode to get automatic form synchronization. The unbuffered mode is not required if the cell attributes are defined before executing the dialog, and are left unchanged until the dialog ends.
**Example**

This is the list.per form file defining the table view:

```
LAYOUT
TABLE
{
  [c1 | c2    ]
}
END
END
ATTRIBUTES
  c1 = FORMONLY.key;
  c2 = FORMONLY.name;
END
INSTRUCTIONS
SCREEN RECORD sr(FORMONLY.*);
END
```

This is the program code (main.4gl):

```
MAIN
  DEFINE arr DYNAMIC ARRAY OF RECORD
    key INTEGER,
    name VARCHAR(100)
  END RECORD
  DEFINE att DYNAMIC ARRAY OF RECORD
    key STRING,
    name STRING
  END RECORD
  DEFINE I INT
  FOR i=1 TO 10
    LET arr[i].key = i
    LET arr[i].name = "Item "||i
    LET att[i].key = "red reverse"
    LET att[i].name = IIF(i MOD 2,"blue","green")
  END FOR
  OPEN FORM f1 FROM "list"
  DISPLAY FORM f1
  DISPLAY ARRAY arr TO sr.* ATTRIBUTES(UNBUFFERED)
  BEFORE DISPLAY
    CALL DIALOG.setCellAttributes(att)
  ON ACTION att_modify_cell
    LET att[2].key = "red reverse"
  ON ACTION att_clear_cell
    LET att[2].key = NULL
  END DISPLAY
END MAIN
```

**Related concepts**

*The buffered and unbuffered modes* on page 1720

The buffered and unbuffered mode control the synchronization of program variables and form fields.

**Multiple row selection**

Multiple row selection allows the end user to select several rows within a list of records.

The `DISPLAY ARRAY` controller supports multiple row selection when the `ON SELECTION CHANGE` block is defined, or by enabling the feature with the `ui.Dialog.setSelectionMode()` method when the dialog starts.
The `setSelectionMode()` method can also be used to enable or disable the multi-row selection during the dialog execution.

**Important:** This feature is not supported on mobile platforms.

When multi-row selection is enabled, the end user can select one or several rows with the standard keyboard and mouse click combinations. When the end user selects or deselected rows, the `ON SELECTION CHANGE` block is fired, if defined. The program can then query the `DIALOG.isRowSelected()` method to check for selected rows.

```display
DISPLAY ARRAY arr TO sr.*
...
ON SELECTION CHANGE
FOR i=1 TO DIALOG.getArrayLength("sr")
    DISPLAY SFMT("Row: %1 s=%2", i, DIALOG.isRowSelected("sr", i) )
END FOR
ON ACTION enable_mrs
    CALL DIALOG.setSelectionMode( "sr", 1 )
ON ACTION disable_mrs
    CALL DIALOG.setSelectionMode( "sr", 0 )
...
END DISPLAY
```

Multiple row selection is GUI-specific and therefore cannot be used in TUI mode.

With multiple row selection, you must distinguish between two concepts: row selection and current row. In GUI mode, a selected row usually has a blue background, while the current row has a dotted focus rectangle. The current row may not be selected, or a selected row may not be the current row. When the default single-row selection is used, the current row is always selected automatically.

If the `ON SELECTION CHANGE` block is not required, use the `ui.Dialog.setSelectionMode()` method to enable multi-row selection for the dialog:

```display
DISPLAY ARRAY arr TO sr.*
BEFORE DISPLAY
    CALL DIALOG.setSelectionMode( "sr", 1 )
...
END DISPLAY
```

Note that without the `ON SELECTION CHANGE` trigger, it is not possible to detect row selection changes when staying on the current row, since no BEFORE ROW/AFTER ROW trigger is fired in this case.

Row selection flags can be changed by program for a range of rows with the `DIALOG.setSelectionRange()` method.

The `DISPLAY ARRAY` dialog implements an implicit row-copy feature. The selected rows can be dragged to another dialog or external program, or the end-user can do an "editcopy" predefined action (Ctrl-C shortcut), to copy the selected rows to the front-end clipboard. The row-copy feature works also when multiple row selection is disabled, but only the current row will be dragged or copied to the front-end clipboard.

If you delete, insert or append rows in the program array with methods such as `array.deleteElement()`, selection information is not synchronized. To sync the selection flags with the data rows, use dialog methods like `DIALOG.insertRow()` (or `DIALOG.insertNode()` for tree-views).
## Behavior of ui.Dialog class methods with multiple row selection

### Table 392: Effect of ui.Dialog class on selection flags when multi-range selection is enabled

<table>
<thead>
<tr>
<th>Dialog class method</th>
<th>Effect on multiple row selection</th>
</tr>
</thead>
</table>
| `appendRow()`       | Selection flags of existing rows are unchanged.  
                      | New row is appended at the end of the list with selection flag set to zero. |
| `appendNode()`      | Selection flags of existing rows are unchanged.  
                      | New node is appended at the end of the tree with selection flag set to zero. |
| `deleteAllRows()`   | Selection flags of all rows are cleared. |
| `deleteRow()`       | Selection flags of existing rows are unchanged.  
                      | Selection information is synchronized (i.e., shifted up) for all rows after the deleted row. |
| `deleteNode()`      | Selection flags of existing rows are unchanged.  
                      | Selection information is synchronized (i.e., shifted up) for all nodes after the deleted node. |
| `insertRow()`       | Selection flags of existing rows are unchanged.  
                      | Selection information is synchronized (i.e., shifted down) for all rows after the new inserted row. |
| `insertNode()`      | Selection flags of existing rows are unchanged.  
                      | Selection information is synchronized (i.e., shifted down) for all nodes after the new inserted node. |
| `setArrayLength()`  | Selection flags of existing rows are unchanged.  
                      | If the new array length is larger than the previous length, selection flags of new rows are not initialized to zero. |
| `setCurrentRow()`   | Selection flags of all rows are reset, and the new current row gets selected. |
| `setSelectionMode()` | When you switch off multiple row selection, the selection flags of existing rows are cleared. |

### Related concepts

- **The buffered and unbuffered modes** on page 1720  
The buffered and unbuffered mode control the synchronization of program variables and form fields.

### Table views

Describes how to implement table/list views.

### Understanding tables views

Table views define the graphical element to display a list of records.

The end user can navigate in the list to select a row or edit rows, depending on the dialog controlling the table.

If the front-end platform standards allow it, the user can resize the table, sort rows, move/resize/hide columns, make multiple-row selections, search rows by criterion, and more.

**Figure 94: Table View (Linux/Gnome GDC)**
Tables views are controlled by a DISPLAY ARRAY or INPUT ARRAY instruction, using a form screen-array bound to a TABLE container.

When controlled by a DISPLAY ARRAY, the table view is by default read-only. However, you can implement modification triggers, to let the end user append, modify and delete rows.

When controlled by an INPUT ARRAY, the table view allows immediate data modification: The rows are editable.

Note that a table view can also be used with an INPUT or CONSTRUCT dialog.

You can customize the rendering and the behavior of table views with form attributes in the TABLE container, and in the program using the dialog implementation.

Related concepts
List dialogs on page 1831
Describes how to program dialogs controlling list containers.

Defining tables in the layout
Define table views in the LAYOUT section of the form definition file.

Designing table views

When using a grid-based layout, the table rows and columns are defined within an area delimited by curly brackets. Columns are defined with item tags and form fields. Every column tag must be properly aligned. You typically use a pipe character to separate the column tags.

A table definition using the TABLE layout item:

```
TABLE
{[
[c1 |c2 |c3 ]
[c1 |c2 |c3 ]
[c1 |c2 |c3 ]
}
END
```

Alternatively, you can define <TABLE> layout tags inside a GRID container, beside other layout tags:

```
GRID
{
<GROUP g1>
[f1 ]
[f2 ]
[
]
<}
```
Important: Avoid Tab characters (ASCII 9) inside the curly-brace delimited area. If used, Tab characters are replaced with 8 blanks at compilation with fglform.

The position of the item tags is detected by the form compiler to build the table. Column item types (widget to be used) and behavior are defined with form items in the ATTRIBUTES section:

```
<TABLE t1
[c1  |c2  |c3
[c1  |c2  |c3
[c1  |c2  |c3
<    |    |
>    |
END
```

```
ATTRIBUTES
EDIT c1 = customer.cust_id;
EDIT c2 = customer.cust_name;
EDIT c3 = customer.cust_address;
END
```

When using a stack-based layout, table views are defined with the TABLE stack item inside a STACK container. In this case, position/size and behavior are defined at a single place:

```
LAYOUT
STACK
  TABLE t1 (UNMOVABLECOLUMNS)
    EDIT customer.cust_id;
    EDIT customer.cust_name;
    EDIT customer.cust_address;
  END
END
END
```

**Controlling the size of the table**

In a grid-based container, the default width and height of a table are defined by the columns and the number of lines used in the table layout respectively. In a stack-based container, you can overwrite the default table by specifying the WIDTH and HEIGHT attributes.

```
TABLE t1 (WIDTH = 5 COLUMNS, HEIGHT = 10 LINES)
```

**Defining column titles**

The TABLE layout item definition can contain column titles as well as the tag identifiers for each column's form fields. The fglform form compiler can associate column titles in the table layout with the form field columns if they are aligned properly.

**Note:** At least two spaces are required between column titles.

```
TABLE
{
  Title1 Title2 Title3
[c1 |c2 |c3
[c1 |c2 |c3
[c1 |c2 |c3
}
END
```
Alternatively, you can set the column titles of a table container by using the \texttt{TITLE} attribute in the definition of the form fields. This allows you to use \texttt{localized strings} for the column titles.

\begin{verbatim}
TABLE
{
| c1 | c2 | c3 |
| c1 | c2 | c3 |
| c1 | c2 | c3 |
}
END
...
ATTRIBUTES
EDIT c1 = customer.cust_id, TITLE=%"label.cust_id";
EDIT c2 = customer.cust_name, TITLE=%"label.cust_name";
EDIT c3 = customer.cust_address, TITLE=%"label.cust_address";
END
\end{verbatim}

Similarly, in a stack item \texttt{TABLE} container, columns can get a \texttt{TITLE} attribute:

\begin{verbatim}
LAYOUT
STACK
TABLE t1(UNMOVABLECOLUMNS)
    EDIT customer.cust_id, TITLE=%"label.cust_id";
    EDIT customer.cust_name, TITLE=%"label.cust_name";
    EDIT customer.cust_address, TITLE=%"label.cust_address";
END
END
END
\end{verbatim}

**Height of table rows**

The height of table rows can be defined with a grid-based layout by adding empty tags underneath column tags (this makes sense only when using widgets that can get a height such as \texttt{TEXTEDIT} or \texttt{IMAGE}).

\begin{verbatim}
LAYOUT
TABLE
{
| c1 | c2 |
|    |    |
|    |    |
|    |    |
}
END
END
ATTRIBUTES
EDIT c1=FORMONLY.key;
TEXTEDIT c2=FORMONLY.thetext;
END
...
\end{verbatim}

In the above example, the second column is defined as a \texttt{TEXTEDIT} item type, that can get a height as a number of grid cells. The height is defined by the number of item tags of the table row in the layout section (height=3 in our example)

**Related concepts**

\texttt{TABLE container} on page 1326
Defines a re-sizable table designed to display a list of records.

**Binding tables to arrays in dialogs**

Program arrays act as data model that are bound to form tables, when implementing list dialogs.

**Identifying list views in program dialogs**

In list dialogs such as the `INPUT ARRAY` or `DISPLAY ARRAY`, the screen array identifies the record list element in the current form to be bound to the program array used by the dialog.

In this example, the `INPUT ARRAY` uses the `custlist` screen array of the form, and binds the `custarr` program array with:

```
INPUT ARRAY custarr FROM custlist.*
```

The screen array members are associated to the program array record members by position. The order and number of the screen array elements matter as they are bound by position to the members the program array. The position of the `TABLE` columns, however, can differ from the members of the screen array and program array.

To omit columns in the `TABLE` layout, yet include them in the definition of the screen array, and define the columns as `PHANTOM` fields in the form definition file. The program array can then be defined from the database table definition with the `DEFINE LIKE` instruction:

```
DEFINE custarr DYNAMIC ARRAY OF RECORD LIKE customer.*
```

Note that the array is usually defined with a flat list of members with `ARRAY OF RECORD / END RECORD`. However, the array can be structured with sub-records and still be used with a list dialog. This is especially useful when you need to define arrays from database tables, and additional information needs to be managed at runtime (for example to hold image resource for each row, to be displayed with the `IMAGECOLUMN` attribute):

```
SCHEMA shop
DEFINE a_items DYNAMIC ARRAY OF RECORD
    item_data RECORD LIKE items.*,
    it_image STRING,
    it_count INTEGER
END RECORD
...
DISPLAY ARRAY a_items TO sr.*
...
```

**Defining screen arrays in grid-based layout TABLES**

When using a grid-based layout, the `TABLE` container is bound to a screen array defined in the `INSTRUCTION` section, by the name of the form fields used in the screen array definition.

The column data type and additional column properties are defined in the `ATTRIBUTES` section as form fields:

```
LAYOUT
...
TABLE
{
[cl | c2 | c3 ]
[cl | c2 | c3 ]
[cl | c2 | c3 ]
[cl | c2 | c3 ]
}
END
...

ATTRIBUTES
EDIT c1 = customer.cust_num;
```
Each form field of the table must be grouped in the INSTRUCTIONS section in a SCREEN RECORD definition.

```
SCREEN RECORD custlist( cust_num, cust_name, cust_cdate );
```

### Defining screen arrays in stack-based layout TABLES

When using a stack-based layout, the TABLE stack item gets an identifier, which defines the screen array to be used in programs:

```
LAYOUT
  STACK
    TABLE custlist (STYLE="regular")
      EDIT customer.cust_num;
      EDIT customer.cust_name,
      EDIT customer.cust_cdate;
  END
END
END
```

This identifier is mandatory for TABLE stack items.

**Related concepts**

- [Array binding in list controllers](#) on page 1831

Program array elements are bound to screen array elements in the definition of the DISPLAY ARRAY or INPUT ARRAY list dialog.

**Controlling table rendering**

Table rendering can be controlled by the use of presentation styles and table attributes.

**Table width**

By default, the width of a TABLE container is defined by the number of visible columns in its layout.

**Note:** Table column fields defined as HIDDEN will not contribute to computing the default table width.

In this example, the table is defined with three visible columns, which in turn define individual field widths. The sum of the field widths defines the default table width.

```
GRID
{<TABLE t1          >
  [c1  |c2       |c3  ]
...}
```

To specify explicitly the width of a table, use the WIDTH attribute:

```
TABLE t1 : table1, WIDTH = 80 COLUMNS, ... ;
```

**Note:** The COLUMNS unit is in the sense of the layout as shown in grid columns.

**Table height**

The height of a TABLE container is by default defined by the number of rows in its layout. In the example, the default table height will be three rows:

```
GRID
```
To specify explicitly the height of a table, use the `HEIGHT` attribute:

```plaintext
TABLE t1 : table1, HEIGHT = 10 LINES, ... ;
```

**Note:** The `LINES` unit is in the sense of the layout as shown in grid lines.

### Defining a table with fixed height

By default, tables can be resized in height and behave as a stretchable element in the graphical window.

Use the `WANTFIXEDPAGESIZE` form file attribute to prevent table resizing:

```plaintext
TABLE t1 : table1, WANTFIXEDPAGESIZE, ... ;
```

### Current row / current cell rendering

In a `TABLE` container, the highlighting of the current row (or current cell, when focus granularity is at the cell level) can be controlled with style attributes.

**Note:** The current row / current cell style attributes apply also to stretchable `SCROLLGRID` containers.

For more details, see Row and cell highlighting in `TABLE` on page 1208.

### Making table columns unmovable

When the Table style attribute `tableType` is set to `frozenTable`, you can define a number of fixed columns on the left and right, respectively with the `leftFrozenColumns` and `rightFrozenColumns` style attributes:

```xml
<Style name="Table.custom_style">
  <StyleAttribute name="tableType" value="frozenTable"/>
  <StyleAttribute name="leftFrozenColumns" value="2"/>
  <StyleAttribute name="rightFrozenColumns" value="1"/>
</Style>
```

**Note:** With some front-ends (GBC), frozen columns can also be used in `Treeviews`.

See the reference topics Table style attributes on page 1204, Tree style attributes on page 1209.

### Mobile-style list view rendering

`TABLE` containers can be rendered with a mobile list view look and feel, by setting the `tableType` style attribute to "listView":

```xml
<Style name="Table.custom_style">
  <StyleAttribute name="tableType" value="listView"/>
</Style>
```

For the complete list of style attributes, see Table style attributes on page 1204.

### Current row visibility after dialog execution

When the dialog controlling the table has finished, the current row may be deselected, depending on the `KEEP CURRENT ROW` dialog attribute.
Related concepts

**TABLE container** on page 1326
Defines a re-sizable table designed to display a list of records.

### Displaying column images

You can use **PHANTOM** fields and the **IMAGECOLUMN** attribute to display images in a column, to the left of the column value.

To display an image on the left of the column value in table views, define a **PHANTOM** field to hold the image name, and bind it to a parent column with the **IMAGECOLUMN** attribute.

```plaintext
LAYOUT TABLE
{
[c1            |c2          
[c1            |c2          
[c1            |c2          
}
END
END
ATTRIBUTES
PHANTOM FORMONLY.file_icon;
EDIT c1 = FORMONLY.file_name, IMAGECOLUMN=file_icon;
EDIT c2 = FORMONLY.file_size;
...
END
INSTRUCTIONS
SCREEN RECORD sr(FORMONLY.*);
END
```

The program code can then display the specified image with each row.

```plaintext
DEFINE arr DYNAMIC ARRAY OF RECORD
  file_icon STRING,
  file_name STRING,
  file_size INTEGER
END RECORD
...
FOR x=1 TO max_files
  CASE file_type(arr[x].file_name)
    WHEN "file" LET arr[x].file_icon = "file"
    WHEN "dir"  LET arr[x].file_icon = "folder"
  END CASE
END FOR
... DISPLAY ARRAY arr TO sr.*
...
END DISPLAY
```

When images come from the database, these are typically fetched into **BYTE** variables. If the **BYTE** variable is located in a file (**LOCATE IN FILE**), it can be bound to the **IMAGECOLUMN** field: The runtime system will automatically display the image data. Note, however, that each **BYTE** element of the array must be located in a distinct file. This can be done as follows:

```plaintext
DEFINE arr DYNAMIC ARRAY OF RECORD
  pic_num INTEGER,
  pic_data BYTE,
  pic_when DATETIME YEAR TO SECOND
END RECORD
...
DECLARE c1 CURSOR FOR SELECT * FROM mypics
```
LET i=1
LOCATE arr[i].pic_data IN FILE
FOREACH c1 INTO arr[i].*
    LOCATE arr[i:=i+1].pic_data IN FILE
END FOREACH
CALL arr.deleteElement(i)

Depending on the data source, you might want to use a program array structured with sub-records, to define database table related data from row information used at runtime only, as described in Variable binding in DISPLAY ARRAY on page 1504:

SCHEMA shop
DEFINE a_items DYNAMIC ARRAY OF RECORD
    item_data RECORD LIKE items.*,
    it_image STRING,
    it_count INTEGER
END RECORD

Related concepts
Runtime images on page 1154
Explains how to display pictures at runtime.

Image columns firing actions
Columns in tables displaying images can trigger action events, when the user selects the image.

TABLE and TREE containers can define columns as IMAGE field, to display pictures or icons. By default, these table cells are not clickable. When you define an ACTION attribute for a table column defined as IMAGE, the action event will fire when the image is selected (with a mouse click, for example). Note that this does not apply to the IMAGECOLUMN concept, which is rather a column decoration.

Important: When selecting an image, the current row may change as when selecting a new row in the table.

The following example defines a TABLE with two IMAGE columns, and attaches the update and delete actions:

LAYOUT
TABLE
{
    [c1 | c2 | i1 | i2]
    [c1 | c2 | i1 | i2]
    [c1 | c2 | i1 | i2]
}
END
END
ATTRIBUTES
EDIT c1 = FORMONLY.id, TITLE="Id", NOENTRY;
EDIT c2 = FORMONLY.name, TITLE="Name";
IMAGE i1 = FORMONLY.i_modify, ACTION=update;
IMAGE i2 = FORMONLY.i_delete, ACTION=delete;
END
INSTRUCTIONS
SCREEN RECORD sr(FORMONLY.*);
END

In the program code, use a dialog instruction to implement the action handlers for the image actions. For example, you can define a DISPLAY ARRAY with ON UPDATE and ON DELETE list modification triggers that will respectively create the update and delete actions:

DISPLAY ARRAY arr TO sr.*
ON UPDATE
    -- user code
ON DELETE
Related concepts

**Binding action views to action handlers** on page 1762
How are action views of the forms bound to action handlers in the program code?

**DISPLAY ARRAY modification triggers** on page 1850
Using dedicated interaction blocks to allow the user to modify a read-only record list.

**Built-in table features**
Several implicit list handling features are provided by table views.

**Columns layout**
By default, a user can position, hide, show, and resize columns in **TABLE** and **TREE** containers.

**Important:** This feature is not supported on mobile platforms.

**Resizing columns**
By default, columns can be resized. On desktop front-ends, the user can drag the right edge of a column header to increase or decrease the width of the column.

To prevent column resizing for all columns in a table, add the **UNSIZABLECOLUMNS** attribute to the **TABLE** or **TREE** container.

To prevent column resizing for an individual column, add the **UNSIZABLE** attribute to the form field definition for that column.

**Hiding/showing columns**
By default, the user can control the visibility of columns. On desktop front-ends, a user right-clicks on a column header to get a context menu that allows the show/hide columns.

To disable the column visibility option for all columns in a table, add the **UNHIDABLECOLUMNS** attribute to the **TABLE** or **TREE** container.

To disable the column visibility option for an individual column, add the **UNHIDABLE** attribute to the form field definition for that column.

To hide a column initially but allow column visibility, set the **HIDDEN** attribute with the value **USER** in the form field definition for that column. This hides the column by default, and allows the user to show the column if needed.

**Changing column positions**
By default, columns can be moved around. On desktop front-ends, a user can rearrange columns by dragging the column header to a the left or to the right.

To disable this option, add the **UNMOVABLECOLUMNS** attribute to the **TABLE** or **TREE** container.

To disable this option for an individual column, add the **UNMOVABLE** attribute to the form field definition for that column.

**List ordering**
List controllers implement a built-in sort. This feature can be disabled if not required.

When a **DISPLAY ARRAY** or **INPUT ARRAY** block is combined with a **TABLE** container, the row sorting feature is implicitly available. Row sorting is supported on **TREE** containers with **DISPLAY ARRAY** dialogs only.

**Important:** This feature is not supported on mobile platforms.

To sort rows in a list, the user must click on a column header of the table. Clicking on a table column header triggers a GUI event that instructs the runtime system to reorder the rows displayed in the list container.
In fact, the rows are only sorted from a visual point of view; the data rows in the program array (the model) are left untouched. Therefore, when rows are sorted, the visual position of the current row might be different from the current row index in the program array.

To sort rows, the runtime system uses the standard collation order of the system, following the current locale settings. As a result, the rows might be ordered a bit differently than when using the database server sort (with an ORDER BY clause of the SELECT statement), since database servers can define their own collation sequences to sort character data.

The built-in sort is enabled by default. To prevent sorting in TABLE or TREE containers, define the UNSORTABLECOLUMNS attribute at the list container level, or set the UNSORTABLE attribute at the column/field level. As rows can be created and modified during an INPUT ARRAY instruction, you may want to use the UNSORTABLECOLUMNS attribute for tables controlled by INPUT ARRAY.

To execute code after a sort was performed, use the ON SORT interaction block in the dialog, for example to display the current row position with ui.Dialog.arrayToVisualIndex on page 2527.

The sorting feature is disabled when using the paged mode of DISPLAY ARRAY, because not all result set rows are known by the runtime system in this mode. However, it is possible to detect a sort request from the user with the ON SORT trigger. You can then re-execute the SQL query with a new sort order. For more details, see Populating a DISPLAY ARRAY on page 1841.

When an application window is closed, the selected sort column and order is stored by the front-end in the user settings database of the system (for example, on Windows® platforms it's the registry database). The sort will be automatically re-applied the next time the window is created. This way, the rows will appear sorted when the program restarts. The saved sort column and order is specific to each list container.

Related concepts
Record list (DISPLAY ARRAY) on page 1501
The DISPLAY ARRAY instruction provides record list navigation in an application form, with optional record modification actions.

Editable record list (INPUT ARRAY) on page 1531
The INPUT ARRAY instruction provides always-editable record list handling in an application form.

Paged mode of DISPLAY ARRAY on page 1843
In order to handle very large result sets, use the paged mode of DISPLAY ARRAY.

Full list mode of DISPLAY ARRAY on page 1842
In order to handle short/medium result sets, use the full list mode of DISPLAY ARRAY.

Find function
List controllers implement a built-in find. This feature can be disabled if not required.

The DISPLAY ARRAY and INPUT ARRAY block blocks support the built-in find feature by default.

Important: This feature is not supported on mobile platforms.

This feature works with any list container (TABLE, TREE, SCROLLGRID).

The built-in find creates the implicit "find" and "findnext" actions. These actions can be decorated, enabled and disabled as regular actions.

Note: If the dialog defines an explicit ON ACTION find or ON ACTION findnext, the default built-in find is disabled.

When the user triggers the "find" action (default accelerator is Ctrl-F), the dialog opens a pop-up window to enter a search value and search options. On validation with the OK button, the dialog starts to search a row where a field value matches the value entered in the find dialog. The "find" action starts the search from the current row (and in the field after the current field, if the dialog is an INPUT ARRAY).

After a "find" action, the user can trigger the "findnext" action (default accelerator is Ctrl-G), in order to continue the search, without opening the find dialog again (the current search value will be reused).
By default, any table column is scanned, but the user can select a specific column in the find dialog box, as long as a column title is available. Case-sensitive or insensitive search as well as wraparound options are also available.

**Important:** Only rows in memory can be searched. When using the paged-mode (ON FILL BUFFER), the built-in search is disabled. When implementing dynamic tree views, the built-in find will only search the tree nodes available in the program array.

The value entered in the find dialog is compared to type of columns, except columns using the type TEXT or BYTE. The comparison is based on the formatted value. For example, a MONEY column will display values formatted with the currency symbol. To match values in that column, the user must enter exactly the same value (that is with the currency symbol and the correct decimal separator).

Only text widgets displaying values are searched. Columns using widgets such as images, radio-groups, checkboxes are not searched. Furthermore, the find function ignores PHANTOM fields, hidden fields and fields defined with the INVISIBLE attribute.

With COMBOBOX fields, the find searches in the visible values of combobox items.

When the dialog is an INPUT ARRAY and no specific search column is selected in the find dialog, the search scans each cell. The search starts in the current row, after the current field. If no cell value matches in the current row, the search continues on the first field of the next row.

**Tip:** The built-in find feature is also available in text mode. In graphical mode, the default keyboard accelerator is Ctrl-F. When using text mode, the accelerator is the / slash key.

**Keyboard seek**

The keyboard seek feature allows a user to find a row in a read-only list, by typing characters.

When a DISPLAY ARRAY is used with a list, the keyboard seek feature is automatically implemented. A user may type alphabetic characters on the keyboard to have the runtime system automatically seek the next row having a character field that contains a value starting with the typed characters. The seek search restarts from the current row when the user types new characters on the keyboard.

**Important:** This feature is not supported on mobile platforms.

This feature works with any list container (TABLE, TREE, SCROLLGRID).

Numeric, date/time and large data (TEXT/BYTE) columns are ignored. Only character columns are searched, fields using widgets like image, radio-group or checkbox are ignored. Furthermore, the seek function ignores PHANTOM fields, hidden fields and fields defined with the INVISIBLE attribute.

The user can rapidly type several characters on the keyboard, to search for a value that starts with the typed characters. After a given timeout (less than a second), the seek buffer is cleared and a new search filter can be applied. The seek search is case-insensitive.

If no row is found from the typed characters, the [Not found] error -8105 is displayed automatically.

If an alphabetic character is used as action accelerator, the built-in seek feature is disabled, because the accelerator must fire the corresponding action.

Only rows in memory can be searched. When using page-mode (ON FILL BUFFER), the built-in seek is disabled. When implementing dynamic tree views, the built-in seek will only search the tree nodes available in the program array.

By default, any character column of the list is scanned. But if the list gets sorted, the runtime system considers that the sort column is the most important and searches only in that column.

**Related concepts**

Find function on page 1863
List controllers implement a built-in find. This feature can be disabled if not required.

**Reduce filter**
The reduce filter allows a user to limit the row set in the list by using a filter.

**Understanding the reduce filter**
On mobile devices, when using a DISPLAY ARRAY with a TABLE or SCROLLGRID container, and if the front-end supports filter search facility, the user can enter a criterion in that search field, to show only the rows matching the content of the filter.

**Important:** This feature is only for mobile platforms.

**Reduce filter with GMI/iOS**
On iOS devices, the reduce filter is only available with full list views.

**Figure 95: iOS list view with filter field**

![iOS list view with filter field](image)

**Reduce filter with GMA/Android™**
On Android™ devices, the reduce filter is available for full and embedded list views.

**Figure 96: Android™ list view with filter field**

![Android™ list view with filter field](image)
Reduce filter usage details

The filter search is case-insensitive.

The value entered in the filter field is compared to all fields of visible columns, except columns of the type TEXT or BYTE. The comparison is based on the formatted value. For example, a MONEY column will display values formatted with the currency symbol. To match values in that column, the user must enter exactly the same value (for example, with the currency symbol and the correct decimal separator). When using COMBOBOX fields, the find searches in the visible values of combobox items.

Only text widgets displaying values are searched. Columns using widgets such as images, radio-groups or checkboxes are not searched. The filter function ignores PHANTOM fields, hidden fields and fields defined with the INVISIBLE attribute.

Only rows in memory can be searched. When using page-mode (ON FILL BUFFER), the built-in filter is disabled. When implementing dynamic tree views, the built-in filter will only search the tree nodes available in the program array.

If the rows are filtered (some value is present in the search field), any non-rowbound action is disabled. On iOS, the action bar is replaced by the search bar.

Disabling the reduce filter

The reduce filter can be disabled with the reduceFilter style attribute:

```
<Style name="Table">
    <StyleAttribute name="reduceFilter" value="no" />
</Style>
```

Summary lines in tables

Table views can display a summary line, to show aggregate values for columns.

To get a summary line in a table, define aggregate field item tags at the bottom of the TABLE container, with the corresponding AGGREGATE form item definitions in the ATTRIBUTES section.

Important: This feature is not supported on mobile platforms.
Define the type of the aggregate field with the `AGGREGATETYPE` attribute: The aggregate value can be automatically computed, or set by program.

To get a global label for the summary line, specify the `AGGREGATETEXT` attribute at the `TABLE` level. This aggregate label will appear on the left in the summary line, if no aggregate text is defined at the aggregate field level.

To decorate the summary line, use presentation style attributes such as `summaryLineAlwaysAtBottom`.

This example defines a "total" aggregate field for the third column of the table:

```plaintext
TABLE (AGGREGATETEXT="Total")
{
  [c1  |c2          |c3         ]
  [c1  |c2          |c3         ]
  [total      ]
}
END
... ATTRIBUTES
... AGGREGATE total = FORMONLY.total, AGGREGATETYPE=PROGRAM;
...```

For details, see `Aggregate fields` on page 1248.

**Defining the action for a row choice**
The row choice in a `TABLE` can be associated with a dedicated action.

When using a `DISPLAY ARRAY` dialog to control a `TABLE` container, a double-click on a row (for a desktop client), or a tap on a row (for mobile clients) fires by default the "accept" action.

When using an `INPUT ARRAY`, a mouse double-click is typically use to select text in the current cell. However, when defining the `DOUBLECLICK` attribute in the `TABLE/TREE/SCROLLGRID` container in the form file, a double-click can fire the specified action during an `INPUT ARRAY`.

Row choice action can be customized as described in `Defining the action for a row choice` on page 1838.

**Using tables on mobile devices**
Table views render in a specific way on mobile devices, in order to take advantage of mobile device ergonomics.

**Unsupported table features**
Some table / list view features are not supported on mobile devices.

The list view features not supported on mobile devices include:

- Multiple row selection on page 1851
- Summary lines in tables on page 1866
- List ordering on page 1862
- Find function on page 1863
- Keyboard seek on page 1864
- Columns layout on page 1862
- Drag & drop on page 1917
- Using `IMAGE` columns. (One `IMAGECOLUMN` can be defined, however)

Note also that there are no column headers/titles in mobile list views.

**Related concepts**
`Built-in table features` on page 1862
Several implicit list handling features are provided by table views.

**Two-column display**

On mobile devices, a `TABLE` container displays as a list view with the first two columns' content.

While a `TABLE` container and the corresponding list controller (`DISPLAY_ARRAY`) can define multiple columns, only the first two columns are rendered on a mobile device. The first column defines the main information to be shown for the row (such as a customer name), while the second column contains additional information (such as a comment, date, address or phone number).

**Figure 97: iOS list view with two-column default rendering**

If the second column contains numeric data or has the `JUSTIFY=RIGHT` attribute, both columns display on a single line with the first column left-aligned and the second column right-aligned.

**Figure 98: iOS list view with side-by-side rendering**
A list view on a mobile device can include an image for each row. To display an image, associate a PHANTOM column to the IMAGECOLUMN attribute of the first column definition. For more details about images in lists, see Displaying column images on page 1860.

Figure 99: iOS list view with row images
Related concepts

Record list (DISPLAY ARRAY) on page 1501

The DISPLAY ARRAY instruction provides record list navigation in an application form, with optional record modification actions.

Full and Embedded list views

On mobile devices, table views are displayed as either full screen lists or embedded lists, based on the layout definition.

Full list view

A full list view displays when the table is the only element in a form.

```
LAYOUT TABLE
  { [c1 | c2 ]
} END
END
```

Figure 100: iOS full list view rendering
Embedded list view

An embedded list view displays when the table is mixed with other form elements.

In this example, the table is inside a GRID container:

```xml
LAYOUT
GRID
{
  <GROUP g1>
    Id: [f1]
    Name: [f2]
  <>
  <GROUP g2>
    <TABLE t1>
      [c1 | c2 ]
      [c1 | c2 ]
      [c1 | c2 ]
    <>
    <>
  </TABLE t1>
  <>
}<
END
END
```

Figure 101: iOS embedded list view rendering
Related concepts

**LAYOUT section** on page 1301

The LAYOUT section defines the graphical alignment of the form by using a tree of layout containers.

**The DOUBLECLICK (tap) action**

On mobile devices, the DOUBLECLICK attribute defines the action to fire when a row is tapped.

By default, no action is fired on mobile devices when the user taps on a row. To fire a dedicated action, add the DOUBLECLICK attribute to the DISPLAY ARRAY dialog and define an ON ACTION action handler.

```plaintext
DISPLAY ARRAY arr TO sr.*
    ATTRIBUTES( DOUBLECLICK=row_select )
    ON ACTION row_select
        CALL process_row(arr_curr())
```

Alternatively, you can add a DOUBLECLICK attribute to your TABLE definition in your form file.

**Tip**: We recommend you specify the DOUBLECLICK attribute with the DISPLAY ARRAY dialog, as it is strongly related to the DISPLAY ARRAY dialog.

**Note:**

- On Android™ devices, a long tap on a row only selects the row. The DOUBLECLICK action is not fired.
• For iOS devices, consider using list view decoration options, as described in Row configuration on iOS devices on page 1876.

**Related concepts**

Defining the action for a row choice on page 1867

The row choice in a TABLE can be associated with a dedicated action.

Record list (DISPLAY ARRAY) on page 1501

The DISPLAY ARRAY instruction provides record list navigation in an application form, with optional record modification actions.

**Rowbound actions**

A rowbound action specifies an action to apply to the selected row. Rowbound actions get specific rendering and behavior on mobile devices.

Rowbound actions are action defined with the ROWBOUND action attribute in ON ACTION handlers. Rowbound actions can also be default actions that are implicitly related to the current row, such as the "delete" action.

```plaintext
DISPLAY ARRAY arr TO sr.*

... ON ACTION clear_list -- not rowbound
... ON ACTION copy_row ATTRIBUTES(ROWBOUND, TEXT="Copy row")
... ON ACTION check_row ATTRIBUTES(ROWBOUND, TEXT="Check row")
... ON DELETE -- implicitly rowbound
...```

**Genero Mobile for Android™ (GMA)**

On Android™ 4 devices, when rowbound actions are defined, each row of a list view shows the three-dot indicator. Tap this icon to bring up a row context menu with options to execute the corresponding rowbound actions. Swipe the row from the right to the left to fire the delete action, it defined.
Figure 102: Android™ list view with rowbound actions

Genero Mobile for iOS (GMI)

On iOS 7 devices, when you swipe your finger from right to left, More... and/or Delete icons show up in the row. Tap More... to bring up a list of rowbound actions to execute. Tap Delete to fire the corresponding delete action code.
Figure 103: iOS list view with rowbound actions

Related concepts
Configuring actions on page 1744
Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with action attributes.

Close, accept and cancel actions
The default rendering of the close, accept and cancel actions with a list view depends on the mobile device.

A DISPLAY ARRAY dialog implements the "close", "accept", and "cancel" actions by default.

When using a full list view, these actions are default action views.
The rendering of these actions vary depending on the type of mobile device. The buttons for accept and cancel actions typically show up on the top of the list view.

For more details, see Action views on mobile devices on page 1777.

**Related concepts**

Default actions in DISPLAY ARRAY on page 1506

**Row configuration on iOS devices**

On iOS devices, table views can be configured to use specific row decorations.

**Note:** The features described in this topic are provided for iOS devices. The decoration attributes are ignored by Genero Mobile for Android™ (GMA)

**The ACCESSORYTYPE attribute**

On iOS devices, the ACCESSORYTYPE attribute used in the DISPLAY ARRAY dialog ATTRIBUTES clause defines the type of icon that appears at the right side of each row.

Possible values for the ACCESSORYTYPE attribute are:

- CHECKMARK
- DETAILBUTTON
- DISCLOSUREINDICATOR

For more details about the ATTRIBUTES syntax, see Syntax of DISPLAY ARRAY instruction on page 1502.

**Checkmark**

When using ACCESSORYTYPE=CHECKMARK, the current row gets a check mark icon on the right hand side.

This decoration is typically used to get a visual indicator for the current row, so the user knows what row will be selected when the DISPLAY ARRAY dialog is validated with an accept (Done) action:

```
DISPLAY ARRAY arr TO sr.*
   ATTRIBUTES( ACCESSORYTYPE=CHECKMARK )
   ...
```
To customize the application, define the color of the check mark with the `iosTintColor` Window-class style attribute.

**Detail button**

When using `ACCESSORYTYPE=DETAILBUTTON`, each row gets an (i) icon on the right-hand side.

To specify what action must be fired when the user taps on the (i) icon, define the `DETAILACTION` in the `DISPLAY ARRAY` attributes, and its corresponding `ON ACTION` handler.

By opening a new window when in the detail action code, a tap on the icon shifts the current window from right to left, to show the new screen.

When tapping on another part of a row, by default, the row becomes then the new current row. To follow typical iOS standards, it is recommended that you also define a `DOUBLECLICK` with its corresponding `ON ACTION` handler, to handle current row selection with a dedicated action. If tapping on any part of a row opens a detail form, use the `DISCLOSUREINDICATOR` solution instead of `DETAILBUTTON`. When selecting a different row, the `AFTER ROW / BEFORE ROW` control blocks are executed before the detail action or double-click action.

```plaintext
DISPLAY ARRAY arr TO sr.*
  ATTRIBUTES ( ACCESSORYTYPE=DETAILBUTTON,
               DETAILACTION=edit_details,
               DOUBLECLICK=select_row )
  ...
  ON ACTION edit_details
    OPEN WINDOW w_details WITH FORM "details"
    INPUT BY NAME arr[i].*
    ...
    END INPUT
  CLOSE WINDOW w_details
  ON ACTION select_row
```
To customize the application, define the color of the disclosure indicator with the `iosTintColor` Window-class style attribute.

**Disclosure indicator**

When using `ACCESSORYTYPE=DISCLOSUREINDICATOR`, each row gets a > gray chevron at the right of each row. This decoration is typically used when tapping the button brings up a list of more choices related to the current row, or to open a detail form to modify the list element.

To execute code when tapping on a row, define the `DOUBLECLICK` attribute and its corresponding `ON ACTION` handler.

By opening a new window when in the detail action code, a tap on a row shifts the current window from right to left, to show the new screen.

When selecting a different row, the `AFTER ROW / BEFORE ROW` control blocks are executed before the double-click action.

```display_array
DISPLAY ARRAY arr TO sr.*
    ATTRIBUTES ( ACCESSORYTYPE=DISCLOSUREINDICATOR,
                   DOUBLECLICK=row_select )
...
ON ACTION row_select
    MENU "Options" ATTRIBUTES (STYLE="dialog")
    COMMAND "Refresh"
    ...
    COMMAND "Duplicate"
    ...
    COMMAND "Compress"
    ...
```
Figure 106: iOS list view with disclosure indicator

Related concepts
AFTER ROW block on page 1511
BEFORE ROW block on page 1510
Record list (DISPLAY ARRAY) on page 1501

The DISPLAY ARRAY instruction provides record list navigation in an application form, with optional record modification actions.

Examples
TABLE (table views) usage examples.

Example 1: Simple list view
The form file table.per (grid-based layout):

```
LAYOUT TABLE
{
[c1 |c2 ]
}
END
END
ATTRIBUTES
PHANTOM FORMONLY.key;
c1 = FORMONLY.name, IMAGECOLUMN=image;
PHANTOM FORMONLY.image;
```
c2 = FORMONLY.detail;
END
INSTRUCTIONS
SCREEN RECORD list1(FORMONLY.*);
END

The form file table.per (stack-based layout):

LAYOUT
STACK
TABLE list1
   PHANTOM FORMONLY.key;
   EDIT FORMONLY.name,
      IMAGECOLUMN=image, TITLE="Name";
   PHANTOM FORMONLY.image;
   EDIT FORMONLY.detail, TITLE="Detail";
END
END
END

The program main.4gl:

MAIN
DEFINE arr DYNAMIC ARRAY OF RECORD
   key INTEGER,
   name STRING,
   image STRING,
   detail STRING
END RECORD,
   i INTEGER
FOR i=1 TO 60
   LET arr[i].key = i
   LET arr[i].name = SFMT("Item %1", i)
   IF i MOD 2 THEN
      LET arr[i].image = "file"
   ELSE
      LET arr[i].image = "smiley"
   END IF
   LET arr[i].detail = SFMT("This is item %1", i)
END FOR
OPEN FORM f1 FROM "table"
DISPLAY FORM f1
DISPLAY ARRAY arr TO list1.* ATTRIBUTES(UNBUFFERED,DOUBLECLICK=myselect)
ON ACTION myselect
   MESSAGE "myselect:", arr_curr()
END DISPLAY
END MAIN

Related concepts
Runtime images on page 1154
Explains how to display pictures at runtime.

Scrollgrid views
Describes how to implement scrollgrid views.

Understanding scrollgrid views
A scrollgrid view defines a graphical element to display a scrolling list of data records in a set of form fields positioned in a grid.

The scrollgrid is defined as a template for form fields that make up the scroll grid elements. The front-end clones this template for each record that fits in the enclosing scrollgrid container. The end user sees a multiple-record view
like a list with a vertical scrollbar that can be navigated to select or edit rows, depending on the dialog controlling the scrollgrid.

By default, scrollgrids display a fixed number of visible records, as defined by the number of templates in the grid layout. If the front-end platform allows it, the scrollgrid can be configured to be resizable in height so that if the front-end screen or page size grows, the scrollgrid resizes and more records are visible.

Figure 107: Form with resizable scrollgrid view (GDC desktop front-end)

A usual pattern on the Web is to render information as a responsive tile list, using tiles displayed in pages. A resizable scrollgrid can easily be configured as a responsive tile list.
Scrollgrid view controllers

Scrollgrid views are similar to table views in that they are controlled by a DISPLAY ARRAY or INPUT ARRAY instruction, using a form screen-array bound to a SCROLLGRID container.

When controlled by a DISPLAY ARRAY, the scrollgrid view is by default read-only. However, you can implement modification triggers, to let the end user append, modify and delete rows.

When controlled by an INPUT ARRAY, the scrollgrid view allows immediate data modification: The rows are editable.

Note: A scrollgrid view can also be used with an INPUT or CONSTRUCT dialog: In such case, the user can input field values in the first scrollgrid row only.

You can customize the rendering and the behavior of scrollgrid views with form attributes in the SCROLLGRID container, and in the program using the dialog implementation.

Related concepts
List dialogs on page 1831
Describes how to program dialogs controlling list containers.

Defining scrollgrid in the layout
Define scrollgrid views in the LAYOUT section of the form definition file.

Designing scrollgrid views
The scrollgrid rows are defined as a template within an area delimited by curly brackets.

Figure 108: Form with resizable scrollgrid view (GBC web front-end)
This is an example of a resizable scrollgrid definition using the SCROLLGRID layout item. It is resizable because it has no fixed page size (WANTFIXEDPAGESIZE=NO). Only one template is needed to define the records.

The position of the item tags is detected by the form compiler to build the scrollgrid. Item types (widget to be used) and behavior are defined with form items in the ATTRIBUTES section.

```
LAYOUT
SCROLLGRID (WANTFIXEDPAGESIZE=NO)
{
    Id: [f1 ]
    Name: [f2 ][f3 ]
}
END
END
...

ATTRIBUTES
EDIT f1 = customer.cust_num;
EDIT f2 = customer.cust_fname,
EDIT f3 = customer.cust_lname;
...
```

You can also define SCROLLGRID layout tags inside a GRID container, beside other layout tags:

```
LAYOUT
GRID
{
    <SCROLLGRID sg1>
    Id: [f1 ]
    Name: [f2 ][f3 ]
    ...
}
END
END

ATTRIBUTES
SCROLLGRID sg1: scrollgrid1,
    WANTFIXEDPAGESIZE=NO,
    GRIDCHILDRENINPARENT;
...
```

**Important:** Avoid Tab characters (ASCII 9) inside the curly-brace delimited area. If used, Tab characters are replaced with 8 blanks at compilation with fglform.

**Height of scrollgrid rows**

The height of scrollgrid rows can be defined within a grid-based layout by adding empty tags underneath (this makes sense only when using widgets that can get a height such as TEXTEDIT or IMAGE).

In this example, the height of the image item type is defined by the number of rows in the layout section (6 in our example).

```
LAYOUT
SCROLLGRID (WANTFIXEDPAGESIZE=NO)
{
    Icon [name ] Index[index ]
    [img ]
    [ ]
    [ ]
    [ ]
    [ ]
}
```
Defining the rendering type of the scrollgrid

The scrollgrid can be configured to be fixed page size or resizable, depending on requirements or the supported platform:

- To define a scrollgrid with a fixed page size, omit the `WANTFIXEDPAGESIZE=NO` attribute. The number of visible rows is defined by the number of row templates in the form layout.
- To define a resizable scrollgrid, the `WANTFIXEDPAGESIZE=NO` is set in the layout to allow the container to stretch vertically. Only one record row template needs to be defined. Then define the initial number of scrollgrid lines with the `INITIALPAGESIZE` form definition attribute.
- To render a scrollgrid as a paged responsive tile list, the `customWidget` presentation style is defined and the `WANTFIXEDPAGESIZE=NO` scrollgrid attribute is used.

For more details, see Controlling scrollgrid rendering on page 1885.

Related concepts

**SCROLLGRID container** on page 1325
Defines a scrollable grid view widget, in a grid-based layout.

**Binding scrollgrids to arrays in dialogs**

Program arrays act as data model that are bound to form scrollgrids, when implementing list dialogs.

**Identifying list views in program dialogs**

In list dialogs such as the `INPUT ARRAY` or `DISPLAY ARRAY`, the screen array identifies the record list element in the current form to be bound to the program array used by the dialog.

In this example, the `INPUT ARRAY` uses the `custlist` screen array of the form, and binds the `custarr` program array with the code:

```
INPUT ARRAY custarr FROM custlist.*
```

The screen array members are associated with the program array record members by position. The order and number of the screen array elements is important as they are bound by position to the members of the program array.

The position of the `SCROLLGRID` fields, however, can differ from the members of the screen array and program array.

To omit fields in the `SCROLLGRID` layout, yet include them in the definition of the screen array, and define the fields as `PHANTOM fields` in the form definition file. The program array can then be defined from the database table definition with the `DEFINE LIKE` instruction:

```
DEFINE custarr DYNAMIC ARRAY OF RECORD LIKE customer.*
```

The array is usually defined with a flat list of members with `ARRAY OF RECORD / END RECORD`. However, the array can be structured with sub-records and still be used with a list dialog. This is especially useful when you need to define arrays from database tables, and additional information needs to be managed at runtime (for example to hold image resource for each row, to be displayed with the `IMAGECOLUMN` attribute):
Defining screen arrays in grid-based layout scrollgrids

When using a grid-based layout, the SCROLLGRID container is bound to a screen array defined in the INSTRUCTIONS section, by the name of the form fields used in the screen array definition.

The form data type and additional properties are defined in the ATTRIBUTES section as form fields:

```
LAYOUT
SCROLLGRID (WANTFIXEDPAGESIZE = NO)
{
  Id:   [f1    ]
  Name: [f2           ]   [f3           ]
}
END
END

ATTRIBUTES
EDIT f1 = customer.cust_num;
EDIT f2 = customer.cust_fname,
EDIT f3 = customer.cust_lname;
...
```

Each form field of the scrollgrid must be grouped in the INSTRUCTIONS section in a SCREEN RECORD definition.

```
INSTRUCTIONS
SCREEN RECORD custlist( cust_num, cust_fname, cust_lname );
END
```

Defining screen arrays in stack-based layout scrollgrids

Since scrollgrids define a grid-based list view, they do not have a stacked view in the sense of TABLE or TREE views.

Related concepts
Variable binding in DISPLAY ARRAY on page 1504
Variable binding in INPUT ARRAY on page 1535

Controlling scrollgrid rendering
Scrollgrid rendering can be controlled by the use of presentation styles and scrollgrid attributes.

Scrollgrid resize control
By default scrollgrids are not resizable and the number of fixed rows is defined by the layout element. The WANTFIXEDPAGESIZE form file attribute controls the vertical resizing of the list elements. Set this attribute to NO, in order to get a resizable scrollgrid:

```
LAYOUT
SCROLLGRID ( WANTFIXEDPAGESIZE=NO )
{
  ...
```
**Minimal number of scrollgrid lines**

With a resizable scrollgrid, you can define the initial number of rows with the `INITIALPAGESIZE` form file attribute:

```
LAYOUT SCROLLGRID ( WANTFIXEDPAGESIZE=NO, INITIALPAGESIZE=4 )
{
...
```

**Note:** This is mainly useful in modal windows, to define the initial window size.

**Paged scrollgrids (tile list)**

A scrollgrid can be rendered as a tile list to fit records horizontally and vertically in a page to respond to the container size when stretched or shrunk.

To enable a paged scrollgrid, in your .4st file, define the `customWidget` style attribute to the value `pagedScrollGrid`:

```
<Style name="ScrollGrid.paged">
  <StyleAttribute name="customWidget" value="pagedScrollGrid" />
</Style>
```

In the scrollgrid layout definition, use the style name as defined in your styles file:

```
LAYOUT SCROLLGRID ( WANTFIXEDPAGESIZE=NO, STYLE="paged" )
{
...
```

For more details see the `customWidget` presentation style attribute reference.

**Controlling element alignment inside a scrollgrid**

To control the alignment of the elements inside the scrollgrid, use the `itemsAlignment` presentation style:

```
<Style name="ScrollGrid.centered">
  <StyleAttribute name="itemsAlignment" value="center" />
</Style>
```

In the scrollgrid layout definition, use the style name as defined in your styles file:

```
LAYOUT SCROLLGRID ( WANTFIXEDPAGESIZE=NO, STYLE="centered" )
{
...
```

This attribute applies to default scrollgrid rendering and paged scrollgrid rendering. However, the possible values of `itemsAlignment` depend on the type of scrollgrid rendering.

For more details see the `itemsAlignment` presentation style attribute reference.

**Current row / current cell rendering**

In a `SCROLLGRID` container, the highlighting of the current row (or current cell, when focus granularity is at the cell level) can be controlled with style attributes.

For more details, see [Row and cell highlighting in SCROLLGRID](#) on page 1204.
Current row visibility after dialog execution

When the dialog controlling the scrollgrid has finished, the current row may be deselected, depending on the KEEP CURRENT ROW dialog attribute.

Related concepts
SCROLLGRID container on page 1325
Defines a scrollable grid view widget, in a grid-based layout.

Built-in scrollgrid features
Several implicit list handling features are provided by scrollgrid views. Some list features are not supported in scrollgrid.

Unsupported features
Some table / list view features are not supported on scrollgrid container.

The list view features not supported on scrollgrid include:

- Columns layout on page 1862
- List ordering on page 1862

Note: As there are no column headers/titles in scrollgrid list views, column attributes, such as UNSIZABLECOLUMNS, and UNSORTABLECOLUMNS that implement column layout and the built-in list sort, are therefore not supported.

Supported features
The list view provides features that are supported on scrollgrid container.

When a DISPLAY ARRAY or INPUT ARRAY block is combined with a SCROLLGRID, the following list features are supported:

- Find function on page 1863
- Keyboard seek on page 1864
- Reduce filter on page 1865

Defining the action for a row choice
The row choice in a SCROLLGRID can be associated with a dedicated action.

If using a DISPLAY ARRAY dialog to control a scrollgrid, a double-click on a row (for a desktop client), or a tap on a row (for mobile clients) fires a default action.

Row choice action can be customized as described in Defining the action for a row choice on page 1838.

Using scrollgrid on mobile devices
Scrollgrid views render in a specific way on mobile devices, in order to take advantage of mobile device ergonomics.

Unsupported scrollgrid features
Some scrollgrid view features are not supported on mobile devices.

The list view features not supported on mobile devices include:

- Find function on page 1863
- Keyboard seek on page 1864

Related concepts
Built-in scrollgrid features on page 1887
Several implicit list handling features are provided by scrollgrid views. Some list features are not supported in scrollgrid.

Supported scrollgrid features
The list view provides features that are supported on mobile devices.

The list view features supported on mobile devices include:

- The DOUBLECLICK (tap) action on page 1872
**Related concepts**

**Built-in scrollgrid features** on page 1887

Several implicit list handling features are provided by scrollgrid views. Some list features are not supported in scrollgrid.

**Examples**

**SCROLLGRID (scrollgrid views)** usage examples.

**Example 1: Simple scrollgrid view**

The form file `scrollgrid.per`:

```plaintext
LAYOUT
SCROLLGRID (WANTFIXEDPAGESIZE=NO, INITIALPAGESIZE=4, STYLE="paged")
{
    [f1 ] Id: [f2 ] Name:[f3 ]
    [ ] Details:
    [ ] [f4 ]
    [ ] [ ]
    [ ] [ ]
}
END

ATTRIBUTES
EDIT  f2 = FORMONLY.key;
EDIT  f3 = FORMONLY.name;
IMAGE f1 = FORMONLY.image, SIZEPOLICY=FIXED, AUTOSCALE;
EDIT  f4 = FORMONLY.detail;
END

INSTRUCTIONS
SCREEN RECORD list1(FORMONLY.*);
END
```

The presentation style file contains:

```plaintext
<Style name="Scrollgrid.paged">
    <StyleAttribute name="customWidget" value="pagedScrollGrid" />
</Style>
```

The program `main.4gl`:

```plaintext
MAIN
    DEFINE arr DYNAMIC ARRAY OF RECORD
    key INTEGER,
    name STRING,
    image STRING,
    detail STRING
    END RECORD,
    i INTEGER
    FOR i=1 TO 60
        LET arr[i].key = i
        LET arr[i].name = SFMT("Item %1", i)
        IF i MOD 2 THEN
            LET arr[i].image = "file"
        ELSE
            LET arr[i].image = "smiley"
        END IF
        LET arr[i].detail = SFMT("This is item %1", i)
    END FOR
    OPEN FORM f1 FROM "scrollgrid"
    DISPLAY FORM f1
    DISPLAY ARRAY arr TO list1.* ATTRIBUTES(UNBUFFERED,DODBLECLICK=myselect)
    ON ACTION myselect
```
Tree views

Describes how to implement tree views.

Understanding tree-views

This is an introduction to treeview programming.

Tree-views can be implemented with a DISPLAY ARRAY instruction using a form screen-array bound to a TREE container with tree-view specific attributes. TREE containers are very similar to TABLE containers, except that the first columns are used to display a tree of nodes on the right of the widget.

Important: This feature is not supported on mobile platforms (in native rendering mode).

The screenshot shows a typical file browser using a tree-view. This example implements a DIALOG instruction with two DISPLAY ARRAY sub-dialogs. The first DISPLAY ARRAY sub-dialog controls the tree-view while the second one controls the file list on the right side.

Figure 109: Form with Tree View

<table>
<thead>
<tr>
<th>Name</th>
<th>Name</th>
<th>Size</th>
<th>Modification Time</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>usr</code></td>
<td>411ppm</td>
<td>10328</td>
<td>2014-11-07 04:43:59</td>
</tr>
<tr>
<td><code>bin</code></td>
<td>7z</td>
<td>39</td>
<td>2018-02-02 22:35:32</td>
</tr>
<tr>
<td><code>X11</code></td>
<td>7a</td>
<td>40</td>
<td>2018-02-02 22:35:32</td>
</tr>
<tr>
<td><code>etc</code></td>
<td>GET</td>
<td>15059</td>
<td>2014-07-26 15:05:20</td>
</tr>
<tr>
<td><code>games</code></td>
<td>HEAD</td>
<td>15059</td>
<td>2014-07-26 15:05:20</td>
</tr>
<tr>
<td><code>include</code></td>
<td>HTMLLinker</td>
<td>29376</td>
<td>2014-10-26 15:07:28</td>
</tr>
<tr>
<td><code>lib</code></td>
<td>Mail</td>
<td>97360</td>
<td>2015-03-15 11:52:33</td>
</tr>
<tr>
<td><code>lib32</code></td>
<td>POST</td>
<td>15059</td>
<td>2014-07-26 15:05:20</td>
</tr>
<tr>
<td><code>lib64</code></td>
<td>VBoxBalloonCtrl</td>
<td>2176</td>
<td>2016-01-26 11:07:21</td>
</tr>
<tr>
<td><code>libx32</code></td>
<td>VBoxHeadless</td>
<td>2176</td>
<td>2016-01-26 11:07:21</td>
</tr>
</tbody>
</table>

The data used to display tree-view nodes must be provided in a program array and controlled by a DISPLAY ARRAY. It is possible to control a tree view table with a singular DISPLAY ARRAY or with a DISPLAY ARRAY sub-dialog within a DIALOG instruction.

A tree view model is implemented with a flat program array (a list of rows), where each row defines parent/child node identifiers to describe the structure of the tree; so, the order of the rows matters:

<table>
<thead>
<tr>
<th>Tree structure</th>
<th>parent-id</th>
<th>child-id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node 1</td>
<td>NULL</td>
<td>1</td>
</tr>
<tr>
<td>Node 1.1</td>
<td>1</td>
<td>1.1</td>
</tr>
<tr>
<td>Node 1.2</td>
<td>1</td>
<td>1.2</td>
</tr>
<tr>
<td>Node 1.2.1</td>
<td>1.2</td>
<td>1.2.1</td>
</tr>
<tr>
<td>Node 1.2.2</td>
<td>1.2</td>
<td>1.2.2</td>
</tr>
<tr>
<td>Node 1.2.3</td>
<td>1.2</td>
<td>1.2.3</td>
</tr>
<tr>
<td>Node 1.3</td>
<td>1</td>
<td>1.3</td>
</tr>
<tr>
<td>Node 1.3.1</td>
<td>1.3</td>
<td>1.3.1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Depending on your need, you can fill the program array with all rows of the tree before dialog execution, or you can fill or reduce the list of nodes dynamically upon expand / collapse action events. In the second case, you must provide additional information for each row of the program array, to indicate whether the node has children. A dynamic build
of the tree view allows you to implement programs displaying very large trees, for example in a bill of materials application, where thousands of elements can be assembled together.

Tree-views can display additional columns for each node, to show specific row data as in a regular table.

### Figure 110: Tree-view with additional columns

#### Related concepts

- **List dialogs** on page 1831
  Describes how to program dialogs controlling list containers.
- **TREE container** on page 1327
  The TREE container defines the presentation of a list of ordered records in a tree-view widget.
- **Screen records / arrays** on page 1251
  Form fields can be grouped in a screen record or screen array definition.

### Arrays** on page 418
Arrays (static or dynamic) allow you to handle an ordered collection of elements.

#### Defining a TREE container

Start a tree-view implementation by defining the TREE container in the form definition file.

Create a form specification file containing a TREE container bound to a screen array. The screen array identifies the presentation elements to be used by the runtime system to display the tree-view and the additional columns.

A TREE container must be present in the LAYOUT section of the form, defining the columns of the tree-view list. The TREE container must hold at least one column defining the node texts (or names). This column will be used on the front-end side to display the tree-view widget. Additional columns can be added in the TREE container to display node information. The TREE container attributes must be declared in the ATTRIBUTES section of the form.

Secondary form fields have to be used to hold tree node information such as icon image, parent node id, current node id, expanded flag and parent flag. While these secondary fields can be defined as regular form fields and displayed in the tree-view list, we recommend that you use PHANTOM fields instead. Phantom fields can be listed in the screen-array but do not need to be part of the LAYOUT section. Phantom fields will only be used by the runtime system to build the tree of nodes.

Example of tree-view definition using a TREE container:

```plaintext
LAYOUT
TREE mytree ( PARENTIDCOLUMN=parentid, IDCOLUMN=id,
  EXPANDEDCOLUMN=expanded, ISNODECOLUMN=isnode )
{
  Tree
```
Example of tree-view definition using the Tree layout tag inside a GRID container, with a TREE form element to define attributes in the ATTRIBUTES section:

```
LAYOUT
GRID
{
    <Tree tv                        >
    Tree
    [name                      |desc     ]
    [name                      |desc     ]
    [name                      |desc     ]
    [name                      |desc     ]
    [name                      |desc     ]
    <                        >
}
END
END
ATTRIBUTES
TREE tv: mytree,
    PARENTIDCOLUMN=parentid, IDCOLUMN=id,
    EXPANDEDCOLUMN=expanded, ISNODECOLUMN=isnode;
EDIT name = FORMONLY.name, IMAGECOLUMN=image;
PHANTOM FORMONLY.image;
PHANTOM FORMONLY.parentid;
PHANTOM FORMONLY.id;
PHANTOM FORMONLY.expanded;
PHANTOM FORMONLY.isnode;
EDIT desc = FORMONLY.description;
END
INSTRUCTIONS
SCREEN RECORD sr( FORMONLY.* );
END
```

The first visual column ("name" in the example) must be the field defining the node names, and the widget must be an EDIT or LABEL.

Several attributes are used to configure a TREE form element:

- The PARENTIDCOLUMN and IDCOLUMN attributes are respectively used to identify the form field containing the identifiers of the parent and child nodes, defining the structure of the tree. You must specify form field column names, not item tag identifiers (used to reference a form item in the layout section). If these attributes are not specified, the parent node id and node id field names default respectively to "parentid" and "id".
• The **EXPANDEDCOLUMN** attribute can be used to define the form field holding the flag indicating that a node is expanded (opened).

• If the **ISNODECOLUMN** attribute is used, it defines the form field indicating that a node has children, even if the program array does not contain child nodes for that parent node. This attribute must be used to implement dynamic filling of tree-views.

• The **IMAGEEXPANDED**, **IMAGECOLLAPSED** and the **IMAGELEAF** attributes are optional attributes defining global images for expanded, collapsed and leaf nodes. You should use these attributes if you want to display the same icons for all nodes.

• The **IMAGEEXPANDED** and **IMAGECOLLAPSED** instruct the runtime system to set a specific icon when a node gets expanded or collapsed. The **IMAGELEAF** attribute defines the global icon for leaf nodes. This saves the programmer from writing code to display common node images.

Tree-view definition must be completed with form fields declaration. These must be defined in the **ATTRIBUTES** section. The fields not used for display are declared as **PHANTOM** fields. The tree-view form fields must be grouped in a screen-array declared in the **INSTRUCTIONS** section.

The form fields required to declare a tree-view table are the following.

### Table 393: Form fields required to declare a tree-view table

<table>
<thead>
<tr>
<th>Description</th>
<th>Field type</th>
<th>Tree attribute</th>
<th>Mandatory</th>
<th>Default name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text to be displayed for the node</td>
<td>EDIT</td>
<td>N/A</td>
<td>yes</td>
<td>N/A</td>
</tr>
<tr>
<td>Id of the node</td>
<td>PHANTOM</td>
<td>IDCOLUMN</td>
<td>yes</td>
<td>id</td>
</tr>
<tr>
<td>Id of the parent node</td>
<td>PHANTOM</td>
<td>PARENTIDCOLUMN</td>
<td>yes</td>
<td>parentid</td>
</tr>
<tr>
<td>Icon image for a node</td>
<td>PHANTOM</td>
<td>IMAGECOLUMN</td>
<td>no</td>
<td>N/A</td>
</tr>
<tr>
<td>Node expansion indicator</td>
<td>PHANTOM</td>
<td>EXPANDEDCOLUMN</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Parent node indicator</td>
<td>PHANTOM</td>
<td>ISNODECOLUMN</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

The first three fields (node text, parent id and node id) are mandatory, and the first visual (non-phantom) field listed in the screen array will be implicitly used to hold the text of tree-view nodes.

Additional fields (like the "desc" field in this example) can be defined to display details for each node in regular columns, that will appear on the right of the tree widget.

The order of the fields in the screen array of the tree-view does not matter, but it must of course match the order of the corresponding variables in the record-array of the program.

If you need to display node-specific images, define a phantom field to hold node images and attach it to the tree-view definition by using the **IMAGECOLUMN** attribute. Alternatively you can globally define images for all nodes with the **IMAGEEXPANDED**, **IMAGECOLLAPSED** and the **IMAGELEAF** attributes of the **TREE** form element.

**Related concepts**

- **Form fields** on page 1242
  Form fields are form elements designed for data input and/or data display.

- **Item tags** on page 1259
  Item tags define the position and size in a grid-based container.

- **Screen records / arrays** on page 1251
  Form fields can be grouped in a screen record or screen array definition.

- **Arrays** on page 418
Arrays (static or dynamic) allow you to handle an ordered collection of elements.

**Defining the program array for tree-views**

The program array containing the tree-view nodes must use a specific record structure.

In the program code, define a dynamic array of records with the DEFINE instruction. The DISPLAY ARRAY dialog will use that program array as the model for the tree-view list. A tree of nodes will be automatically built based on the data found in the program array. The front-end can then render the tree of nodes in a tree-view widget.

The members of the program array must correspond to the elements of the screen-array bound to the TREE container, by number and data types.

The name of the array members does not matter; the purpose of each member is defined by the name of the corresponding screen-array members declared in the form file. Program array members and screen-array members are bound by position.

The following code example defines a program array with a member structure corresponding to the screen-array defined in the form example of the previous section.

```cl
DEFINE tree_arr DYNAMIC ARRAY OF RECORD
  name STRING,       -- text to be displayed for the node
  pid STRING,        -- id of the parent node
  id STRING,         -- id of the current node
  image STRING,      -- name of the image file for the node (can be null)
  expanded BOOLEAN,  -- node expansion flag (TRUE/FALSE) (optional)
  isnode BOOLEAN,    -- children indicator flag (TRUE/FALSE) (optional)
  description STRING -- user field describing the node
END RECORD
```

The `name`, `pid`, `id` members are mandatory. These hold respectively the node text, parent and current node identifiers that define the structure of the tree.

The `image` member will hold the name of the little icon to be displayed for each node and leaf. You can omit this member, if you do not want to display images, or when the tree defines default images with the IMAGEEXPANDED, IMAGECOLLAPSED and the IMAGELEAF attributes.

The `expanded` member can be used to handle node expansion by program. You can query this member to check whether a node is expanded, or set the value to expand a specific node.

The `isnode` member can be used to indicate whether a given node has children, without filling the array with rows defining the child nodes. This information will be used by front-ends to decorate a node as a parent, even if no children are present. It is recommended that the program then fills the array with child nodes when an expand action is invoked, to implement dynamic tree-views.

The program array can hold more columns (like the "description" field), which can be displayed in regular table columns as part of a node’s data.

Remember the order of the program array members must match the screen-array members in the form file, but this order can be different from the column order used in the layout, with the exception of the first column defining the text of nodes (that is the `name` field in the example).

**Related concepts**

- TREE container on page 1327
- Screen records / arrays on page 1251
- Phantom fields on page 1246

A PHANTOM field defines a screen-record field which is not rendered in the layout (it acts as a hidden field).
Arrays (static or dynamic) allow you to handle an ordered collection of elements.

**Filling the program array with rows**
This topic describes how to fill a program array for a tree-view.

Once the program array is defined based on the screen-array definition of the tree-view table, fill the array with the tree-view definition.

You can directly fill the program array before the dialog execution. Once the dialog has started, you must use the methods `DIALOG.insertNode()`, `DIALOG.appendNode()` and `DIALOG.deleteNode()`, if you want to modify the tree, otherwise information like multi-range selection flags and cell attributes will not be synchronized.

Fill the rows in the correct order defining the structure of the tree, to reflect the parent/child relationship of the tree nodes. If a row defines a tree-view node with a parent identifier that does not exist, or if the child row is inserted under the wrong parent row, the orphan row will become a new node at the root of the tree.

In order to fill the program array with database rows defining the tree structure, you will need to write a recursive function, keeping track of the current level of the nodes to be created for a given parent.

The following example shows how to fill the array with data coming from a database table having the following structure:

```sql
CREATE TABLE dbtree (  
id SERIAL NOT NULL,  
    parentid INTEGER NOT NULL,  
    name VARCHAR(20) NOT NULL  
)
```

The difficulty with fetching a tree from a database table is in the cursor management, which cannot be used recursively. A workaround for this problem is to fetch all the children of a given node at once, then call the function recursively for each of the fetched nodes:

```plaintext
TYPE tree_t RECORD  
    id INTEGER,  
    parentid INTEGER,  
    name VARCHAR(20)  
END RECORD

DEFINE tree_arr tree_t

FUNCTION fetch_tree(pid)  
    DEFINE pid, i, j, n INTEGER  
    DEFINE a DYNAMIC ARRAY OF tree_t  
    DEFINE t tree_t  

    DECLARE cu1 CURSOR FOR SELECT * FROM dbtree WHERE parentid = pid  
    LET n = 0  
    FOREACH cu1 INTO t.*  
        LET n = n + 1  
        LET a[n].* = t.*  
    END FOREACH  

    FOR i = 1 TO n  
        LET j = tree_arr.getLength() + 1  
        LET tree_arr[j].name = a[i].name  
        LET tree_arr[j].id = a[i].id  
        LET tree_arr[j].parentid = a[i].parentid  
        CALL fetch_tree(a[i].id)  
    END FOR

END FUNCTION
```
**Related concepts**

*Arrays* on page 418
Arrays (static or dynamic) allow you to handle an ordered collection of elements.

**Controlling a tree-view with DISPLAY ARRAY**

A DISPLAY ARRAY dialog needs to be used to control a tree-view.

After the program array has been filled, you must execute a DISPLAY ARRAY dialog.

The following code example implements a DISPLAY ARRAY binding the program array called `tree_arr` to the `sr` screen-array, attaching the dialog to the tree table defined in the form:

```plaintext
CALL fill_tree(tree_arr)
DISPLAY ARRAY tree_arr TO sr.* ATTRIBUTES(UNBUFFERED)
  BEFORE ROW
    DISPLAY "Current row is: ", DIALOG.getCurrentRow("sr")
  END DISPLAY
END DISPLAY
```

It is not possible to use the DISPLAY ARRAY paged mode (ON FILL BUFFER) when the decoration is a tree view list. The dialog needs the complete set of open nodes with parent/child relation to handle the tree view display, with the paged mode only a given window of the dataset is known by the dialog. If you use the paged mode in DISPLAY ARRAY with a tree view as decoration, the program will raise an error at runtime.

However, tree-views can be filled dynamically with ON EXPAND/ON COLLAPSE triggers.

**Related concepts**

*Dynamic filling of very large trees* on page 1896
How to optimize the implementation of large tree-views?

**Modifying the tree during dialog execution**

The tree-view content can be changed while executing the DISPLAY ARRAY dialog.

During the DISPLAY ARRAY execution, it is possible to modify the content of the tree model (that is the program array), by inserting, adding or removing nodes by program.

However, it is not recommended to directly modify the program array: you must use the dialog class methods `DIALOG.insertNode()`, `DIALOG.appendNode()`, and `DIALOG.deleteNode()` to modify the tree model instead. By using these methods, the dialog can synchronize internal data, otherwise the tree display would be corrupted.

It is recommended to be in UNBUFFERED mode to get a front-end synchronization of the tree-view content.

**Related concepts**

*The buffered and unbuffered modes* on page 1720
The buffered and unbuffered mode control the synchronization of program variables and form fields.

**Using regular DISPLAY ARRAY control blocks**

Simple table DISPLAY ARRAY control blocks can be used with tree-views.

If needed, you can implement traditional DISPLAY ARRAY control blocks like BEFORE ROW or AFTER ROW:

```plaintext
DISPLAY ARRAY tree_arr TO sr.* ATTRIBUTES(UNBUFFERED)
  BEFORE ROW
    DISPLAY "BEFORE ROW - Current row is: ", DIALOG.getCurrentRow("sr")
  AFTER ROW
    DISPLAY "AFTER ROW - Current row is: ", DIALOG.getCurrentRow("sr")
  END DISPLAY
```

**Related concepts**

DISPLAY ARRAY control blocks on page 1508
DIALOG control blocks on page 1607
**Dialog control blocks** are predefined dialog triggers where you can implement specific code to control the interactive instruction.

**The buffered and unbuffered modes** on page 1720
The buffered and unbuffered mode control the synchronization of program variables and form fields.

**Dynamic filling of very large trees**
How to optimize the implementation of large tree-views?

When a huge tree needs to be displayed, tree data filling can be optimized by creating the nodes on demand. There is no need to fill the complete program array with all possible nodes (down to the last leaf), when only the first levels/branches of the tree are displayed on the screen.

To implement a dynamically filled tree, first define an additional column in the TREE container, to indicate whether a given node has children. That field will be used to render a node with a [+ ] button, so as to allow the end user click on the node to expand it, even if no child nodes are created yet.

In the DISPLAY ARRAY code, if a node is expanded (or collapsed), the dialog will invoke the ON EXPAND or ON COLLAPSE triggers, to let the program add (or remove) rows in the array, and to adapt the tree data dynamically by following navigation events.

```plaintext
DEFINE row_index INTEGER

...;
DISPLAY ARRAY tree_arr TO sr.* ATTRIBUTES(UNBUFFERED)
  ON EXPAND (row_index)
    DISPLAY "EXPAND - Expanded row is: ", row_index
    -- Fill with children nodes for tree_arr[row_index]
  ON COLLAPSE (row_index)
    DISPLAY "COLLAPSE - Collapsed row is: ", row_index
    -- Remove children nodes of tree_arr[row_index]
END DISPLAY
```

The program array can be filled directly before the dialog execution, but once the dialog has started, use dialog methods such as DIALOG.insertNode() to modify the tree, otherwise information like multi-range selection flags and cell attributes will not be synchronized. This is typically the case when implementing a dynamically-filled tree with ON EXPAND/ON COLLAPSE triggers.

**Related concepts**
-/ui.Dialog.appendNode on page 2526
  Appends a new node in the specified tree-view.
- ui.Dialog.deleteNode on page 2529
  Deletes a node from the specified tree-view.

**Example 2: Dynamic tree view (filled on demand)** on page 1899

**Built-in sort and tree-views**
Build-in sort on tree-views can be disabled if needed.

By default, the built-in sort is enabled in a TREE container; when the end user clicks on column headers, the runtime system sorts the visual representation of the program array. Tree nodes are ordered by levels; the children nodes are ordered inside a given parent node.

This is a powerful built-in feature. However, in some cases, the tree structure must be static (the order of the nodes must not change) and you don't want the end user to sort the rows. To prevent the built-in sort, use the UNSORTABLECOLUMNS attribute for the TREE container definition:

```plaintext
LAYOUT
  
END
ATTRIBUTES
TREE tv: mytree, UNSORTABLECOLUMNS, 
```
Related concepts
UNSORTABLECOLUMNS attribute on page 1402
The UNSORTABLECOLUMNS attribute indicates that the columns of the table cannot be selected by the user for sorting.

List ordering on page 1862
List controllers implement a built-in sort. This feature can be disabled if not required.

Multi-row selection and tree-views
Multi-row selection can be used with a DISPLAY ARRAY controlling a TREE container.

Due to the tree-view ergonomic differences with simple tables, the selection of tree nodes follows some specific rules, compared to multiple-row selection in simple tables:

1. When selecting a range of nodes, only visible nodes will get the selection flag. For example, if you select all nodes with Ctrl-A, and if the root node is collapsed, only the root node will be selected. This applies also when selecting nodes by program with the DIALOG.setSelectionRange().
2. Collapsing a node will deselect all child nodes.

Related concepts
Multiple row selection on page 1851
Multiple row selection allows the end user to select several rows within a list of records.

Drag and drop in tree-views
Drag and drop is supported with a DISPLAY ARRAY controlling a TREE.

Use the ON DRAG* and ON DROP interactive blocks, to implement drag and drop within a DISPLAY ARRAY controlling a TREE container.

The nodes can be moved around in the same tree, can be dropped outside the tree or can be inserted in the tree from external sources.

Related concepts
Drag & drop on page 1917
Explains programming techniques for the drag & drop feature.

Examples
TREE (treeview) usage examples.
Example 1: Static tree view (filled before dialog starts)
Form file "form1.per":

```
LAYOUT
GRID
{
  <Tree t1>
    Name               Index
    [c1   c2    ]
    [c1   c2    ]
    [c1   c2    ]
    [c1   c2    ]
  }
END
END

ATTRIBUTES
LABEL c1 = FORMONLY.name;
LABEL c2 = FORMONLY.idx;
PHANTOM FORMONLY.pid;
PHANTOM FORMONLY.id;
```
PHANTOM FORMAL ONLY.exp;

TREE t1: tree1
  IMAGEEXPANDED = "open",
  IMAGECOLLAPSED = "folder",
  IMAGELEAF = "file",
  PARENTIDCOLUMN = pid,
  IDCOLUMN = id,
  EXPANDEDCOLUMN = exp;
END

INSTRUCTIONS
SCREEN RECORD sr_tree(name, pid, id, idx, exp);
END

Static tree DISPLAY ARRAY:

DEFINE tree DYNAMIC ARRAY OF RECORD
  name STRING,
  pid STRING,
  id STRING,
  idx INTEGER,
  expanded BOOLEAN
END RECORD

MAIN
  OPEN FORM f FROM "form1"
  DISPLAY FORM f
  CALL fill(4)
  DISPLAY ARRAY tree TO sr_tree.* ATTRIBUTES(UNBUFFERED)
  BEFORE ROW
    DISPLAY "Current row: ", arr_curr()
  END DISPLAY
END MAIN

FUNCTION fill(max_level)
  DEFINE max_level, p INTEGER
  CALL tree.clear()
  LET p = fill_tree(max_level, 1, 0, NULL)
END FUNCTION

FUNCTION fill_tree(max_level, level, p, pid)
  DEFINE max_level, level INTEGER
  DEFINE p INTEGER
  DEFINE i INTEGER
  DEFINE id, pid STRING
  DEFINE name STRING
  IF level < max_level THEN
    LET name = "Node "
  ELSE
    LET name = "Leaf "
  END IF
  FOR i = 1 TO level
    LET p = p + 1
    IF pid IS NULL THEN
      LET id = i
    ELSE
      LET id = pid || "." || i
    END IF
    LET tree[p].id = id
    LET tree[p].pid = pid
    LET tree[p].idx = p
    LET tree[p].expanded = FALSE
    LET tree[p].name = name || level || "." || i
IF level < max_level THEN
    LET p = fill_tree(max_level, level + 1, p, id)
END IF
END FOR
RETURN p
END FUNCTION

Example 2: Dynamic tree view (filled on demand)

Form file "form1.per":

LAYOUT
GRID
{
  <Tree t1   >
    Name        Description
      [c1   |c2   ]
      [c1   |c2   ]
      [c1   |c2   ]
      [c1   |c2   ]
  }
END
END

ATTRIBUTES
LABEL c1 = FORMONLY.name;
PHANTOM FORMONLY.pid;
PHANTOM FORMONLY.id;
PHANTOM FORMONLY.hasChildren;
LABEL c2 = FORMONLY.descr;
TREE t1: tree1
    IMAGEEXPANDED = "open",
    IMAGECOLLAPSED = "folder",
    IMAGELEAF = "file",
    PARENTIDCOLUMN = pid,
    IDCOLUMN = id,
    ISNODECOLUMN = hasChildren;
END

INSTRUCTIONS
SCREEN RECORD sr_tree(FORMONLY.*);
END

Dynamic tree  DISPLAY ARRAY:

DEFINE tree DYNAMIC ARRAY OF RECORD
    name STRING,
    pid STRING,
    id STRING,
    hasChildren BOOLEAN,
    description STRING
END RECORD

MAIN
    DEFINE id INTEGER
    OPEN FORM f FROM "form1"
    DISPLAY FORM f
    LET tree[1].pid = 0
    LET tree[1].id = 1
    LET tree[1].name = "Root"
    LET tree[1].hasChildren = TRUE
DISPLAY ARRAY tree TO sr_tree.* ATTRIBUTES(UNBUFFERED)
BEFORE DISPLAY
   CALL DIALOG.setSelectionMode("sr_tree",1)
ON EXPAND(id)
   CALL expand(DIALOG,id)
ON COLLAPSE(id)
   CALL collapse(DIALOG,id)
END DISPLAY
END MAIN

FUNCTION collapse(d,p)
   DEFINE d ui.Dialog
   DEFINE p INTEGER
   WHILE p < tree.getLength()
      IF tree[p + 1].pid != tree[p].id THEN EXIT WHILE END IF
      CALL d.deleteNode("sr_tree", p + 1)
   END WHILE
END FUNCTION

FUNCTION expand(d,p)
   DEFINE d ui.Dialog
   DEFINE p INTEGER
   DEFINE id STRING
   DEFINE i, x INTEGER
   FOR i = 1 TO 4
      LET x = d.appendNode("sr_tree", p)
      LET id = tree[p].id || "." || i
      LET tree[x].id = id
      -- tree[x].pid is implicitly set by the appendNode() method...
      LET tree[x].name = "Node " || id
      IF i MOD 2 THEN
         LET tree[x].hasChildren = TRUE
      ELSE
         LET tree[x].hasChildren = FALSE
      END IF
      LET tree[x].description = "This is node " || tree[x].name
   END FOR
END FUNCTION

Split views
These topics describe split view programming in the language.

Understanding split views
Split views refer to the ability to access two forms side by side on a mobile device. This feature is mainly provided for tablet devices, as most phones can only display one window/form at a time.

A split view is composed of a "left pane" and "right pane". In programs, the panes are implemented with window objects displaying forms, which are controlled by parallel dialogs.

Important: This feature is only for mobile platforms.
Split views (controlled by parallel dialogs) are typically used to browse the application data, while modal dialogs are used for data input in a single form. An application based on split views will start with parallel dialogs, and switch to a simple modal dialog when the user chooses to edit application data. Parallel dialog handling is suspended by the runtime system, when a modal dialog executes. For details about parallel dialogs compared to modal dialogs, see What are dialog controllers? on page 1707.

If the application displays several split views simultaneously, implement a navigator pane, to let the end user switch between the different split views.

**Creating split view windows**

The application specifies which window opens in which pane by using the `STYLE` attribute in the `OPEN WINDOW` instruction.

Specify either `LEFT` or `RIGHT` for the `TYPE` attribute, to define a left-hand side pane and a right-hand side pane of the split view respectively.

**Important:** Both left (`TYPE=LEFT`) and right (`TYPE=RIGHT`) split-view windows need to be created.
This example specifies that the window \texttt{w\_left} (with the form \texttt{customer\_list}) display in the left pane, and the window \texttt{w\_right} (with the form \texttt{customer\_detail}) display in the right pane:

\begin{verbatim}
OPEN WINDOW w\_left WITH FORM "customer\_list" ATTRIBUTES(TYPE=LEFT)
OPEN WINDOW w\_right WITH FORM "customer\_detail" ATTRIBUTES(TYPE=RIGHT)
\end{verbatim}

The window content of both panels will be controlled by parallel dialogs.

**Important:**

Split-view windows must be the root window (after closing the default \texttt{SCREEN} window), or direct children of the \texttt{NAVIGATOR} window, if it is used. If regular windows are created before split views, these must be closed:

Case 1: Close regular windows created before the split-views:

\begin{verbatim}
CLOSE WINDOW screen
OPEN WINDOW w1 WITH FORM "form1"
...
CLOSE WINDOW w1
...
OPEN WINDOW w\_left WITH FORM "customer\_list" ATTRIBUTES(TYPE=LEFT)
OPEN WINDOW w\_right WITH FORM "customer\_detail" ATTRIBUTES(TYPE=RIGHT)
...
\end{verbatim}

Case 2: Create split-views as direct \texttt{NAVIGATOR} children

\begin{verbatim}
CLOSE WINDOW screen
OPEN WINDOW w\_main WITH 10 ROWS, 80 COLUMNNS ATTRIBUTES(TYPE=NAVIGATOR)
...
OPEN WINDOW w\_left WITH FORM "customer\_list" ATTRIBUTES(TYPE=LEFT)
OPEN WINDOW w\_right WITH FORM "customer\_detail" ATTRIBUTES(TYPE=RIGHT)
...
\end{verbatim}

When using a navigator window, the names of the split view windows must match the action names created in the parallel dialog controlling the options of the navigator pane. For more details, see \texttt{Navigator} pane on page 1904.

**Parallel dialogs for split views**

In order to control the left-hand and right-hand split view content, you must implement two parallel dialogs, each dedicated to a pane.

Create each window and start the parallel dialog for that window. Repeat for each window. When all windows have been created and all dialogs started, run the event loop to activate them.

\begin{verbatim}
OPEN WINDOW w\_left WITH FORM "customer\_list"
    ATTRIBUTES(TYPE=LEFT)
START DIALOG d\_list\_view
OPEN WINDOW w\_right WITH FORM "customer\_detail"
    ATTRIBUTES(TYPE=RIGHT)
START DIALOG d\_detail\_view
WHILE fgl\_eventLoop()
END WHILE
\end{verbatim}

The parallel dialogs must be implemented with a declarative dialog block. See \texttt{Parallel dialogs (START DIALOG)} on page 1700 for more details.

For small iOS devices (not tablets), consider using the \texttt{ACCESSORYTYPE=DISCLOSUREINDICATOR} in the \texttt{DISPLAY ARRAY} dialog, for left-pane controllers.

**Related concepts**

\texttt{fgl\_eventloop()} on page 2284
Waits for a user interaction event.

**Refreshing a parallel dialog**

To restart a parallel dialog, use TERMINATE DIALOG + START DIALOG.

Once the split view parallel dialogs are started, the typically programming pattern to refresh the detail view of the right pane is to restart the detail dialog by executing a TERMINATE DIALOG followed by a START DIALOG.

The following example shows the case of a list view master (d_list_view dialog) displayed on the left pane, which is bound to a detail view of the right pane (d_detail_view dialog). The detail information must be refreshed when moving to a new row (BEFORE ROW control block):

```dialog
DIALOG d_list_view()
  DISPLAY ARRAY arr TO sr.*
  ATTRIBUTES(ACCESSORIETYPE=DISCLOSUREINDICATOR)
  BEFORE ROW -- in BEFORE ROW, we restart the details view
    CURRENT WINDOW IS w_right
    TERMINATE DIALOG d_detail_view
    LET curr_pa = arr_curr()
    DISPLAY BY NAME arr[curr_pa],* 
    DISPLAY SFMT("tapped row %1",arr_curr()) TO info
    START DIALOG d_detail_view
    CURRENT WINDOW IS w_left
END DISPLAY
END DIALOG
```

**Related concepts**

Syntax of the **START DIALOG instruction** on page 1702

Starts the instance of a declarative dialog.

Syntax of the **TERMINATE DIALOG instruction** on page 1703

Terminates the instance of a declarative dialog.

**One or two panes**

The same application displays as a split view application with two panes on some devices, yet displays as a single pane on other devices. What controls this?

With split views, you open two windows, assigning one to the left pane and one to the right pane of the split view. Not all mobile devices, however, can display multiple panes on the same screen. While the application code is the same, the mobile client displays either one pane (typical for phones) or two panes (typical for tablets).

If the device only allows a single pane to display, the window in the left pane is the first window displayed.

The rules for single-pane or two-pane display differ depending on the mobile platform:

- **On Android™ devices**, the two-pane mode is activated if the width of the screen is more than 900 density-independent pixels (dp). The width of the screen depends on the orientation; you may notice that you have two panes when the tablet is held in landscape mode (width greater than height), yet only one pane when the tablet is held in portrait mode (height greater than width).

  **Note:** A density-independent pixel (dp) is an abstract unit that is based on the physical density of the screen. The unit is relative to a 160 dpi screen, so one dp is one pixel on a 160 dpi screen. The ratio of dp-to-pixel will change with the screen density, but not necessarily in direct proportion.

- **On iOS devices**:
  - With the iPad, the two-pane mode is activated, regardless of the orientation of the tablet.
  - With the iPhone® or iTouch devices, only a single pane displays.
**Switching between panes**

How to switch between the left and right panes of a split view depends on the mobile platform and the ergonomic standards of that platform.

**Switching between panes by program**

After creating the split view windows and starting the parallel dialogs to control them, the application program can switch between the left and right panes of a split view by selecting the corresponding window with the `CURRENT WINDOW IS` instruction.

```plaintext
CURRENT WINDOW IS w_customers
```

**Switching between panes on phone devices**

On a mobile device (such as phones) that only displays one split view pane at the time, switching from the left pane to the right pane is handled automatically by the front-end.

**Note:** The ergonomics and rendering depend on the device's operating system.

When starting the application, the left-pane is displayed first. This pane typically uses a table view controlled by a `DISPLAY ARRAY` dialog.

On an iOS phone, consider using the `ACCESSORYTYPE=DISCLOSUREINDICATOR` in the `DISPLAY ARRAY` dialog of left-pane controllers.

If the end user taps on a row in the list of the left pane, the right pane is automatically shown. To avoid this implicit switch from the left to the right pane, define a `DOUBLECLICK = action-name` attribute in the `DISPLAY ARRAY` dialog, and bind this action to an `ON ACTION` handler which does not change the current window.

Once the right pane is displayed, the user can switch to the left pane:

- On an Android™ phone, press the physical back button.
- On an iOS phone, press the back arrow on the top left of the window.

**Important:** This automatic "back to left panel" option is only possible if the dialog on the right side does not have a `close`, `cancel` or `accept` action defined. If one of these actions are defined, it will be attached to the back button, and that action will be executed when pressed.

**Navigator pane**

A **navigator pane** enables access to several views in an application from a main panel.

For many mobile applications, you will want to provide a view that allows you to show different forms and views that are active at the same time, to expose different functional areas of your application. This can be achieved by providing a top-level navigator with several views, controlled by parallel dialogs.

In order to implement a **top-level navigator**, create a window with the `TYPE=NAVIGATOR` attribute and without a form (using the `x ROWS y COLUMNS` clause). This window will only be used to display a set of actions views, to let the user switch between views. A view can be implemented as a split view by using a left and right typed window.

**Important:** The navigator window must be the root window (after closing the default `SCREEN` window). If regular windows are created before the navigator window, these must be closed:

```plaintext
-- Case 1: Screen window is closed, navigator is the root window
CLOSE WINDOW screen
...
OPEN WINDOW w_main WITH 10 ROWS, 80 COLUMNNS ATTRIBUTES(TYPE=NAVIGATOR)

-- Case 2: Close regular windows created before the navigator window
OPEN WINDOW w1 WITH FORM "form1"
....
CLOSE WINDOW w1
...
OPEN WINDOW w_main WITH 10 ROWS, 80 COLUMNS ATTRIBUTES(TYPE=NAVIGATOR)

- On iOS devices, the navigator window displays in a typical iOS tab bar at the bottom of the screen:

To customize the iOS application, define the color of the iOS tab bar with the `iosTabBarTintColor` and `iosTabBarUnselectedColor` Window-class style attribute.

For example, define a STYLE attribute when creating the window in the program code:

```
OPEN WINDOW tabbar WITH 1 ROWS, 2 COLUMNS
ATTRIBUTES(TYPE=NAVIGATOR, STYLE=mytabbar)
```
Then, in your .4st style definition file, define a global style for the Window elements, and a specific style to define the colors for the tab bar elements:

```xml
<StyleList>
  <Style name="Window">
    <StyleAttribute name="windowType" value="normal" />
    <StyleAttribute name="startMenuPosition" value="menu" />
    <StyleAttribute name="iosTintColor" value="blue" />
    <StyleAttribute name="iosNavigationBarTintColor" value="#00366B" />
    <StyleAttribute name="iosToolBarTintColor" value="#00366B" />
    <StyleAttribute name="iosTabBarTintColor" value="#00366B" />
  </Style>
  <Style name="Window.mytabbar">
    <StyleAttribute name="iosTintColor" value="#ffff00" />
    <StyleAttribute name="iosTabBarUnselectedColor" value="#ff0000" />
  </Style>
</StyleList>
```

• On Android™ devices, the navigator window displays in the top of the screen, in the view control of the action bar (2):

The navigator window will be controlled by a dedicated parallel dialog implementing a MENU instruction, with the action handlers to select the related window, when the corresponding action is fired.

**Important:** The name of the actions in the navigator menu must match the name of the corresponding window, which is typically, the left-typed window when using split views.
The following example implements:

- The `w_main` window, and its corresponding controller, the `d_navigator` parallel dialog.
- The `w_customers` window as a left-pane, with the `d_customers` parallel dialog.
- The `w_orders` window as a right-pane, with the `d_orders` parallel dialog.
- The navigator `MENU` dialog implements the `w_customers` and `w_orders` action handlers.

```plaintext
... OPEN WINDOW w_main WITH 10 ROWS, 80 COLUMNS ATTRIBUTES(TYPE=NAVIGATOR)
START DIALOG d_navigator
OPEN WINDOW w_customers WITH FORM "customers" ATTRIBUTES(TYPE=LEFT)
START DIALOG d_customers
OPEN WINDOW w_orders WITH FORM "orders" ATTRIBUTES(TYPE=RIGHT)
START DIALOG d_orders
...
DIALOG d_navigator()
   MENU
      ON ACTION w_customers ATTRIBUTES(TEXT="Customers", IMAGE="smiley")
         CURRENT WINDOW IS w_customers
      ON ACTION w_orders ATTRIBUTES(TEXT="Orders")
         CURRENT WINDOW IS w_orders
   END MENU
END DIALOG

The functionality is the same on either mobile platforms: providing the user with the ability to navigate between multiple views efficiently. The rendering depends on the platform:

- On an iOS device, navigator window renders as a tab bar, displaying at the bottom of the screen.
- On an Android™ device, navigator window renders as a spinner, which is a drop-down menu in the action bar.

**Rendering an HBox as a splitview**

Achieve a split view display with HBOX container and style attribute.

**Note:** This feature is only available with GMA on Android™ devices, and the GBC front-end.

By defining a TABLE and a GRID container in a parent HBOX container, it is possible to get a split view display by setting the `splitViewRendering` style attribute of the HBox class. When using this style attribute, the TABLE displays as a listview on the left of the screen, while the GRID displays as a form on the right.

First define a form with the HBOX container, TABLE and GRID. In the code example below, the HBOX container gets a STYLE attribute referencing a style named "splitview" in the .4st file:

```plaintext
LAYOUT
HBOX (STYLE="splitview")
   TABLE
   { [c1 | c2 ] [c1 | c2 ] [c1 | c2 ] [c1 | c2 ] }
   END
   GRID
   { First name: [f1 ] Last name: [f2 ] ... }
   END
END ...
```
The .4st file must define the `splitViewRendering` style attribute to "yes":

```xml
<StyleList>
  <Style name="HBox.mystyle">
    <StyleAttribute name="splitViewRendering" value="yes" />
  </Style>
  ...
</StyleList>
```

The program must implement a dialog that handles both parts of the split view. You typically implement a `DISPLAY ARRAY` to handle the `TABLE`, and refresh the right part of the screen contained in the `GRID`, with code in the `BEFORE ROW` control block:

```4GL
DISPLAY ARRAY arr TO sr.*
  BEFORE ROW
      DISPLAY arr[arrCurr()].first_name TO f_first_name
      DISPLAY arr[arrCurr()].last_name TO f_last_name
END DISPLAY
```

**Related concepts**

[HBOX item type](#) on page 1278

Defines a layout area to render child elements in horizontal direction.

**Examples**

Split views usage examples.

**Example 1: Single split view application**

This application uses a minimum amount of code to describe a typical implementation of parallel dialogs that result in a split view application, with a list in the left pane and the detail for the selected row in the right pane. It uses only one split view.

**main.4gl**

The code in the `MAIN` block creates the left pane/window and the right pane/window by specifying the `TYPE` attribute in `OPEN WINDOW`.

The left window will display a form comprised of a table view of all records (`a1_list_view`), the other window contains a form with the detail view of a single record from the array (`a1_detail_view`).

The `START DIALOG` statements, along with the `WHILE fgl_eventLoop()` loop, creates the parallel dialog on which a split view depends.

```4GL
DEFINE arr DYNAMIC ARRAY OF RECORD
  id INTEGER,
  name VARCHAR(15),
  date DATE,
  comment VARCHAR(30)
END RECORD
DEFINE curr_pa SMALLINT

MAIN
  CLOSE WINDOW SCREEN
  CALL populate_array()

  OPEN WINDOW w_left WITH FORM "list_view"
     ATTRIBUTES(TYPE=LEFT)
  START DIALOG d_list_view

  OPEN WINDOW w_right WITH FORM "detail_view"
     ATTRIBUTES(TYPE=RIGHT)
  START DIALOG d_detail_view
```
WHILE fgl_eventLoop()
END WHILE

END MAIN

DIALOG d_list_view()
  DISPLAY ARRAY arr TO sr.*
  ATTRIBUTES(ACCESSORYTYPE=DISCLOSUREINDICATOR)
  BEFORE ROW -- in BEFORE ROW, we restart the details view
  CURRENT WINDOW IS w_right
  TERMINATE DIALOG d_detail_view
  LET curr_pa = arr_curr()
  DISPLAY BY NAME arr[curr_pa].*
  DISPLAY SFMT("tapped row %1",arr_curr()) TO info
  START DIALOG d_detail_view
  CURRENT WINDOW IS w_left
END DISPLAY
END DIALOG

DIALOG d_detail_view()
  MENU
    ON ACTION an_action
      MESSAGE "The action an_action was selected!"
    ON ACTION details
      IF edit_details() THEN
        DISPLAY BY NAME arr[curr_pa].*
      END IF
    END MENU
END DIALOG

FUNCTION edit_details()
  -- A modal dialog disables all parallel dialogs
  OPEN WINDOW w_details WITH FORM "details"
    ATTRIBUTES(TYPE=POPUP, STYLE="popup")
  LET int_flag=FALSE
  INPUT BY NAME
    arr[curr_pa].name,
    arr[curr_pa].comment
    WITHOUT DEFAULTS
  CLOSE WINDOW w_details
  RETURN (int_flag==FALSE)
END FUNCTION

FUNCTION populate_array()
  DEFINE i INT
  FOR i=1 TO 40
    LET arr[i].id=i
    LET arr[i].name="item "||i
    LET arr[i].date=TODAY
    LET arr[i].comment="item-detail "||i
  END FOR
END FUNCTION

Left form definition file (list_view.per)
This form definition file provides the table, or list, of records in the array. Even though four table columns are defined, only two display.

LAYOUT (TEXT="Items")
TABLE
{
Right form definition file (detail_view.per)

This form definition file displays the details for a single record in the array.

LAYOUT (TEXT="Details")
GRID
{
 Id      [f01               ]
 Name    [f02               ]
 Date    [f03               ]
 Comment [f04               ]
 Info    [f05               ]
 [b1_details        ]
}
END
END
ATTRIBUTES
EDIT f01=FORMONLY.id;
EDIT f02=FORMONLY.name, SCROLL;
EDIT f03=FORMONLY.date;
EDIT f04=FORMONLY.comment, SCROLL;
EDIT f05=FORMONLY.info;
BUTTON b1_details:details,TEXT="Modify details";
END

Detail form definition file (details.per)

This is a simple form containing two fields that will be used in the program by the edit_details() function to modify item details.

LAYOUT (TEXT="Edit details")
GRID
{
 Name:    [f01                   ]
 Comment: [f02                   ]
 [                      ]
}
END
END
ATTRIBUTES
EDIT f01=FORMONLY.name, SCROLL;
TEXTEDIT f02=FORMONLY.comment, STRETCH=BOTH;
END
Example 2: Multiple split views with navigation bar

This example shows how to write an application that handles two split views, each having a left and right pane, with a top level navigation pane that allows the end user to easily switch between the two split views.

main.4gl

This module implements the window creation and the parallel dialogs to control their content.

The code in the MAIN block creates four windows:

- The main window is the navigation window/pane, defined by the TYPE=NAVIGATION attribute. Only the d_navigator() main dialog is started.
- Two other windows are created for the customer list and details, in the customers() function. This function is called when the main dialog starts. The function checks if the w_customers window exists and if needed, opens the split-view windows and starts the dialogs handling customer records. If windows already exists, it performs a CURRENT WINDOW IS w_customers, to select the customer pane.
- The second window showing orders and its corresponding dialog are created in the orders() function, using the same programming pattern as in the customers() function.
- When the user selects one of the main dialog actions, it calls either the customers(), the orders(), or the params() function, to show the corresponding pane.
- The configuration pane is handled in the params() function, with the corresponding d_params_menu dialog: When selected, the form is in read-only mode by default. The menu implements the "modify" action to edit the parameters. This action will create a modal dialog, that temporarily stops the parallel dialogs.

```main.4gl
DEFINE c_arr DYNAMIC ARRAY OF RECORD
  id INTEGER,
  name VARCHAR(30),
  address VARCHAR(100)
END RECORD,
c_curr INTEGER

DEFINE o_arr DYNAMIC ARRAY OF RECORD
  id INTEGER,
  info VARCHAR(100),
  deliv DATE
END RECORD

DEFINE params RECORD
  user_name VARCHAR(30),
  auto_sync CHAR(1)
END RECORD

MAIN
  CLOSE WINDOW SCREEN
  OPEN WINDOW w_navigator WITH 10 ROWS, 80 COLUMNS
    ATTRIBUTES(TYPE=NAVIGATOR)
  START DIALOG d_navigator
  WHILE fgl_eventLoop()
  END WHILE
END MAIN

DIALOG d_navigator()
  MENU
    BEFORE MENU
      CALL customers()
    -- Note that action names must match the window names
    ON ACTION w_customers ATTRIBUTES(TEXT="Customers",IMAGE="customers")
      CALL customers()
    ON ACTION w_orders ATTRIBUTES(TEXT="Orders",IMAGE="orders")
      CALL orders()
    ON ACTION w_params ATTRIBUTES(TEXT="Params",IMAGE="sync")
```
CALL params()
END DIALOG

FUNCTION params()
 IF ui.Window.forName("w_params") IS NULL THEN
    OPEN WINDOW w_params WITH FORM "parameters"
    LET params.user_name="Tom"
    LET params.auto_sync="Y"
    DISPLAY BY NAME params.*
    START DIALOG d_params_menu
 END IF
 CURRENT WINDOW IS w_params
END FUNCTION

DIALOG d_params_menu()
 MENU
   ON ACTION modify ATTRIBUTES(TEXT="Modify")
       CALL edit_params()
   ON ACTION options ATTRIBUTES(TEXT="Options")
       CALL options()
 END MENU
END DIALOG

FUNCTION edit_params() -- This is a modal dialog
 LET int_flag=FALSE
 INPUT BY NAME params.* ATTRIBUTES(WITHOUT DEFAULTS)
 IF NOT int_flag THEN
   -- CALL save_params()
 END IF
END FUNCTION

FUNCTION options()
 MENU "Options" ATTRIBUTES(STYLE="dialog")
   ON ACTION sync ATTRIBUTES(TEXT="Synchronize")
   --
   ON ACTION exit ATTRIBUTES(TEXT="Exit")
       EXIT PROGRAM
   ON ACTION cancel
       EXIT MENU
 END MENU
END FUNCTION

FUNCTION customers()
 IF ui.Window.forName("w_customers") IS NULL THEN
    CALL populate_customers()
    OPEN WINDOW w_customers WITH FORM "customer_list"
        ATTRIBUTES(TYPE=LEFT)
    START DIALOG d_customer_list
    OPEN WINDOW w_customer_detail WITH FORM "customer_detail"
        ATTRIBUTES(TYPE=RIGHT)
    END IF
 CURRENT WINDOW IS w_customers
END FUNCTION

DIALOG d_customer_list()
 DISPLAY ARRAY c_arr TO c_sr.*
     ATTRIBUTES(ACCESSORYTYPE=DISCLOSUREINDICATOR)
 BEFORE ROW
    CURRENT WINDOW IS w_customer_detail
    TERMINATE DIALOG d_customer_detail
    LET c_curr = arr_curr()
    DISPLAY BY NAME c_arr[c_curr].*
START DIALOG d_customer_detail
  CURRENT WINDOW IS w_customers
END DISPLAY
END DIALOG

DIALOG d_customer_detail()
  MENU
  ON ACTION details
    LET int_flag=FALSE
    INPUT BY NAME c_arr[c_curr].name, c_arr[c_curr].address
      WITHOUT DEFAULTS
    IF NOT int_flag THEN
      DISPLAY BY NAME c_arr[c_curr].*
    END IF
  END ON
END MENU
END DIALOG

FUNCTION populate_customers()
  LET c_arr[1].id = 324
  LET c_arr[1].name = "Mike Treeman"
  LET c_arr[1].address = "56 Gamleed st."
  LET c_arr[2].id = 8934
  LET c_arr[2].name = "Stepfan Plombier"
  LET c_arr[2].address = "78 Pokam st."
  LET c_arr[3].id = 451
  LET c_arr[3].name = "Ted Barber"
  LET c_arr[3].address = "1243b Western st."
END FUNCTION

FUNCTION orders()
  IF ui.Window.forName("w_orders") IS NULL THEN
    CALL populate_orders()
    OPEN WINDOW w_orders WITH FORM "order_list"
    START DIALOG d_order_list
  END IF
  CURRENT WINDOW IS w_orders
END FUNCTION

DIALOG d_order_list()
  DISPLAY ARRAY o_arr TO o_sr.*
END DISPLAY
END DIALOG

FUNCTION populate_orders()
  LET o_arr[1].id = 43249
  LET o_arr[1].info = "Xmass gifts"
  LET o_arr[1].deliv = MDY(12,23,2011)
  LET o_arr[2].id = 33424
  LET o_arr[2].info = "Dressing items"
  LET o_arr[2].deliv = MDY(2,13,2012)
END FUNCTION

customer_list.per
This is the form defining the customer list, it is used for the left-pane of the customers split view.

LAYOUT (TEXT="Customers")
TABLE
  |
[c1  |c2    ]
END
**customer_detail.per**

This is the form defining fields to show customer details, it is used for the right-pane of the customers split view.

```plaintext
LAYOUT (TEXT="Customer details")
GRID
{
  Id      [f01               ]
  Name    [f02               ]
  Address [f03               ]
  [b1_details        ]
}
END
END
ATTRIBUTES
EDIT f01=FORMONLY.id;
EDIT f02=FORMONLY.name, SCROLL;
EDIT f03=FORMONLY.address, SCROLL;
BUTTON b1_details:details,TEXT="Modify details";
END
```

**order_list.per**

This is the form defining the order list, it is a single form (not a split view)

```plaintext
LAYOUT (TEXT="Orders")
TABLE
{
  [c1    |c2              ]
}
END
END
ATTRIBUTES
PHANTOM FORMONLY.id;
EDIT c1=FORMONLY.info;
EDIT c2=FORMONLY.date;
END
INSTRUCTIONS
SCREEN RECORD o_sr(FORMONLY.*);
END
```

**parameters.per**

```plaintext
LAYOUT (TEXT="Settings")
GRID
{
  User       [f01               ]
  Auto sync  [f02               ]
}
END
```
Example 3: Split view using an HBox

This app uses a minimum amount of code to show a split view implementation using an hbox container.

Styles definition file (mystyles.4st)

For this example, we start with the style file. The style file specifies the `splitViewRendering` attribute for the HBox container when the style is set to `mysplitview`.

```xml
<?xml version="1.0" encoding="ANSI_X3.4-1968"?>
<StyleList>
  <Style name="HBox.mysplitview">
    <StyleAttribute name="splitViewRendering" value="yes"/>
  </Style>
</StyleList>
```

Form definition file (splitview.per)

The form definition file defines a HBOX container using the `mysplitview` style. It contains a TABLE followed by a GRID. The table will become the left pane of the split view app, and the grid will become the right pane of the split view app.

```xml
LAYOUT
  HBOX (STYLE="mysplitview")
  TABLE
  { [c1            |c2                            ]
   [c1            |c2                            ]
   [c1            |c2                            ]
   [c1            |c2                            ]
  }
  END
  GRID
  { <GROUP g1                                      >
    Name:          [lb_name             :lb_id  ]
    E-mail:        [lb_email                    ]
    Address:       [lb_address                  ]
    City:          [lb_city                     ]
    <                >
  <GROUP g2                                      >
    Phone:         [lb_phone                    ]
    Mobile:        [lb_mobile                   ]
    <                >
  }
  END
END

ATTRIBUTES
PHANTOM FORMONLY.id;
EDIT c1 = FORMONLY.name;
EDIT c2 = FORMONLY.address;
PHANTOM FORMONLY.city;
```
PHANTOM FORMONLY.phone;
PHANTOM FORMONLY.mobile;
PHANTOM FORMONLY.email;
GROUP g1: group1, TEXT="Contact";
EDIT lb_id = FORMONLY.cont_id;
EDIT lb_name = FORMONLY.cont_name;
EDIT lb_address = FORMONLY.cont_address;
EDIT lb_city = FORMONLY.cont_city;
EDIT lb_phone = FORMONLY.cont_phone;
EDIT lb_mobile = FORMONLY.cont_mobile;
EDIT lb_email = FORMONLY.cont_email;
END

INSTRUCTIONS
SCREEN RECORD sr(id, name, address, city,
    phone, mobile, email);
END

Application (main.4gl)
The application starts by loading the mystyles.4st style file.
After populating the array with our sample data, the splitview.per form is loaded and displayed in the default SCREEN window.
Then, a DISPLAY ARRAY statement takes control, and fills the fields in the grid in the BEFORE ROW trigger, when a new row is selected by the user.

DEFINE carr DYNAMIC ARRAY OF RECORD
    cont_id INTEGER,
    cont_name VARCHAR(40),
    cont_address VARCHAR(100),
    cont_city VARCHAR(50),
    cont_phone VARCHAR(20),
    cont_mobile VARCHAR(20),
    cont_email VARCHAR(30)
END RECORD

MAIN
    CALL ui.Interface.loadStyles("mystyles")
    CALL load_samples()
    OPEN FORM f FROM "splitview"
    DISPLAY FORM f
    DISPLAY ARRAY carr TO sr.* ATTRIBUTES(UNBUFFERED)
    BEFORE ROW
        DISPLAY BY NAME carr[arr_curr()].*
    END DISPLAY
END MAIN

FUNCTION load_samples()
    DEFINE i INTEGER
    LET i=0
    LET carr[i:=i+1].cont_id = 982
    LET carr[i].cont_name = "Mike Stanford"
    LET carr[i].cont_address = "5 Marbel St."
    LET carr[i].cont_city = "Balmberg"
    LET carr[i].cont_phone = "8723847234"
    LET carr[i].cont_mobile= "8732487833"
    LET carr[i].cont_email = "mikest@xyz.com"
    LET carr[i:=i+1].cont_id = 8234
    LET carr[i].cont_name = "Phil Karlmon"
    LET carr[i].cont_address = "341 Merlo Bld"
Drag & drop

Explains programming techniques for the drag & drop feature.

**Understanding drag & drop**

This is an introduction to drag & drop programming.

Drag & drop is a well-known feature of graphical applications, allowing the end user to use the mouse to drag an element of a window to another window in the same program or into an external application. The front-end platform/device must support this feature.

**Important:** This feature is not supported on mobile platforms.

Drag & drop can be implemented in regular tables and tree-views controlled by a singular **DISPLAY ARRAY** or a **DISPLAY ARRAY** sub-dialog within a **DIALOG** instruction. Drag & drop is not supported in other dialog contexts, such as a singular **INPUT**, **INPUT ARRAY** or **CONSTRUCT**.

With drag & drop, end users can:

- Move drag-able objects between lists and tree-views in the same Genero form or program.
- Move drag-able objects between lists and tree-views in different Genero forms and programs.
- Move drag-able objects between other desktop applications and tables / tree-views in Genero programs.

Drag & drop control is implemented in a **DISPLAY ARRAY** with specific interaction blocks, to handle the events related to the drag and drop operation. These specific blocks will be triggered when drag and drop events arrive from the front-end.

- **ON DRAG_START**
- **ON DRAG_FINISHED**
- **ON DRAG_ENTER**
- **ON DRAG_OVER**
- **ON DROP**

Each of these interaction blocks takes an **ui.DragDrop** object as a parameter. A reference variable to that object must be declared before the dialog. In the interaction block, the **ui.DragDrop** object can be used to configure the drag & drop action to take. For example, a "drag enter" event can be refused.

The **ON DRAG_START** and **ON DRAG_FINISHED** triggers apply to the source of the drag & drop operation; the dialog where the object was dragged. The other triggers provide notification to the drop target dialog, used to inform the program when the different drop events occur and to let the target accept or reject the drop action.

This example illustrates the use of a drag & drop interaction block with the **ui.DragDrop** control object:

```plaintext
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr TO sr.* ...
...
ON DRAG_ENTER(dnd)
    IF ok_to_drop THEN
        CALL dnd.setOperation("move")

END FUNCTION
```
Related concepts
The DragDrop class on page 2570
The ui.DragDrop class is used to control the events related to drag & drop events.

Syntax of drag & drop interaction blocks
The ON DRAG*/ON DROP interaction blocks implement drag & drop operations.

Special drag & drop interaction blocks

1. *dnd-object* is a variable referencing an object of the class ui.DragDrop.

Related concepts
ON DRAG_START block on page 1521
ON DRAG_FINISHED block on page 1522
ON DRAG_ENTER block on page 1522
ON DRAG_OVER block on page 1523
ON DROP block on page 1525

Default drag & drop operation
DISPLAY ARRAY dialogs implement a default drag operation.

The DISPLAY ARRAY dialog provides a default drag operation, that copies all selected rows to the drag & drop buffer, as a tab-separated list of values.

The user code equivalent to the default drag & drop operation would look like this:

```
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr TO sr.* ...
...
  ON DRAG_START(dnd)
      CALL dnd.setOperation("copy")
      CALL dnd.setMimeType("text/plain")
      CALL dnd.setBuffer(DIALOG.selectionToString("sr"))
  ...
END DISPLAY
```

Related concepts
The DragDrop class on page 2570
The ui.DragDrop class is used to control the events related to drag & drop events.

Control block execution order
What is the execution order of drag & drop related dialog control blocks?

The table below shows the order in which the runtime system executes the control blocks related to drag & drop events:
Table 394: Control block execution order for drag & drop events

<table>
<thead>
<tr>
<th>Context / User action</th>
<th>Control block execution order</th>
</tr>
</thead>
<tbody>
<tr>
<td>The user starts to drag an object from the source dialog.</td>
<td>1. ON_DRAG_START (in source dialog)</td>
</tr>
<tr>
<td>The mouse cursor enters the drop target dialog.</td>
<td>1. ON_DRAG_ENTER (in target dialog)</td>
</tr>
<tr>
<td>After entering the target dialog, the mouse cursor moves from row to row, or user chooses to change the drop operation (move or copy).</td>
<td>1. ON_DRAG_OVER (in target dialog)</td>
</tr>
<tr>
<td>The user releases the mouse button over the target dialog.</td>
<td>1. ON_DROP (in target dialog) 2. ON_DRAG_FINISHED (in source dialog)</td>
</tr>
</tbody>
</table>

Related concepts
The DragDrop class on page 2570
The ui.DragDrop class is used to control the events related to drag & drop events.

Handle drag & drop data with MIME types
How to handle MIME types with drag & drop?
If a drag & drop is intended to work only in the same application, data can be passed with variables in the context of the current program. For example, in a program using two tables where the user can drag & drop elements between the two lists, identify the selected rows and update the program arrays accordingly. When drag & drop is limited to the current application, avoid the drop outside the current application.

When a drag & drop operation comes from (or goes to) external applications, data can be of various types/formats: plain text, formatted text, documents, images, sounds, videos, and so on. In order to handle the drag & drop data, you must identify the type of data held in the drag & drop buffer. The type of data in the buffer is identified by the Multiple Internet Mail Extensions (MIME) type. MIME types are a widely used internet standard specification, first introduced to identify the content of e-mail attachments.

Only text data can be passed with drag & drop; binary data is not supported. However, you can pass files by using the fgl_getfile() file transfer function, and identify the file with a URI (text-uri-list MIME type). For a working example, see the demos in $FGLDIR/demo/DragAndDrop.

Note: When using URI MIME types (for file paths for example), the string returned from the getBuffer() method can contain URL-encoded characters such as %5E, which represents the ^ caret. Therefore, you must URL-decode strings returned from getBuffer() with util.Strings.urlDecode(). When setting the drag & drop buffer content, if required by the front-end platform, the string can be URL-encoded using util.Strings.urlEncode() for setBuffer(). However, URL-encoding file paths for setBuffer() is usually not required.

Example of MIME types:

- text/plain
- text/uri-list
- text/x-vcard

You can also define your own MIME type, as long as it does not conflict with existing standard MIME types. For example:

- text/my-remote-file
- text/my-customer-record

If you do not specify a MIME type when the drag starts, the type defaults to text/plain, and the dialog will by default copy the data from selected rows into the drag & drop buffer. To prevent drag & drop to external applications, you must pass an application-specific MIME type to the ui.DragDrop.setMimeType() method, to ensure that other applications do not recognize the MIME type and therefore reject the drop.
Prepared for the dragged object for external targets

If the program implements drag & drop of objects that can be dropped to external programs, you must specify the MIME type of the object and copy the data to the drag & drop buffer, so that the external application can identify the data format and receive it.

In the ON DRAG_START block, you must call the `ui.DragDrop.setMimeType()` method to define the MIME type of the object, and copy the text data into the buffer with the `ui.DragDrop.setBuffer()` method.

This example shows a DISPLAY ARRAY dialog preparing the drag & drop buffer to export VCard data from a dragged row:

```plaintext
DEFINE dnd ui.DragDrop
... DISPLAY ARRAY arr TO sr.* ...
... ON DRAG_START(dnd)
   -- Define the MIME type and copy text data to DnD buffer
   CALL dnd.setMimeType("text/x-vcard")
   CALL dnd.setBuffer( buildVCardData( arr[arr_curr()].cid ) )
   CALL dnd.setOperation("copy")
... END DISPLAY
```

Receiving the dragged object from external sources

This describes how to handle the drop action when the target dialog receives an object dragged from an external source, by identifying the MIME type of the object.

In the ON DRAG_ENTER block, you must call the `ui.DragDrop.selectMimeType()` method to check that data is available in a format identified by the MIME type, passed as a parameter. If the type of data is available in the buffer, the method returns TRUE. Later, when the dragged object is dropped (ON DROP), you can get the previously selected MIME type with `ui.DragDrop.getSelectedMimeType()` before calling `ui.DragDrop.getBuffer()` to retrieve the actual data.

The next example shows the usage of those methods: In ON DRAG_ENTER, the program checks available MIME types, and denies the drop operation if the buffer does not hold any of the MIME types that can be treated by the program. In ON DROP, the program calls `getSelectMimeType()` to check what MIME type was selected, retrieves the data with `getBuffer()` , then inserts a new row and puts the data in dedicated fields depending on the MIME type:

```plaintext
DEFINE dnd ui.DragDrop
... DISPLAY ARRAY arr TO sr.* ...
... ON DRAG_ENTER(dnd)
   -- Set operation to NULL if unexpected MIME type found
   CASE
   WHEN dnd.selectMimeType("text/plain")
   WHEN dnd.selectMimeType("text/uri-list")
   OTHERWISE
      CALL dnd.setOperation(NULL)
   END CASE
... ON DROP(dnd)
   -- Select MIME type and get data from buffer
   LET row = dnd.getLocationRow()
   CALL DIALOG.insertRow("sr", row)
   IF dnd.getSelectMimeType() == "text/plain" THEN
      LET arr[row].text_data = dnd.getBuffer()
   END IF
...```
Related concepts

The DragDrop class on page 2570

The `ui.DragDrop` class is used to control the events related to drag & drop events.

Examples

Drag & Drop usage examples.

**Example 1: Two lists side-by-side with drag & drop**

Form file:

```plaintext
LAYOUT
GRID
{
    <t left>    <t right>
Left      Right
 [a1]      [a2]
 [a1]      [a2]
 [a1]      [a2]
 [a1]      [a2]
 [s1]      [s2]
}
END
END
ATTRIBUTES
a1 = FORMONLY.a1;
a2 = FORMONLY.a2;
s1 = FORMONLY.s1;
s2 = FORMONLY.s2;
END
INSTRUCTIONS
SCREEN RECORD sr_left(a1);
SCREEN RECORD sr_right(a2);
END
```

Program code:

```plaintext
MAIN
    DEFINE drag_index, drop_index, i INT
    DEFINE drag_source, drag_value STRING
    DEFINE arr_left, arr_right DYNAMIC ARRAY OF STRING
    DEFINE dnd ui.DragDrop
    CONSTANT S_LEFT="sr_left"
    CONSTANT S_RIGHT="sr_right"

    OPEN FORM f FROM "dnd"
    DISPLAY FORM f

    FOR i = 1 TO 10
        LET arr_left[i] = "left " || i
        LET arr_right[i] = "right" || i
    END FOR

    INITIALIZE drag_index TO NULL

    DIALOG ATTRIBUTES(UNBUFFERED)
        DISPLAY ARRAY arr_left TO sr_left.*
        ON DRAG_START(dnd)
            LET drag_source = S_LEFT
```
LET drag_index = arr_curr()
LET drag_value = arr_left[drag_index]
ON DRAG_FINISHED(dnd)
    INITIALIZE drag_source TO NULL
ON DRAG_ENTER(dnd)
    IF drag_source IS NULL THEN
        CALL dnd.setOperation(NULL)
    END IF
ON DROP(dnd)
    IF drag_source == S_LEFT THEN
        CALL dnd.dropInternal()
    ELSE
        LET drop_index = dnd.getLocationRow()
        CALL DIALOG.insertRow(S_LEFT, drop_index)
        CALL DIALOG.setCurrentRow(S_LEFT, drop_index)
        LET arr_left[drop_index] = drag_value
        CALL DIALOG.deleteRow(S_RIGHT, drag_index)
    END IF
END IF
END DISPLAY

DISPLAY ARRAY arr_right TO sr_right.*
ON DRAG_START(dnd)
    LET drag_source = S_RIGHT
    LET drag_index = arr_curr()
    LET drag_value = arr_right[drag_index]
ON DRAG_FINISHED(dnd)
    INITIALIZE drag_source TO NULL
ON DRAG_ENTER(dnd)
    IF drag_source IS NULL THEN
        CALL dnd.setOperation(NULL)
    END IF
ON DROP(dnd)
    IF drag_source == S_RIGHT THEN
        CALL dnd.dropInternal()
    ELSE
        LET drop_index = dnd.getLocationRow()
        CALL DIALOG.insertRow(S_RIGHT, drop_index)
        CALL DIALOG.setCurrentRow(S_RIGHT, drop_index)
        LET arr_right[drop_index] = drag_value
        CALL DIALOG.deleteRow(S_LEFT, drag_index)
    END IF
END IF
END DISPLAY

ON ACTION cancel
    EXIT DIALOG
END DIALOG
END MAIN

Web components

This section describes how to use web components in your application.

Understanding web components

External graphical components can be integrated into forms by using the WEBCOMPONENT form item type.

A WEBCOMPONENT form field is a form element that defines an area in the form layout to hold an external component, typically not available as a native widget on the front-end platform.

Important: The purpose of a web component form field is not to replace a web browser, PDF viewer, or to implement sub-forms in HTML. A WEBCOMPONENT field must behave as a single form field and integrate with the other standard Genero form fields and form elements. When implementing a gICAPI webcomponent, the JavaScript
code must keep the focus in the area reserved for the field. For example, the JS code should not create another window or tab with the `window.open()` API.

This section describes how to implement your own web component form fields. Genero provides a set of ready-to-use web components that are available for all front-ends. For more details, see Built-in web components on page 1972.

Web component form fields are used to fulfill a specific display and/or input, with advanced and powerful features which can bring added value to your applications. For example, you can find chart and graph widgets, calendar widgets, drawing widgets, and more. Such specialized widgets are not part of the standard GUI toolkits used by Genero front-ends. They need to be integrated as external components with `WEBCOMPONENT` fields.

**Note:** Depending on the type of front-end, the web components can have limitations: When using native front-ends (GDC, GMA, GMI), the web components are implemented with a "webview" widget, which is not a fully-featured web browser.

The main web component limitations on native front-ends are:

- lack of plugin support,
- less accurate JavaScript engine,
- lack of advanced HTML+CSS features.

Some web components are free, and some are licensed, so it is recommended that you take the cost into account before integrating a new web component in your application.

Web components can be implemented with two different techniques:

1. **Using an URL specification**, by setting the URL as value of the `WEBCOMPONENT` field at runtime. This is the easiest way to implement a web component. The widget is controlled with URL values by the program, but requires some additional coding to handle URLs, instead of flat field values.

2. **Using an gICAPI object** (based on JavaScript), by defining the `COMPONENTTYPE` attribute in the form file. This kind of web component requires some JavaScript coding, to write a form field "plugin", which is usable in a normal dialog instruction, that behaves as all the other widgets in terms of value setting/getting.

The content and/or behavior of a web component can be controlled in the program code by using the field value. To detect events inside the web component, the program dialogs must implement an `ON CHANGE` control block, that will be fired immediately after a user action on the web component.

**Related concepts**

- [ON CHANGE block on page 1492](#)

**WEBCOMPONENT item type**

Defines a specialized form item that holds an external component.

**WEBCOMPONENT item basics**

The `WEBCOMPONENT` form item defines a form field that will hold an external component, implemented with a front-end plug-in mechanism.

This topic describes the `WEBCOMPONENT` item type in form definition files. For more details see the chapter dedicated to web component programming.

**Defining a WEBCOMPONENT**

The `COMPONENTTYPE` attribute identifies gICAPI external objects to be used for the field. The `PROPERTIES` attribute is typically used to define attributes that are specific to a given gICAPI-based web component. For example, a chart component might have properties to define x-axis and y-axis labels. For more details, see Using a gICAPI web component on page 1932.

If the `COMPONENTTYPE` attribute is not used, the web component will be a URL-based web component. For more details, see Using a URL-based web component on page 1928.

Front-ends support different presentation and behavior options, which can be controlled by a `STYLE` attribute. For more details, see Style attributes common to all elements on page 1182.
Where to use a WEBCOMPONENT

A WEBCOMPONENT form item can be defined in different ways:

1. With an item tag and a WEBCOMPONENT item definition on page 1349 in a grid-layout container (GRID, SCROLLGRID and TABLE).
2. As a WEBCOMPONENT stack item on page 1324 in a STACK container.

Built-in Web Components

Genero BDL provides a set of ready-to-use web components, that are deployed by default. For more details, see Built-in web components on page 1972.

Defining the widget size

The size of a WEBCOMPONENT widget can be controlled in grid-based or stack-based layout, based on several attributes such as SIZEPOLICY and STRETCH.

For more details about image sizing, see Controlling the web component layout on page 1924.

Related concepts

SIZEPOLICY attribute on page 1391

The SIZEPOLICY attribute is a sizing directive based on the content of a form item.

Controlling the web component layout

Web components are usually complex widgets displaying detailed information, such as charts, graphs, or calendars. Such widgets are generally resizable. Therefore, the WEBCOMPONENT form item must be large and stretchable.

As web components are displayed in an individual web viewer, several layout concepts need to be considered:

- The layout attributes of the WEBCOMPONENT form item, such as SIZEPOLICY, SCROLLBARS, STRETCH and HEIGHT.
- By default, a WEBCOMPONENT form item is stretchable (STRETCH=BOTH).
- By default, a WEBCOMPONENT form item gets a vertical scrollbar (SCROLLBARS=VERTICAL).
- The layout attributes of the HTML root and body element (viewport, height).
- The layout attributes of responsive elements such as SVG content.

Web component in a grid-based layout

In a grid-based layout, the item tag of the WEBCOMPONENT defines the default dimensions of the web component area:

```
LAYOUT GRID
{
  <GROUP g1           >
  [f1 ] [f2 ]
  [f3 ]
  ...>
  [f5 ]
  [ ]
  [ ]
  }
END
```

In the ATTRIBUTES section, use the SIZEPOLICY, SCROLLBARS, STRETCH and HEIGHT attributes, to define the sizing policy of a web component field:
By default, the WEBCOMPONENT widget gets the size of the form item. This is the behavior of SIZEPOLICY=FIXED, and is the only option. This defines the minimal size of the web component when it is stretchable.

**Note:** SIZEPOLICY=DYNAMIC or SIZEPOLICY=INITIAL are not supported with WEBCOMPONENT fields.

By default, WEBCOMPONENT widget are stretchable in both directions (STRETCH=BOTH). This default behavior can be changed by using another value for the STRETCH attribute. In the next example, the web component will get its initial size from the form item size, and will keep this size even if the window is resized because STRETCH=NONE:

```plaintext
WEBCOMPONENT f5 = FORMONLY.mymap, SIZEPOLICY = FIXED, STRETCH = NONE;
```

**Related concepts**

WEBCOMPONENT item type on page 1293

Defines a specialized form item that holds an external component.

**Web component in stack-based layout**

In a stack-based layout, a WEBCOMPONENT item is defined with other items in a logical presentation order, without any size information:

```plaintext
LAYOUT
  STACK
    GROUP (TEXT="Chart example")
      COMBOBOX FORMONLY.chart_type, NOT NULL,
          INITIALIZER=chart_type_init;
      WEBCOMPONENT FORMONLY.chart, COMPONENTTYPE = "chartjs",
          STYLE="regular";
    END
  END
END
```

By default, the WEBCOMPONENT widget size will adapt to the content of the web component. It will stretch vertically to the appropriate size, in order to show the complete web component content.

To limit the size of the WEBCOMPONENT widget, you can use the HEIGHT attribute in the form definition:

```plaintext
WEBCOMPONENT FORMONLY.chart, HEIGHT = 5, -- 5 lines ...
```

**Note:** If the HEIGHT attribute of the web component is defined in the form file, it fixes the widget height, which may result in vertical scrollbars inside the widget. This is like using SIZEPOLICY=FIXED for a web component in a grid-based layout.

If the HEIGHT attribute is not specified in the .per file, the front-end will take the height attribute of the HTML elements of the web component HTML file into account, for example when using a `<canvas />` element:

```html
<body>
  <canvas id="myChart" height="100px" />
</body>
```

**Related concepts**

WEBCOMPONENT item type on page 1293
Defines a specialized form item that holds an external component.

**Web component HTML layout**

**Stretchable WEBCOMPONENT form field**

In order to get a stretchable HTML content in your web component, start by defining the `WEBCOMPONENT` form field with the following attributes:

```html
WEBCOMPONENT wc1=FORMONLY.wc1,
    COMPONENTTYPE="mywebcomp",
    SIZEPOLICY=FIXED,
    STRETCH=BOTH,
    SCROLLBARS=NONE;
```

**Viewport zooming on mobile devices**

In order to avoid automatic viewport zooming with mobile applications, consider adding a meta tag with `name='viewport'` in the HTML file of your gICAPI-based web components, with initial and maximal scale attributes set to 1:

```html
<meta name='viewport' content='initial-scale=1.0, maximum-scale=1.0' />
```

**Note:** Don't use such responsive meta tag, if your web component isn't specifically designed to be responsive.

**HTML body auto-resize**

In order to force the resizing of the HTML content to the parent container, use the following CSS style:

```html
<style>
  html, body {
    height:100%;
    padding:0;
    margin:0;
    border:0;
  }
  ...
</style>
```

**Controlling SVG layout**

When displaying SVG inside your web component, you need to define how the SVG adapts to the parent container. SVG root elements define their own viewport (`width` and `height` attributes), and local layout rules through `viewBox` and `preserveAspectRatio` attributes.

For general purpose, do not set the `width` and `height` attributes in your root SVG element, and use `preserveAspectRatio="xMidYMid meet"`.

In the CSS style of the HTML page, define CSS styles for a parent div and for the `svg` root element, depending on the needs (to get scrollbars for example):

```html
<style>
  ...

  .svg_container_nsb {
    overflow: hidden;
  }
</style>
```
Control the layout of the svg root element by using the appropriate CSS styles (typically set when building your SVG by program):

SVG adapting to container, no scrollbars in div container (all SVG will be visible):

```html
<body>
  <div id="svg_container" class="svg_container_nsb">
    <svg viewBox="-15 -10 158 143" preserveAspectRatio="xMidYMid meet" ... class="root_svg_max_h" ...
    ...
  </svg>
</div>
</body>
```

SVG adapting to container, with vertical scrollbars in the div container (SVG height is much bigger than the width):

```html
<body>
  <div id="svg_container" class="svg_container_vsb">
    <svg viewBox="-20 -7 268 727" preserveAspectRatio="xMidYMin slice" ...
    ...
  </svg>
</div>
</body>
```

SVG adapting to container, with horizontal scrollbars in the div container (SVG width is much bigger than the height):

```html
<body>
  <div id="svg_container" class="svg_container_hsb">
    <svg viewBox="-20 -7 268 727" preserveAspectRatio="xMidYMin slice" class="root_svg_h_max_h" ...
    ...
  </svg>
</div>
</body>
```
Debugging a web component

**Debugging a Web Component with the GBC front-end**

In order to debug a Web Component when displayed on the GBC front-end, start the debug mode of your browser (for example, in Chrome, hit the F12 key), and use the integrated HTML / JavaScript debugger to inspect the Web Component contents.

**Debugging a Web Component with the GDC front-end**

The WebEngine debugger is a debugging tool that opens in a separate window, providing access to the QT WebEngine Developer Tools. These tools include a variety of tools. One such tool is the inspector, which shows the HTML and JavaScript code when you click on an element. To learn more about tools for Web developers, see Chrome DevTools.

To open the Chrome Developer Tools window, use the [Alt] + [Shift] + [Right Click] shortcut combination over the web component field. This shortcut will work in native mode and universal rendering mode.

**Debugging a Web Component with the GMA front-end**

In order to debug a web component displayed on GMA, you need a Chrome web browser, and an USB cable to plug the device to your computer.

Steps to debug a web component on GMA:

1. Connect your device to your desktop with the USB cable.
2. Stop GMA on the device, if it is already running.
3. Enable debugging option in GMA parameters.
4. Start the application with a web component. This will start GMA with debug service enabled (please wait as the application could take longer to start)
5. Open Chrome on the desktop computer.
6. In the Chrome URL address bar, enter: `chrome://inspect/#devices`
7. On the device, accept the USB debugging.
8. Click on “inspect” link to open the HTML debug window.

**Debugging a Web Component with the GMI front-end**

In order to debug a web component displayed on GMI, you need to setup the Safari® Web Inspector. Check the Apple® development site for more details about enabling the Safari® Web Inspector: http://developer.apple.com.

Steps to debug a web component on GMI:

1. On you Mac, open Safari® and make sure the "Develop" option is available. If this option is not available, go to Safari® preferences and enable the option.
2. On the device, go to Settings. In Safari® preferences, go to Advanced and enable the Web Inspector option.
3. Run the app on the device or simulator, or start the program in development mode to display on the GMI front-end.
4. On the Mac, find the connected device/simulator in the Develop menu of Safari® and browse the debuggable components.

**Using a URL-based web component**

This section describes how to add a *URL-based web component* to your application.

To implement an URL-based web component:

URL-based web components are hosted on a third party server and provide a specific service, such as a geographical location on a map. Your application will be dependent on this external service. Consider verifying its permanent availability.
1. Identify the URL of the hosted web component you want to use.
2. In the form file, define a WEBCOMPONENT field, without a COMPONENTTYPE attribute.
3. In the program, set the URL of the hosted web component in the form field value.
4. In the program, detect user interactions with an ON CHANGE control block, and control the URL-based web component with dedicated front calls.

Detailed information about these tasks are provided in the next topics.

**Defining a URL-based web component in forms**
A URL-based web component needs to be declared in the form definition.

**Adding a WEBCOMPONENT to the form file**
To define a URL-based web component field, add a form field with the WEBCOMPONENT item type, without the COMPONENTTYPE attribute:

```
WEBCOMPONENT f001 = FORMONLY.mymap;
```

A web component field is typically defined with the FORMONLY prefix, as the data for the field is rarely stored in a database column.

The field type (and its corresponding program variable) must be a character string type. Consider using the STRING type to avoid any size limitation for the URL specification.

**Sizing policy for web component fields**
Web components are usually complex widgets displaying detailed information, such as charts, graphs, or calendars, which are generally resizable. Use the appropriate form item attributes to get the expected layout and behavior. For more details, see Controlling the web component layout on page 1924.

**Example**

```
LAYOUT
GRID
{
[wc ]
[
[
[
[
]
]
]
]
}
END
END
ATTRIBUTES
WEBCOMPONENT wc = FORMONLY.mychart,
STRETCH = BOTH;
END
```

**Related concepts**
Example 1: URL-based web component using Google maps on page 1931

**Specifying the URL source of a web component**
The content of URL-based web components is defined by the form field value. It can only be set by program.

**Setting the initial URL**
When the current form defines a WEBCOMPONENT form item without the COMPONENTTYPE attribute, it is a URL-based web component. The program can set the URL dynamically in field value:

```
DISPLAY "wc-URL" TO wc-field
```
or with:

```plaintext
DEFINE wc_field STRING
LET wc_field = "wc-URL"
DISPLAY BY NAME wc_field
```

or by using the variable in an INPUT dialog with the UNBUFFERED option:

```plaintext
DEFINE rec RECORD
    name STRING,
    mymap STRING
END RECORD
...
LET rec.mymap = "http://www.openstreetmap.org"
INPUT BY NAME rec.* WITHOUT DEFAULTS
    ATTRIBUTES(UNBUFFERED)
...
```

Once the URL of the web component is defined, the initial URL content is shown by the front-end, and the end user can interact with it.

### Changing the URL

During program execution, you can assigning another URL to the web component field value. The content will be updated to show the new URL.

This example implements a MENU dialog with actions that set different URLs to the web component field, changing the content based on the selected action:

```plaintext
MENU "test"
    ON ACTION map_1
        DISPLAY "http://www.openstreetmap.org" TO wc_field
    ON ACTION map_2
        DISPLAY "http://www.wikimapia.org" TO wc_field
    ON ACTION map_3
        DISPLAY "http://maps.google.com" TO wc_field
END MENU
```

### Controlling the URL web component in programs

URL-based web components can be controlled with the field value and with front calls.

#### Detecting user interaction in a web component with ON CHANGE

The content of an URL-based web component is defined by the field value.

When the end user interacts with the content, and if the remote service points to a different URL, the field value changes.

The URL change can be detected by implementing an **ON CHANGE** control block for the web component field.

**Important:** Due to security rules in modern web browsers, URL-based web components may not trigger **ON CHANGE** block, if the web component URL contains a domain name different from the Genero application. Thus, any changes/navigation in cross domain location cannot be detected in the program. Only HTML pages in the same domain will trigger **ON CHANGE** blocks.

The **ON CHANGE** trigger will be fired immediately when the URL changes:

```plaintext
DEFINE rec RECORD
    num INTEGER,
    name STRING,
    map STRING
```
**Controlling URL-based web components with front calls**

The web component can be manipulated with specific front calls. The web component-specific front calls are provided in the "webcomponent" front call module.

The `webcomponent.call` is a front call that can be used for general purposes. It takes as parameters the name of the form field, a JavaScript function to call, and optional parameters as required. The JavaScript function must be implemented in the HTML content pointed to by the URL of the web component field. The front call returns the result of the JavaScript function.

```plaintext
DEFINE title STRING
CALL ui.Interface.frontCall("webcomponent", "call",
    ["formonly.url_field", "eval", "document.title"],
    [title])
```

**Note:** When passing a RECORD or DYNAMIC ARRAY as front call parameter, it will be converted to a JSON string for the JavaScript function.

Some web component providers return key information in the title of the HTML document. The `webcomponent.getTitle` function is another useful `webcomponent` front call that can get the title of the HTML document of the web component:

```plaintext
DEFINE info STRING
CALL ui.Interface.frontCall("webcomponent", "getTitle",
    ["formonly.url_field"], [info])
```

**Related concepts**

- [Web component front calls on page 2666](#)

This section describes web component specific front calls.

**Examples**

URL-based Web Components usage examples.

*Example 1: URL-based web component using Google maps*

This example shows how to implement a simple mobile application using a `WEBCOMPONENT` field interacting with Google maps.

The form file: `webcomp.per`

```plaintext
LAYOUT GRID
{
  [f1 ]
  [    ]
  [    ]
  [    ]
  [    ]
  [    ]
  [    ]
  [f2 ]
  [f3 ]
```
The program file: webcomp.4gl

```
MAIN
  CONSTANT c_gmaps = "http://maps.google.com/
  DEFINE rec RECORD
    mymap STRING,  
    location STRING,  
    value STRING
  END RECORD
  OPEN FORM f1 FROM "webcomp"
  DISPLAY FORM f1
  LET rec.location = "Paris"
  LET rec.mymap = c_gmaps||"?q="||rec.location
  INPUT BY NAME rec.* WITHOUT DEFAULTS
  ATTRIBUTES(UNBUFFERED)
    ON ACTION set_loc
      LET rec.mymap = c_gmaps||"?q="||rec.location
      LET rec.value = rec.mymap
    ON CHANGE mymap
      LET rec.value = rec.mymap
  MESSAGE "URL has changed! "||CURRENT HOUR TO FRACTION(3)
END INPUT
END MAIN
```

**Related concepts**

Defining a URL-based web component in forms on page 1929

A URL-based web component needs to be declared in the form definition.

**Using a gICAPI web component**

This section describes how to add a *gICAPI-based web component* to your application.

To implement a gICAPI-based web component:

1. Identify the web component you want to use and get the source code (HTML, JavaScript, CSS).
2. Implement the gICAPI interface script for the web component.
3. Define the location where the front-end can find the gICAPI interface files. This depends on the front-end technology used by your application.
4. Define a `WEBCOMPONENT` field in the form file. Use the `COMPONENTTYPE` attribute to define the root HTML file name describing the gICAPI web component.
5. Use the web component in the dialog of the program.
6. If image resources are required by your web component, you must provide them as part of the gICAPI web component assets, or provide them from the program with a specific API.

Detailed information about these tasks are provided in the next topics.
**HTML document and JavaScript for the gICAPI object**

A gICAPI web component is identified by an HTML document containing the JavaScript interface (or a reference to the .js file).

The HTML document is defined by the `COMPONENTTYPE` attribute of the `WEBCOMPONENT` form field. The name specified in this attribute will be used to identify the HTML file:

```html
WEBCOMPONENT wc = FORMOMLY.chart,
    COMPONENTTYPE = "mychart"; -- Identifies "mychart.html"
```

The HTML document must reference (or contain) the JavaScript implementing the gICAPI interface:

```html
<!DOCTYPE html>
<html>
<head>
    <title>The title</title>
    <script language="JavaScript" type="text/javascript" src="wc_echo.js"></script>
</head>
<body>
    <div style="background-color:green;width:3000px;height:3000px;" > here 
</div>
</body>
</html>
```

**The gICAPI web component interface script**

The gICAPI web components are controlled on the front-end through a gICAPI interface object, defined in a JavaScript script.

**gICAPI interface basics**

The goal of the gICAPI interface is to manage communication between the program and the web component with a basic API, to handle the interaction events, the focus, and the value of the web component field.

The interface script is written in JavaScript and bound to the `WEBCOMPONENT` form field by using an HTML document as container.

**Figure 112: Web Component communication management**

The gICAPI web component API relies on a published global JavaScript object named `gICAPI`.

**gICAPI initialization function**

The `onICHostReady()` global function must be implemented, to execute code after the HTML page has been loaded and the gICAPI interface has been initialized.
The gICAPI object is ready in the context of onICHostReady().

**Important:** The onICHostReady() function is only called for the initial HTML page defined by the WEBCOMPONENT form item. If you implement JavaScript code that loads another HTML page with a set of gICAPI interface functions, these will be ignored. For example, using `window.location='./new-page.html'` will load another HTML page, making all current gICAPI WEBCOMPONENT interaction functions invalid.

**Table 395: Function to handle the gICAPI interface**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>onICHostReady( version String )</td>
<td>Called when the gICAPI web component interface is ready. The version passed in the parameter allows you to check that your component is compatible with the API, and initialization code can be execute in this function.</td>
</tr>
</tbody>
</table>

The programming interface of the gICAPI class is identified by a version number, to make sure that the user code corresponds to the current gICAPI implementation. Verify that the runtime version number matches the gICAPI version used during development, by checking the value passed as parameter to onICHostReady():

```javascript
var onICHostReady = function(version) {
    if ( version != "1.0" ) {
        alert('Invalid API version');
    }
    ...
}
```

The rest of the onICHostReady() function body is used to do some initialization and to assign the gICAPI.on*() callback functions as described later in this topic.

**gICAPI field management functions**

The gICAPI object supports a set of callback functions (like onFlushData()) and control functions (like Action()), to handle field value changes, properties changes and focus requests.

**Important:** The gICAPI object must be instantiated, before defining and assigning these methods. The gICAPI object is created and initialized by the web component framework before calling the onICHostReady() global function. Therefore, on* callback methods are typically defined and assigned to the gICAPI object, inside the body of the onICHostReady() function.

```javascript
var onICHostReady = function(version) {
    if ( version != "1.0" ) {
        alert('Invalid API version');
    }
    current_color = "#000000";
    gICAPI.onProperty = function(properties) {
        ...
    }
    ...
}
```

**Note:** The execution order of the gICAPI.on* functions is undefined. For example, the onData() function may be fired before the onProperty() function when the form is initialized. Consider writing code that takes this behavior into account.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action( action String )</td>
<td>Triggers an action event, which will execute the corresponding ON ACTION code. For more details, see gICAPI.Action() on page 1935.</td>
</tr>
<tr>
<td>onData( data String )</td>
<td>Called when the program value of the form field is updated by the runtime system. For more details, see gICAPI.onData() on page 1937.</td>
</tr>
<tr>
<td>onFlushData( )</td>
<td>Called when the gICAPI framework needs to send a value to the runtime system. For more details, see gICAPI.onFlushData() on page 1938.</td>
</tr>
<tr>
<td>onFocus( polarity Boolean )</td>
<td>Called when the runtime system / program changes the focus. For more details, see gICAPI.onFocus() on page 1939.</td>
</tr>
<tr>
<td>onProperty( properties String )</td>
<td>Called to get WEBCOMPONENT PROPERTIES attributes. For more details, see gICAPI.onProperty() on page 1940.</td>
</tr>
<tr>
<td>onStateChanged( params [] )</td>
<td>Called when the field state changes (for example, when it gets active/inactive). For more details, see gICAPI.onStateChanged() on page 1944.</td>
</tr>
<tr>
<td>SetData( data String )</td>
<td>Registers data to be sent to the program, in order to set the form field value in the program. For more details, see gICAPI.SetData() on page 1945.</td>
</tr>
<tr>
<td>SetFocus()</td>
<td>Generates a focus request. For more details, see gICAPI.SetFocus() on page 1946.</td>
</tr>
</tbody>
</table>

**Related concepts**

- Web component front calls on page 2666
  This section describes web component specific front calls.

- gICAPI.Action()
  The gICAPI.Action() function is used to perform an action in the current dialog.

**Purpose of gICAPI.Action()**

Use the gICAPI.Action("action-name") function in order to execute an action in the context of the current dialog.

This function takes the name of the action as parameter. The corresponding ON ACTION handler will be called.

**Note:** Make sure that the parameter passed to gICAPI.action() matches the action name in lowercase letters.

If the named action is not available (if it does not exist, or if it is disabled), the gICAPI.Action() function has no effect.

After calling gICAPI.Action("action-name"), the gICAPI framework will perform a gICAPI.onFlushData() callback (if defined), to let you provide the web component field value with gICAPI.setData().

However, if you do not use the gICAPI.onFlushData() callback, it is also possible to provide the value with gICAPI.setData(), just before calling with gICAPI.Action().
Example 1: Using `gICAPI.SetData()` directly before `gICAPI.Action()`

The following code executes the "color_selected" action, after setting the value of the `WEBCOMPONENT` field with `gICAPI.SetData()`:

```javascript
var selectColor = function(c) {
    gICAPI.SetData(c);
    gICAPI.Action("color_selected");
}

// In the HTML code, an element defines the onclick handler as follows:
... onclick="selectColor('#FFFFCC')" ...
```

Example 2: Using `gICAPI.Action()` and `gICAPI.onFlushData()`

The following code executes the "color_selected" action, and sets the value of the `WEBCOMPONENT` field with `gICAPI.SetData()` in the `gICAPI.onFlushData()` callback:

```javascript
var color = null;

var selectColor = function(c) {
    color = c;
    gICAPI.Action("color_selected");
}

gICAPI.onFlushData = function() {
    gICAPI.SetData(color);
}

// In the HTML code, an element defines the onclick handler as follows:
... onclick="selectColor('#FFFFCC')" ...
```

Action handler in program code

In the program code, define an `ON ACTION` block in a dialog instruction, to execute code when the corresponding web component action is fired.

**Note:** `WEBCOMPONENT` actions are not known by the front-end. A default action view will be created for each `ON ACTION` handler. Consider using the `DEFAULTVIEW=NO` action attribute, in order to avoid default action views being displayed for your web component actions.

```sql
ON ACTION color_selected ATTRIBUTES( DEFAULTVIEW=NO )
    IF rec.webcomp == "#000000" THEN
        LET rec.webcomp = rec.pgcolor
        LET rec.info = NULL
        ERROR "Black color is denied!"
    ELSE
        LET rec.pgcolor = rec.webcomp
        LET rec.info = "Color selected:", rec.pgcolor
    END IF
```

Related concepts

`gICAPI.SetData()` on page 1945
The `gICAPI.SetData()` function registers `WEBCOMPONENT` field data to be sent to the program.

**gICAPI.onData()**
The `gICAPI.onData()` function is executed when field data is sent by the program.

**Purpose of gICAPI.onData()**
The `gICAPI.onData()` function is called each time the `WEBCOMPONENT` field content modification comes from the program. This occurs for example when the current dialog sets the field value, or when a `DISPLAY value TO wc_field` instruction is performed.

The `gICAPI.onData()` function is also used to check that the runtime system has accepted the web component value change, after a call to the `gICAPI.SetData()` function, when the value needs to be transmitted from the `WEBCOMPONENT` field to the program.

**Handling gICAPI.onData() values**
When the `gICAPI.onData()` function is fired, assign the data value to the web component, or check that the runtime system has validated the value provided with `gICAPI.SetData()`.

The data parameter is a string that contains the field value as provided by the program. It is up to your JavaScript code to interpret the program value to be rendered on the HTML page. For example, the value may just be GPS coordinates, that will be used to display a location on a map. The data is typically serialized as a JSON string.

**Note:** Use `util.JSON` classes to serialize / de-serialize structured data (RECORDs or ARRAYs)

If the `WEBCOMPONENT` field value can be **NULL**, the `onData()` function must check for null values as follows:

```javascript
if (value == null || value.length == 0) {
    // Process null case.
    ...
}
```

**Example**
The following code example defines the `onData()` function to set the content on a textarea element:

```javascript
var onICHostReady = function(version) {
    ...
    gICAPI.onData = function(content) {
        $('textarea#value').val(content);
    }
    ...
};
```

**Related concepts**
`gICAPI.SetData()` on page 1945
The `gICAPI.SetData()` function registers `WEBCOMPONENT` field data to be sent to the program.

`gICAPI.onFlushData()`

The `gICAPI.onFlushData()` function is executed when the front-end must send the field value to the program.

**Purpose of `gICAPI.onFlushData()`**

The `gICAPI.onFlushData()` function is called when the front-end must sync the `WEBCOMPONENT` field content with the program.

This occurs when the `WEBCOMPONENT` field loses the focus, or when calling `gICAPI.Action()`.

**Note:** If the `gICAPI.onFlushData()` function is not implemented, the front-end will use the value set from the last `gICAPI.SetData()` call.

**Sending values with `gICAPI.onFlushData()`**

In the `gICAPI.onFlushData()` function, use `gICAPI.SetData()` to provide the value to be sent to the runtime system.

The `gICAPI.onFlushData()` function is called when the `gICAPI` framework requires a field value synchronization, or after `gICAPI.Action()` is called, to let you provide the value.

After a `gICAPI.onFlushData()`, the value is sent to program for validation. The runtime system can accept or reject the field value change. In order to detect that the runtime has accepted the value, the `gICAPI.onData()` function will be called, with the same value as the value provided by `gICAPI.SetData()`. The web component then receives an indication that the VM has accepted the value change. Note that `gICAPI.onData()` is also fired when the web component value is changed by the program.

**Details about the behavior of `gICAPI.onFlushData()`**

Consider the following facts when implementing `gICAPI.onFlushData()`:

1. If the `WEBCOMPONENT` field is focused, executing an action will trigger `gICAPI.onFlushData()`
2. If the `WEBCOMPONENT` field is focused, and the front-end gives the focus to another element, `gICAPI.onFlushData()` will be executed.
3. Perform a `gICAPI.SetData()` call in `gICAPI.onFlushData()` when needed (for example, if the value of the `WEBCOMPONENT` field has changed and needs to be sent to the program)
4. Do not use other `gICAPI` functions such as `gICAPI.Action()` or `gICAPI.SetFocus()` in the `gICAPI.onFlushData()` function.
5. The code inside `gICAPI.onFlushData()` must be non-blocking and should execute rapidly.

**Example**

The following code example defines the `gICAPI.onFlushData()` function to provide the content on a textarea element:

```javascript
var onICHostReady = function(version) {
    ...
    gICAPI.onFlushData = function() {
        gICAPI.SetData( $('textarea#value').val() );
    }
    ...
};
```

```javascript
```
Related concepts

The `gICAPI.SetData()` function registers `WEBCOMPONENT` field data to be sent to the program.

`gICAPI.onFocus()`

The `gICAPI.onFocus()` function is used to detect if the `WEBCOMPONENT` field lost or acquired focus.

Purpose of `gICAPI.onFocus()`

The `gICAPI.onFocus()` function is called when the runtime system gives/grabs the focus to/from the `WEBCOMPONENT` field.

**Note:** The `gICAPI.onFocus()` function is mandatory to implement a `gICAPI` web component.

Focus gain or loss is defined by the `polarity` parameter passed to the `gICAPI.onFocus()` function:

- When the runtime system gives the focus to the `WEBCOMPONENT` field, the `polarity` parameter is `true`.
- When the runtime system grabs the focus from the `WEBCOMPONENT` field, the `polarity` parameter is `false`.

When `polarity` is `true` and HTML elements of the web component can get the focus, the `gICAPI.onFocus()` function must set the focus to the expected element inside the web component.

This function can also be used to handle an internal flag, to know if the web component has got the focus. If the web component has the focus, you can call the `gICAPI.SetData()` function to provide the current value of the `WEBCOMPONENT` field.

When is `gICAPI.onFocus()` called?

The `gICAPI.onFocus(polarity)` function is called with `polarity=true` in the following cases:

- If the current dialog selects the `WEBCOMPONENT` field, as a normal tabbing candidate,
- If the user code executes an explicit `NEXT FIELD` instruction (or equivalent dialog instruction) to the web component,
- If the web component JavaScript code asked to get the focus with `gICAPI.SetFocus()`, and the program has accepted to give the focus to the `WEBCOMPONENT` field.

The `gICAPI.onFocus(polarity)` function is called with `polarity=false` in the following cases:

- If the current dialog gives the focus to another field, as a normal tabbing candidate,
- If the user code executes an explicit `NEXT FIELD` instruction (or equivalent dialog instruction) to move to another field,

**Note:** The `gICAPI.onFocus()` function will not be called with `polarity=false`, if the web component JavaScript code asked the focus with `gICAPI.SetFocus()`, and the program refused to give the focus.

Example

The following code example defines the `gICAPI.onFocus()` function to set an internal flag and give the focus to the textarea:

```javascript
var has_focus;

var onICHostReady = function(version) {
    ...

    gICAPI.onFocus = function(polarity) {
        has_focus = polarity;
        if (has_focus) {
            $('textarea#value').focus();
        }
    }
}
```
The `gICAPI.SetFocus()` function must be used to request the focus to the runtime system.

The `gICAPI.SetData()` function registers WEBCOMPONENT field data to be sent to the program.

The `gICAPI.onProperty()` function is executed when web component properties change.

### Purpose of gICAPI.onProperty()

If defined, the `gICAPI.onProperty()` function is called to get the list of properties of the `<PropertyDict>` AUI node. This node is created in the .42f form file under the form field node, when using the `PROPERTIES` attribute in a WEBCOMPONENT field.

**Note:** `gICAPI.onProperty()` may also be called when `<PropertyDict>` is changed at runtime, after the form was loaded.

This function gets a single string parameter, that holds all current properties defined for the gICAPI web component.

The property list is provided as a JSON-formatted string.

**Note:** Each time the `gICAPI.onProperty()` function is called, all properties are provided in the JSON parameter.

For each `gICAPI.onProperty()` call, the complete list of existing properties is passed to the function. In order to detect changes, compare the current set of properties with the new set of properties passed to the `gICAPI.onProperty()` function.

### Converting properties JSON string to a JSON object

The `gICAPI.onProperty()` function gets a JSON formatted string, that can be converted to a JSON object with `JSON.parse()`.

A typical `gICAPI.onProperty()` function starts as follows:

```javascript
function properties_object = JSON.parse( properties );
```

### Handling typed property values

In the `<PropertyDict>` nodes of the AUI tree, attribute type information is not provided. Consequently, if the underlying component expects for example properties with JSON booleans (true/false), your boolean properties need to be identified explicitly and converted to JSON booleans.

In the AUI tree, boolean values are set to '0' or '1'. In JSON, a boolean must be `true` or `false`.

For example, if you define the following `PROPERTIES` attribute for the WEBCOMPONENT form field:

```javascript
PROPERTIES = {
    -- numeric
    height = 500,
    -- string
    theme = "modern",
    -- boolean
    menubar = TRUE,
```
```javascript
statusbar = FALSE,
-- array
toolbar = ( "undo redo", "print preview" )
)

The resulting JSON string passed to the onProperties() function will look like this:

```javascript
{
    "height" : "500",
    "theme" : "modern",
    "menubar" : "1",
    "statusbar" : "0",
    "toolbar" : [ "undo redo", "print preview" ]
}
```

Therefore, the JSON string needs to be parsed and some fields must be converted, before it can be used as a JSON object.

In order to replace AUI tree boolean fields to JSON booleans, you can implement a generic function like the following:

```javascript
// Converts the JSON string from onProperty() into a JSON object.
// // The function needs the list of properties that can be booleans.
// // // AUI booleans must be converted as follows:
// // .per source : PROPERTY ( menubar = FALSE )
// // .42f/AUI XML : <Property name="menubar" value="0"/>
// // onProperty : 'menubar' : '0'
// // JS/JSON : "menubar" : false
// var aui_to_json = function(properties, pl_booleans) {
//     var pso = JSON.parse(properties, function (key, value) {
//         if (value && typeof value === 'string') {
//             if ( pl_booleans.indexOf(key) > -1 ) {
//                 // Make a real boolean from the string value
//                 if (value === "1" || value === "true") {
//                     return true;
//                 } else if (value === "0" || value === "false") {
//                     return false;
//                 } else if (value === "1" || value === "true") {
//                     return true;
//                 } else if (value === "0" || value === "false") {
//                     return false;
//                 } else {
//                     return value;
//                 }
//             }
//         }
//         return value;
//     });
//     return pso;
// }
```

The above function can then be used in onProperty(), to get a well formatted JSON object of properties.

**Example 1: Implementing gICAPI.onProperty()**

This code example defines a default set of properties, that will be merged with a set of properties passed to the gICAPI.onProperty() function.

Once the new set of properties is created, it is assigned to the underlying web component object referenced here as "component", by using the resetProperties() function. Here we assume that component.resetProperties() expects a valid Java Script object, with JSON arrays and JSON boolean values.
Note: The aui_to_json() function is described in the previous section.

```javascript
var properties = {
  options: ["fastsearch", "textpattern", "colorpicker"],
  resize: true,
  menubar: true,
  statusbar: false
};

var onICHostReady = function(version) {
  ...

  gICAPI.onProperty = function(ps) {
    var pso = aui_to_json(ps, ["menubar","toolbar","statusbar"]);
    jQuery.extend(properties, pso);
    component.resetProperties(pso);
  }

  ...
};
```

Example 2: Setting WEBCOMPONENT properties at runtime

This example shows BDL code that can be used to change the PropertyDict AUI node, in order to set the properties of a WEBCOMPONENT field at runtime:

```javascript
PRIVATE FUNCTION _get_property_node(fieldname, tc, property)
  DEFINE fieldname STRING,
      tc STRING, -- "P"=Property or "A"=PropertyArray
      property STRING
  DEFINE tagname STRING,
      w ui.Window,
      f ui.Form,
      n_ff, n_wc, n_pd, n_p om.DomNode,
      nl om.NodeList,
      is_new BOOLEAN
  CASE tc
    WHEN "P" LET tagname = "Property"
    WHEN "A" LET tagname = "PropertyArray"
    OTHERWISE
      DISPLAY "ERROR: Invalid node type:", tc
      EXIT PROGRAM 1
  END CASE
  LET w = ui.Window.getCurrent()
  LET f = w.getForm()
  LET n_ff = f.findNode("FormField", fieldname)
  LET n_wc = n_ff.getFirstChild()
  LET n_pd = n_wc.getFirstChild()
  IF n_pd IS NULL THEN
    LET n_pd = n_wc.createChild("PropertyDict")
  END IF
  LET nl = n_pd.selectByPath(SFMT("//%1[@name="%2"]",tagname,property))
  IF nl.getLength() = 1 THEN
    LET n_p = nl.item(1)
    LET is_new = FALSE
  ELSE
    LET n_p=n_pd.createChild(tagname)
    CALL n_p.setAttribute("name", property)
    LET is_new = TRUE
```
PUBLIC FUNCTION setProperty(fieldname, property, value)
  DEFINE fieldname STRING,
      property STRING,
      value STRING
  DEFINE n om.DomNode,
      is_new BOOLEAN
  CALL _get_property_node(fieldname, "P", property)
      RETURNING n, is_new
  IF n IS NULL THEN
    DISPLAY "ERROR: Property node could not be found/created."
    EXIT PROGRAM 1
  END IF
  CALL n.setAttribute("value", value)
END FUNCTION

PUBLIC FUNCTION setPropertyBoolean(fieldname, property, value)
  DEFINE fieldname STRING,
      property STRING,
      value BOOLEAN
  DEFINE n om.DomNode,
      is_new BOOLEAN
  CALL _get_property_node(fieldname, "P", property)
      RETURNING n, is_new
  IF n IS NULL THEN
    DISPLAY "ERROR: Property node could not be found/created."
    EXIT PROGRAM 1
  END IF
  CALL n.setAttribute("value", IIF(value,"1","0"))
END FUNCTION

PUBLIC FUNCTION setPropertyArray(fieldname, property, value)
  DEFINE fieldname STRING,
      property STRING,
      value DYNAMIC ARRAY OF STRING
  DEFINE n, p, e om.DomNode,
      is_new BOOLEAN,
      i INTEGER
  CALL _get_property_node(fieldname, "A", property)
      RETURNING n, is_new
  IF n IS NULL THEN
    DISPLAY "ERROR: PropertyArray node could not be found/created."
    EXIT PROGRAM 1
  END IF
  IF NOT is_new THEN
    -- To rebuild child list, remove prop node, then re-create.
    LET p = n.getParent()
    CALL p.removeChild(n)
    LET n = p.createChild("PropertyArray")
    CALL n.setAttribute("name", property)
  END IF
  FOR i=1 TO value.getLength()
    LET e = n.createChild("Property")
    CALL e.setAttribute("value", value[i])
  END FOR
END FUNCTION

Related concepts
PROPERTIES attribute on page 1386
The PROPERTIES attribute is used to define a list of widget-specific characteristics.

**gICAPI.onStateChanged()**

The gICAPI.onStateChanged() function is executed when the state of the field changes.

**Purpose of gICAPI.onStateChanged()**

The gICAPI.onStateChanged() function is called when the state of the WEBCOMPONENT field changes. This occurs for example when the field is enabled or disabled when starting/stopping a dialog instruction, or when using the DIALOG.setFieldActive() method, or when the current dialog type changes (when a DISPLAY ARRAY executes a nested INPUT for example).

**Implementing gICAPI.onStateChanged()**

When the gICAPI.onStateChanged() function is fired, set up your WEBCOMPONENT field accordingly, following the state parameters passed to the function.

The gICAPI.onStateChanged() function gets a string representing a JSON structure with all parameters set (like the gICAPI.onProperty() function). Convert this string to a JSON object with JSON.parse(), then use the object properties as follows:

1. params.active: The active state of the field (0: field is disabled, 1: field is enabled)
2. params.dialogType: The current dialog type (Input, DisplayArray, InputArray, Construct)

**Example**

This code example defines the gICAPI.onStateChanged() function to set up a textarea element:

```javascript
var onICHostReady = function(version) {
    ...

    gICAPI.onStateChanged = function(ps) {
        var params = JSON.parse(ps);
        if (params.active) {
            $('textarea#value').disabled = false;
        } else {
            $('textarea#value').disabled = true;
        }
        ...
        if (params.dialogType == 'Input') {
            ...
        }
    }
    ...
};
```

**Related concepts**

gICAPI.onFocus() on page 1939
The `gICAPI.onFocus()` function is used to detect if the WEBCOMPONENT field lost or acquired focus.

`gICAPI.SetData()`
The `gICAPI.SetData()` function registers WEBCOMPONENT field data to be sent to the program.

**Purpose of gICAPI.SetData()**
If the content of the WEBCOMPONENT field needs to be transmitted to the program, use the `gICAPI.SetData()` function to register the data to be sent to the runtime system.

**Important:** The WEBCOMPONENT field must be the current field (therefore the field must have the focus), otherwise `gICAPI.SetData()` will be ignored. Use the `gICAPI.onFocus()` callback function to detect if the WEBCOMPONENT field has the focus.

The data must be a string. It is typically serialized as a JSON string.

**Note:** Data is transmitted as plain text. Sending a large amount of data is not recommended.

**When to use gICAPI.SetData()?**
If the `gICAPI.onFlushData()` callback function is used, use the `gICAPI.SetData()` function in this callback function, to provide the WEBCOMPONENT field value. When the WEBCOMPONENT field loses the focus, or when the `gICAPI.Action()` function is called, the gICAPI framework will call the `gICAPI.onFlushData()` function implicitly.

The `gICAPI.SetData()` can also be called outside the context of `gICAPI.onFlushData()`, or when this callback is not implemented, typically before calling `gICAPI.Action()`. If no `gICAPI.onFlushData()` is implemented, the value provided by the last `gICAPI.SetData()` call will be used.

**Handling NULL values**
In order to send a NULL value, call the `gICAPI.SetData()` function with JavaScript `null` as parameter:

```javascript
if (value.length==0) {
  gICAPI.SetData(null);
} else {
  gICAPI.SetData(value);
}
```

**Example**
The following code example registers data to be sent to the runtime system when the `gICAPI.onFlushData()` callback function is invoked:

```javascript
var onICHostReady = function(version) {
  ...
  gICAPI.onFlushData = function(content) {
    gICAPI.SetData( $('textarea#value').val() );
  }
  ...
};
```

**Related concepts**
`gICAPI.onData()` on page 1937
The `gICAPI.onData()` function is executed when field data is sent by the program.
`gICAPI onFocus()` on page 1939
The `gICAPI.onFocus()` function is used to detect if the `WEBCOMPONENT` field lost or acquired focus.

`gICAPI.SetFocus()`
The `gICAPI.SetFocus()` function must be used to request the focus to the runtime system.

**Purpose of `gICAPI.SetFocus()`**
The tabbing order and focus management is controlled by the runtime system and the dialog code (`NEXT FIELD`). If your `WEBCOMPONENT` field receives an event that requests the focus, you must first perform a `gICAPI.SetFocus()` call to determine if the program can put the focus in the field.

**Focus acknowledgment**
The `gICAPI.SetFocus()` function is used in conjunction with the `gICAPI.onFocus()` callback function. If the program can give the focus to the field, `gICAPI.onFocus()` is called with `true` as parameter. The `gICAPI.onFocus()` is not called if one of the following conditions is true:

- the current field cannot release the focus because it does not satisfy constraints (VERIFY, data type conversion, and so on),
- the dialog code logic prevents focus change (`AFTER FIELD ... , NEXT FIELD`, etc).

A good practice is to define an internal flag, to know if your `WEBCOMPONENT` field has gained the focus.

**Example**
The following code example uses the `gICAPI.SetFocus()` function to get the focus from the runtime system:

```javascript
var has_focus;
var onICHostReady = function(version) {
    has_focus = false;
    ...
gICAPI.onFocus = function(polarity) {
    has_focus = polarity;
    if (has_focus) {
        $('textarea#value').focus();
    }
}
$('textarea#log').on("focus", function() {
gICAPI.SetFocus();
});
...
};
```

**Related concepts**
`gICAPI.onFocus()` on page 1939
The gICAPI.onFocus() function is used to detect if the WEBCOMPONENT field lost or acquired focus.

**Deploying the gICAPI web component files**
Deploy web component files to the front-end platform before using gICAPI web components.

**Standard Genero gICAPI Web Components**
The Genero BDL package provides a set of common, ready-to-use web components, that can be used in your application.

The Genero BDL Web Components are located in $FGLDIR/webcomponents and are found by default with any type of front-end configuration.

As these files are part of the FGLDIR installation directory, no deployment is required.

**Deploying the HTML document and the JavaScript gICAPI interface**
The gICAPI web component files (main HTML file, additional JavaScript files and other potential assets) must be available on the platform where the front-end executes. Depending on your configuration, Genero supports several solutions to provide the gICAPI web component files from a single location. In a distributed configuration with many individual front-end nodes, consider centralizing the gICAPI files on a server, instead of copying the gICAPI web component files manually to each front-end device.

Important: If the main gICAPI HTML document references external JavaScript files, put these files in the same directory as the HTML file referencing them.

**Recommended web component directory layout**
When using the default settings in any configuration (for example, no FGLIMAGEPATH is defined, default GAS settings), put the gICAPI web component files under a webcomponents directory, along with the other program files, for example:

```
appdir
appdir/main.42m
appdir/form1.42f
appdir/form2.42f
appdir/webcomponents/3DChart
appdir/webcomponents/3DChart/3DChart.html
appdir/webcomponents/3DChart/3DChart.js
appdir/webcomponents/3DChart/3DChart.css
appdir/webcomponents/3DChart/icon_close.png
...
```

**Deployment methods for front-end types**
Web components assets are deployed with different solutions, depending on the front-end configuration type:

- Deploying gICAPI assets in direct mode on page 1948
- Deploying gICAPI assets with GAS on page 1948
- Deploying gICAPI assets on mobile on page 1949

Note: The GDC front-end supports also local gICAPI file lookup in the GDC installation directory. However, this solution is deprecated. Consider centralizing the gICAPI web component files on the application server, as described above in the dedicated topics.

**Defining the gICAPI files search path by program**
With older versions, it was possible to use the setWebComponentPath on page 2665 front call, to define by program the base URL to the web component files.

Important: This front call is deprecated, consider using one of the other mechanisms described in this topic.
Related concepts

Using image resources with the gICAPI web component on page 1952
This section explains how to use image resources in a gICAPI web component.

Deploying gICAPI assets in direct mode
Using GDC, GMI or GMA front-ends in direct mode (not through GAS)

When using a front-end with a direct connection (not through the GAS), web component files can be automatically transferred to the front-end.

Note: Providing gICAPI web component files with the direct mode mechanism simplifies the development process for mobile applications, as you do not have to copy the files to the device.

In direct mode, gICAPI web component assets are typically located in the directory where the Main program module resides, under a webcomponents sub-directory. If web component files are located in a different base directory, add the search path to the FGLIMAGEPATH on page 276 environment variable. However, if your application is intended for different front-ends, consider using the recommended gICAPI web component directory layout, to avoid FGLIMAGEPATH for web component assets.

Genero BDL provides a set of standard web components in $FGLDIR/webcomponents. These standard web component assets will be implicitly found.

The gICAPI web component files are searched in the following directories:

1. $FGLDIR/webcomponents/component-type/component-type.html
2. appdir/webcomponents/component-type/component-type.html
3. fglimagepath-dir/webcomponents/component-type/component-type.html
4. fglimagepath-dir/component-type.html

Where:

• $FGLDIR is the runtime installation directory on the application server.
• appdir is the directory where the application program resides.
• fglimagepath-dir is one of the base directories defined in FGLIMAGEPATH.
• component-type is the name defined by the COMPONENTTYPE attribute in the form definition file.

If assets such as .js, .css, .png files are referenced by a relative path name in the HTML content, the resources are also transferred via the direct-mode mechanism. If the assets use an absolute path with a concrete URL scheme (such as http://something), the HTML viewer will try to get the resource from the URL location.

For example, if you define the gICAPI web component field as follows:

```webcomponent wc = FORMOMLY.mychart,
COMPONENTTYPE = "3DChart";```

If the FGLIMAGEPATH search path contains "/opt/myapp", and the gICAPI files are located under "/opt/myapp/webcomponents/3DChart", the gICAPI web component HTML document will be found on the server at:

• /opt/myapp/webcomponents/3DChart/3DChart.html

Note: For backward compatibility, the GDC front-end is able to find web components locally on the workstation where it executes. However, this solution is deprecated. Consider centralizing your gICAPI web components on the application server.

Deploying gICAPI assets with GAS
Using GDC, GBC front-ends through the GAS

When using the Genero Application Server, the gICAPI web component files must be deployed as part of the application program files.

Note: To simplify deployment of gICAPI web components with the GAS, consider using the fglgar utility. For more details, see Packaging web applications on page 2183.
The .xcf configuration file of your application can define the base path to search for HTML web component files. This base path is defined by the WEB_COMPONENT_DIRECTORY entry of the EXECUTION element:

```xml
<APPLICATION ...
<EXECUTION>
  ...
  <WEB_COMPONENT_DIRECTORY>$(res.fgldir)/webcomponents;$(application.path)/webcomponents</WEB_COMPONENT_DIRECTORY>
  ...
</EXECUTION>
</APPLICATION>
```

While specific web component deployment directories can be defined in the .xcf file, consider using the recommended gICAPI web component directory layout.

Genero BDL provides a set of standard web components in $FGLDIR/webcomponents, that can be found with the default as.xcf settings. If you want to use a standard Genero web component with the GAS, and you have defined a specific WEB_COMPONENT_DIRECTORY entry in your application .xcf configuration file, you must add $(res.fgldir)/webcomponents to the WEB_COMPONENT_DIRECTORY entry.

The HTML document must be located in a sub-directory below the base path, using the same name as defined by the COMPONENTTYPE attribute. As result, the complete path to the HTML document will be:

```
base-path/component-type/component-type.html
```

For example, if the form file defines the COMPONENTTYPE attribute as follows:

```xml
WEBCOMPONENT wc = FORMOMLY.mychart,
  COMPONENTTYPE = "3DChart";
```

If application.path is "/opt/var/gas/appdata/app/myapp", the HTML document will be found in:

- /opt/var/gas/appdata/app/myapp/webcomponents/3DChart/3DChart.html

**Note:** The above .xcf example shows the default value of the WEB_COMPONENT_DIRECTORY parameter that can be inherited by all application configuration nodes. If your gICAPI web component files are located under appdir/webcomponents, or if your programs use one of the standard Genero web components provided in $FGLDIR/webcomponents, there is no need to set the WEB_COMPONENT_DIRECTORY element in the .xcf file.

**Related concepts**

**Introducing the GAS and JGAS** on page 2183
The Genero Application Server (GAS) is an engine that plugs in to a Web server for the purpose of delivering Genero Web applications and services. The Genero Application Server for Java (JGAS) is designed to run your applications on the Java EE servlet. A general knowledge of how they operate can be helpful in testing and deploying Web applications.

**Deploying gICAPI assets on mobile**
Using GMI and GMA front-ends, executing app on mobile device

When running the application on mobile (in embedded mode), the gICAPI web component files (along with other assets) must be deployed on the device: The files will be found locally on the device.

**Note:** In mobile development mode (in direct mode, when the application runs on a computer and forms display on the GMA or GMI on the mobile device), the gICAPI web component assets are transferred to the device with the direct-mode solution. Web component assets must be located on the computer when the program executes.

If your application is intended for different front-ends, consider using the recommended gICAPI web component directory layout.

Genero BDL provides a set of standard web components in $FGLDIR/webcomponents. When executing on a mobile device, the standard web component assets will be implicitly found.

In embedded mode, mobile front-ends make a local search for gICAPI web component files in the following order:

1. $FGLDIR/webcomponents/component-type/component-type.html
2. `appdir/webcomponents/component-type/component-type.html`
3. `appdir/component-type.html`

Where:
- `$FGLDIR` is the runtime system directory inside the deployed app.
- `appdir` is the application directory where program files are located.
- `component-type` is the name defined by the `COMPONENTTYPE` attribute in the form definition file.

For example, if you define a custom gICAPI web component field as follows:

```java
WEBCOMPONENT wc = FORMONLY.mychart,
    COMPONENTTYPE = "3DChart";
```

The gICAPI web component HTML document will be found on the mobile device at:
- `appdir/webcomponents/3DChart/3DChart.html`
- `appdir/3DChart.html`

However, using the second location is not recommended (always use a `webcomponents` directory).

For more details about `appdir` on mobile devices, see Deploying mobile apps on Android devices on page 3589 and Deploying mobile apps on iOS devices on page 3604.

### Defining a gICAPI web component in forms

When defining a gICAPI web component in a form specification file, you can also provide a sizing policy and define additional properties.

### Adding a WEBCOMPONENT to the form file

To define an gICAPI web component field, add a form field with the `WEBCOMPONENT` item type and the `COMPONENTTYPE` attribute. The `COMPONENTTYPE` attribute is mandatory when defining a gICAPI web component; it defines the root HTML file name describing the gICAPI web component.

A web component field is typically defined with the `FORMONLY` prefix, as the data for the field is rarely stored in a database column.

### Sizing policy for web component fields

Web components are usually complex widgets displaying detailed information, such as charts, graphs, or calendars, which are generally resizable. Use the appropriate form item attributes to get the expected layout and behavior. For more details, see Controlling the web component layout on page 1924.

### Defining gICAPI web component properties

Since web component field definitions are generic, you must use the `PROPERTIES` attribute to set specific parameters for the component.

The `PROPERTIES` attribute can define a list of:
- simple properties `(name = value),`
- array properties `(name = ( value1, value2, ... ))`
- map/dictionary properties `(name=( name1=value1,name2=value2, ... ))`

where `name` is a simple identifier, and where `values` can be numeric or string literals.

Component properties defined in the `PROPERTIES` attribute are transmitted to the web component through the `onProperty()` method of the gICAPI object.

The name of a property defined in the `PROPERTIES` attribute is converted to lowercase by the form compiler. To avoid mistakes, a good programming pattern is to define properties in lowercase, in both the interface script and in the
form definition file. Property names are not checked at compile time, so nonexistent or mistyped properties will be ignored at runtime.

Example

```plaintext
LAYOUT
GRID
{
  [wc ]
  [     ]
  [     ]
  [     ]
  [     ]
}
END
END

ATTRIBUTES
WEBCOMPONENT wc = FORMONLY.mychart,
  COMPONENTTYPE = "3DCharts",
  STRETCH = BOTH,
  PROPERTIES = ( type = "bars",
                x_label = "Months",
                y_label = "Sales" );
END
```

Related concepts

WEBCOMPONENT item type on page 1293
Defines a specialized form item that holds an external component.

Controlling the gICAPI web component in programs

Controlling the gICAPI-based web components with ON ACTION

Once a WEBCOMPONENT field is defined in the form file with the COMPONENTTYPE attribute pointing to an HTML content file, it can be used as a regular edit field in program dialogs. The data of the gICAPI web component is transmitted with the field value, and usually needs to be serialized and deserialized (typically in JSON), when the data is not a simple scalar value.

When the web component field value is changed in the program, the `onData()` method of the gICAPI object is fired, and you can parse the serialized string in your JavaScript.

In order to detect web component value changes in the program, you need to combine the `gICAPI.setData()` and `gICAPI.Action()` methods, to transmit the value and fire an action, that will be handled by an ON ACTION block.

**Note:** The ON CHANGE trigger is not executed automatically for gICAPI-based web components, just by using `gICAPI.SetData()`.

This example serializes and deserializes a dynamic array using the JSON format:

```plaintext
IMPORT util ...
DEFINE mywc STRING
DEFINE data_array DYNAMIC ARRAY OF RECORD ...
...
INPUT BY NAME mywc, ...
  ATTRIBUTES(WITHOUT DEFAULTS, UNBUFFERED)
  ...
  ON ACTION set_wc_values -- Bound to form button
    LET mywc = util.JSON.stringify( data_array )
...
ON ACTION wc_data_changed -- Triggered by gICAPI.Action()
```
CALL util.JSON.parse( mywc, data_array )

**Important:** All data will be transmitted through the abstract user interface protocol. Transmitting a lot of data will not be efficient and is likely to slow down your application.

### Controlling the glICAPI-based web components with properties

Use the `PROPERTIES` attribute in the form specification, to define the configuration of the field. When a property of the web component is modified, the `onProperty()` method of the glICAPI object in the JavaScript will be invoked with the list of properties in JSON notation. Note that the complete property set will be passed, even if a single property is modified.

### Controlling glICAPI-based web components with front calls

The web component can be manipulated with specific front calls. The web component-specific front calls are provided in the "webcomponent" front call module.

**Important:** Use the `webcomponent.call` front call only for specific needs: With desktop and web front ends, a front call implies a network roundtrip and abstract user interface update that may cause unwanted delays. Prefer using glICAPI interface triggers such as `onData`/`onProperty` instead.

The `call` front call can be used for general purposes. It takes as parameters the name of the form field, a JavaScript function to call, and optional parameters as required. The JavaScript function must be implemented in the HTML content of the glICAPI web component field. The front call returns the result of the JavaScript function.

```plaintext
DEFINE title STRING
CALL ui.Interface.frontCall("webcomponent", "call",
   ["formonly.mychart", "eval", "document.title"],
   [title] )
```

The `getTitle` function is another useful `webcomponent` front call that can get the title of the HTML document of the web component:

```plaintext
DEFINE info STRING
CALL ui.Interface.frontCall("webcomponent", "getTitle",
   ["formonly.url_field"], [info] )
```

Some providers return key information in the title of the HTML document.

### Related concepts

**Defining a glICAPI web component in forms** on page 1950

When defining a glICAPI web component in a form specification file, you can also provide a sizing policy and define additional properties.

**getTitle** on page 2668

Returns the title of the HTML doc rendered by a web component.

### Using image resources with the glICAPI web component

This section explains how to use image resources in a glICAPI web component.

### Image resources in glICAPI web components

In some cases, web components require image resources, which can be classified as follows:

1. Common (static) image resources, that are part of the glICAPI web component implementation. This category of image resource can be referenced with absolute URLs (retrieved automatically by the HTML viewer), or can be deployed as part of the glICAPI web component assets, when referenced with relative URLs.
2. Private (variable) image resources, that are displayed by the program at runtime. This category of image resource can be referenced with absolute URLs (retrieved automatically by the HTML viewer), or can be provided by using the `ui.Interface.filenameToURI()` / direct-mode mechanism (as described below).

**Referencing image resources in HTML**

Image resources are typically referenced in HTML within the `<img/>` element, by setting the `src` attribute to a relative or absolute URL:

The following example uses an absolute URL:

```html
<img src="http://www.4js.com/images/smiley.gif" alt="Smiley face" height="42" width="42" />
```

This example uses a relative URL:

```html
<img src="smiley.gif" alt="Smiley face" height="42" width="42" />
```

The gICAPI web component framework can automatically retrieve image resources. If the value is not an absolute or relative URL that can be resolved by the HTML viewer, the image resources are retrieved from the Genero application using the `ui.Interface.filenameToURI()` / direct-mode mechanism.

**Providing static images in gICAPI web component files**

To provide common static images as assets of your gICAPI web component, provide the image files along with the main HTML file, typically in a dedicated directory. For example, if you define the following directory structure:

```
3DChart/3DChart.html
3DChart/images/redraw.gif
3DChart/images/fetchdata.gif
```

The HTML content of the web component can reference common static images as follows:

```html
<img src="images/redraw.gif" alt="Smiley face" height="42" width="42" />
```

**Providing application images from Genero programs**

Some gICAPI web components display variable image resources provided at runtime. For example, a photo gallery web component displaying pictures. Such image resources are usually private to the application.

To use image resources that are not static images as part of the gICAPI web component assets:

1. Reference absolute URLs directly in the HTML content (in "src" attributes of image elements) with `http://`, `https://` or `file:` themes, to be retrieved automatically by the HTML viewer, or:

2. Reference image resources in the HTML content with the URI returned from the `ui.Interface.filenameToURI()` method, to provide image files from the platform where the application executes (can be a server or mobile device):

   - When running the application on a server behind the GAS, the `filenameToURI()` method will convert the local file path to a URL that will make the image file available through the GAS.
   - When using a direct connection to the front-end (typical GDC desktop configuration with application running on a server), the file name will be returned as is and the images will then be transmitted through the direct-mode mechanism (using FGLIMAGEPATH), as described in Providing the image resource on page 1149.
   - When running apps on mobile devices, the `filenameToURI()` method will build the complete path to the local file, based on the list of directories defined in the `FGLIMAGEPATH` environment variable. The image resource is then directly read from the device file system.

Try Example 3: Application images on page 1960, to see this method in practice.
Related concepts
Static images on page 1153
Describes how to decorate forms with icons.

Runtime images on page 1154
Explains how to display pictures at runtime.

Examples
GICAPI Web Component usage examples.
Example 1: Calling a JavaScript function

This example shows how to call a JavaScript function with the `webcomponent.call` front call

The form file: `wc_echo.per`

```
LAYOUT
GRID
{
  [wc1                                    ]
  [                                       ]
  [                                       ]
  Info: [f1                               ]
}
END
END

ATTRIBUTES
WEBCOMPONENT wc1 = FORMONLY.mywebcomp,
  COMPONENTTYPE="wc_echo", STRETCH=BOTH;
EDIT f1 = FORMONLY.result;
END
```

The HTML file: `wc_echo.html`

```
<!DOCTYPE html>
<html>
<head>
  <title>The title</title>
  <script language="JavaScript" type="text/javascript" src="wc_echo.js"></script>
</head>
<body>
  <div style="background-color:green;width:3000px;height:3000px;" > here </div>
</body>
</html>
```

The JavaScript file: `wc_echo.js`

```
var echoString = function(str) {
  return str;
}

var echoObject = function(ostr) {
  var o = JSON.parse(ostr);
  return JSON.stringify(o);
}

var onICHostReady = function(version) {
  if ( version != 1.0 ) {
    alert('Invalid API version');
  }
```
The program file: wc_echo.4gl

```plaintext
IMPORT util

MAIN
    DEFINE rec RECORD
        mywebcomp STRING,
        result STRING
    END RECORD

    OPEN FORM f FROM "wc_echo"
    DISPLAY FORM f

    INPUT BY NAME rec.* ATTRIBUTES(UNBUFFERED)
        ON ACTION get_title
            CALL ui.Interface.frontCall("webcomponent","getTitle",
                ["formonly.mywebcomp"],[rec.result])
        ON ACTION echo_string
            LET rec.result = wc_call("formonly.mywebcomp", "echoString",
                "Hello!")
        ON ACTION echo_object
            LET rec.result = wc_call("formonly.mywebcomp", "echoObject",
                '{"id":"235","name":"Doorman"}')
    END INPUT

END MAIN

FUNCTION wc_call(fn,fc,p1)
    DEFINE fn,fc,p1 STRING
    DEFINE res STRING
    TRY
        CALL ui.Interface.frontCall("webcomponent","call",[fn,fc,p1],[res])
    CATCH
        ERROR err_get(status)
        RETURN NULL
    END TRY
    RETURN res
END FUNCTION
```

Example 2: Simple text input

Introduction
This topic describes the different steps to implement a simple gICAPI-based web component.

In this example, we will implement a simple text editor based on a textarea HTML element.

The dialog code implements a couple of triggers to show how the WEBCOMPONENT field interacts with the program.

The HTML file is described in detail, and complete code example with program and form file is available at the end of this topic.

HTML code description
As any HTML source code, the file starts with the typical HTML tags:

```html
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="content-type" content="text/html" charset="utf-8" />
```
Note: The "viewport" meta is provided to adjust the viewport size for mobile devices.

A bunch of CSS is added to

```html
<style>
html, body {
  height: 100%;
  padding: 0;
  margin: 0;
  border: 0;
  overflow: hidden;
}
textarea#value {
  font-weight: bold;
}
textarea {
  display: block;
  font-family: fixed;
  font-size: 10px;
  padding: 0;
  margin: 0;
  width: 99%;
}
</style>
```

The HTML head is then ended with the typical ending tag:

```html
</head>
```

The body of the HTML file defines a `<textarea>` element and references the external JavaScript files in `<script>` elements:

```html
<body>
<textarea id="value"></textarea>
<script type="text/javascript" src="js/jquery.min.js"></script>
<script type="text/javascript" src="js/wc_simple.js"></script>
</body>
```

Finally, we end the HTML page with the final tag:

```html
</html>
```

The **wc_simple.js** file

The JavaScript code implementing the gICAPI web component starts with some global variables. These variables will hold information that must be persistent during the web component life:

```javascript
var has_focus;
```

The global function `onICHostReady()` will be called by the front-end, when the web component interface is ready. The version passed as parameter allows you to check that your component code is compatible with the current gICAPI framework, and to define and assign the `gICAPI.on*` callback methods (these will be defined in the body of the `onICHostReady()` function:

```javascript
var onICHostReady = function(version) {
```
if ( version != "1.0" ) {
    alert('Invalid API version');
}

... see below for gICAPI interface functions ...

At this point, the gICAPI interface is ready and the gICAPI object can be used.

The gICAPI.onData() function must be implemented to detect web component value changes done in the program, and to acknowledge SetData() calls:

```javascript
    gICAPI.onData = function(content) {
        $('textarea#value').val(content);
    }
```

The onFocus() function is used to detect that the web component has got or lost the focus. If the focus is gained, we need to explicitly set the focus to the expected web component element:

```javascript
    gICAPI.onFocus = function(polarity) {
        has_focus = polarity;
        if (has_focus) {
            $('textarea#value').focus();
        }
    }
```

**Note:** The only way to detect that the focus was gained by the web component field, is when onFocus(true) is called.

We bind a function to the window.resize() event in order to reset the size of the textarea element:

```javascript
    var wc_resize = function() {
        var h = document.body.clientHeight;
        try {
            var log_height = $('textarea#log').height();
            $('textarea#value').height(h - log_height - 10);
        } catch (err) {
        }
    }

    $(window).resize(function() {
        wc_resize();
    });
```

When the textarea element gets the focus, we ask the focus to the runtime system:

```javascript
    $('textarea#value').on("focus", function() {
        gICAPI.SetFocus();
    });
```

Implement the gICAPI.onFlushData() function, to provide textarea content, when the front-end needs to send the field value to the runtime system:

```javascript
    gICAPI.onFlushData = function() {
        gICAPI.SetData( $('textarea#value').val() );
    }
```
Setup the web component when the field state changes by implementing the `gICAPI.onStateChanged()` function:

```javascript
    gICAPI.onStateChanged = function(ps) {
      var params = JSON.parse(ps);
      if ( params.active ) {
        $('textarea#value').attr('disabled', false);
      } else {
        $('textarea#value').attr('disabled', true);
      }
    };
```

At the end of the `onICHostReady()` function, we perform an initial resize:

```javascript
    wc_resize(); // Force adjustment when starting ...
```

**Complete source code**

File `webcomp.per`:

```plaintext
LAYOUT (TEXT="Simple web component")
GRID
  { id: [id] [wc] [ ] [ ] [ ] [ ] [ ] [ ] [tx] [ ] [ ] [ ] [ ] [ ] } END
END

ATTRIBUTES
EDIT id = FORMONLY.id;
WEBCOMPONENT wc = FORMONLY.wc,
  COMPONENTTYPE = "wc_simple",
  SCROLLBARS = NONE,
  STRETCH = BOTH;
TEXTEDIT tx = FORMONLY.info;
END
```

File `webcomp.4gl`:

```plaintext
MAIN
  DEFINE rec
  RECORD
    id INTEGER,
    wc STRING,
    info STRING
  END RECORD

  OPTIONS INPUT WRAP, FIELD ORDER FORM

  OPEN FORM f1 FROM "webcomp"
  DISPLAY FORM f1
```
LET rec.id = 123
LET rec.wc = "Hello, world!"

INPUT BY NAME rec.* ATTRIBUTES(UNBUFFERED, WITHOUT DEFAULTS)
BEFORE FIELD wc
  LET rec.info = "BEFORE FIELD wc=
", rec.wc
ON CHANGE wc
  LET rec.info = "ON CHANGE wc =
", rec.wc
AFTER FIELD wc
  LET rec.info = "AFTER FIELD wc =
", rec.wc
ON ACTION show
  LET rec.info = "ON ACTION show wc=
", rec.wc
ON ACTION disable
  CALL DIALOG.setFieldActive("wc", FALSE)
ON ACTION enable
  CALL DIALOG.setFieldActive("wc", TRUE)
ON IDLE 2
  LET rec.info = "ON IDLE wc=
", rec.wc
END INPUT

END MAIN

File wc_simple.html:

```html
<!DOCTYPE html>
<html>
<head>
  <meta http-equiv="content-type" content="text/html" charset="utf-8" />
  <meta name='viewport' content='initial-scale=1.0, maximum-scale=1.0' />

  <style>
    html, body {
      height:100%;
      padding:0;
      margin:0;
      border:0;
      overflow:hidden;
    }

    textarea#value {
      font-weight: bold;
    }

    textarea {
      display: block;
      font-family: fixed;
      font-size: 10px;
      padding:0;
      margin:0;
      width: 99%;
    }
  </style>

</head>
<body>
<textarea id="value"></textarea>
<script type="text/javascript" src="js/jquery.min.js"></script>
<script type="text/javascript" src="js/wc_simple.js"></script>
</body>
</html>
```
File wc_simple.js:

```javascript
var has_focus = false;

var wc_resize = function() {
    var h = document.body.clientHeight;
    try {
        var log_height = $('textarea#log').height();
        $('textarea#value').height(h - log_height - 10);
    } catch (err) {
    }
}

var onICHostReady = function(version) {
    if ( version != "1.0" ) {
        alert('Invalid API version');
    }

    gICAPI.onData = function(content) {
        $('textarea#value').val(content);
    }

    gICAPI.onFocus = function(polarity) {
        has_focus = polarity;
        if (has_focus) {
            $('textarea#value').focus();
        }
    }

    $(window).resize(function() {
        wc_resize();
    });

    $('textarea#value').on("focus", function() {
        gICAPI.SetFocus();
    });

    // $('textarea#value').on("change keyup paste cut", function() {
    //     gICAPI.SetData( $(this).val() );
    // });

    gICAPI.onFlushData = function() {
        gICAPI.SetFlushedData( $('textarea#value').val() );
    }

    gICAPI.onStateChanged = function(params) {
        var params = JSON.parse(ps);
        if ( params.active ) {
            $('textarea#value').disabled = false;
        } else {
            $('textarea#value').disabled = true;
        }
    }

    wc_resize(); // Force adjustment when starting ...
}
```

Example 3: Application images

Introduction

This topic shows how to display application images in a gICAPI-based web component.
In this example, we will focus on the technique to display application images dynamically in gICAPI web component HTML content, by using the `ui.Interface.filenameToURI()` method.

This sample application can be used with any Genero front-end configuration (as a web application with the GAS, in direct (development) mode with GDC/GMA/GMI, or as a mobile app running on a device)

For gICAPI programming basics, see Example 2: Simple text input on page 1955.

The complete code example with program and form file is available at the end of this topic.

**HTML code description**

The HTML source file starts with the typical HTML tags:

```html
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="content-type" content="text/html" charset="utf-8" />
<meta name='viewport' content='initial-scale=1.0, maximum-scale=1.0' />
</head>
<body height="100%" width="100%">
<h2>Testing application images in gICAPI Web Component</h2>
<img name="myimage" />
</body>
</html>
```

The JavaScript code defines the `onICHostReady()` function, and the `set_image()` function, to be called with the `webcomponent.call` front call. The `set_image()` function sets the `src` attribute in the image element:

```javascript
<script language="JavaScript">
var onICHostReady = function(version) {
  if ( version != 1.0 )
    alert('Invalid API version');
}

var set_image = function(ressource) {
  var ie = document.getElementsByName("myimage")[0];
  ie.src = ressource;
}
</script>
```

Close the HTML head element with the `</head>` ending tag:

```html
</head>
```

The body of the HTML page contains two elements:

- an `h2` title,
- the image element, identified by a name:

```html
<body height="100%" width="100%">
  <h2>Testing application images in gICAPI Web Component</h2>
  <img name="myimage" />
</body>
```

**Application directory structure**

In order to easily build and install on mobile devices, create the following directory structure:

```plaintext
top-dir
|-- fglprofile
|-- main.4gl
|-- main.42m
```
Providing image files

Copy some of your favorite images in the "images" directory.

The sample program will scan this directory to fill a combobox and let you choose the image to be displayed:

```gml
FUNCTION init_image_list(cb)
  DEFINE cb ui.ComboBox
  DEFINE h INTEGER,
       fn STRING
  LET
    h=os.Path.dirOpen(os.Path.join(base.Application.getProgramDir(),"images"))
  WHILE h > 0
    LET fn = os.Path.dirNext(h)
    IF fn IS NULL THEN EXIT WHILE END IF
    IF fn=="." OR fn==".." THEN CONTINUE WHILE END IF
    CALL cb.addItem(fn, fn)
  END WHILE
END FUNCTION
```

Note: When deployed on a mobile device, the images directory will be part of the application program files. Thus to access the directory you need to add the `base.Application.getProgramDir` on page 2408 path. For more details, see Directory structure for GMA apps on page 3589 and Directory structure for GMI apps on page 3604.

In the program code, the ON CHANGE image interaction block will perform a front call to set the image resource in the gICAPI web component:

```gml
ON CHANGE image
  LET rec.uri = ui.Interface.filenameToURI(rec.image)
  CALL ui.Interface.frontCall("webcomponent","call",
    ["formonly.wc","set_image",rec.uri],[])
```

FGLIMAGEPATH environment settings

In order to find image resources when not executing behind a GAS, you need to define the FGLIMAGEPATH environment variable as follows:

```
$ FGLIMAGEPATH=$PWD/images:.
```
For deployed mobile applications, the FGLIMAGEPATH environment variable must be set in the default fglprofile file, by using the $FGLAPPDIR place holder:

```
mobile.environment.FGLIMAGEPATH = "$FGLAPPDIR/images:"  
```

For more details about FGLIMAGEPATH settings, see Providing the image resource on page 1149.

**Complete source code**

File *myform.per*:

```
LAYOUT
GRID
{
  Current image:  [f1]
  Image URI:     [f2]
  [wc]
  [       ]
  [       ]
  [       ]
  [       ]
}
END
END

ATTRIBUTES
COMBOBOX f1 = FORMONLY.image,
            INITIALIZER = init_image_list;
EDIT f2 = FORMONLY.uri, SCROLL;
WEBCOMPONENT wc = FORMONLY.wc,
                   COMPONENTTYPE="mywebcomp",
                   STRETCH=BOTH;
END
```

File *main.4gl*:

```
IMPORT os

MAIN
  DEFINE rec RECORD
      image STRING,
      uri STRING,
      wc STRING
  END RECORD
  OPEN FORM f1 FROM "myform"
  DISPLAY FORM f1
  INPUT BY NAME rec.* WITHOUT DEFAULTS ATTRIBUTES(UNBUFFERED)
    ON CHANGE image
      LET rec.uri = ui.Interface.filenameToURI(rec.image)
      CALL ui.Interface.frontCall("webcomponent","call",
                                ["formonly.wc","set_image",rec.uri],[])
  END INPUT
END MAIN

FUNCTION init_image_list(cb)
  DEFINE cb ui.ComboBox
  DEFINE h INTEGER,
      fn STRING
  LET
    h=os.Path.dirOpen(os.Path.join(base.Application.getProgramDir(),"images"))
    WHILE h > 0
      LET fn = os.Path.dirNext(h)
    END WHILE
  END LET
```

File mywebcomp.html:

```html
<!DOCTYPE html>
<html>
<head>
<title>Test</title>
<meta http-equiv="Cache-Control" content="no-cache">
<meta http-equiv="Pragma" content="no-cache">
<meta http-equiv="Expires" content="0">
<script LANGUAGE="JavaScript">
var onICHostReady = function(version) { 
    if ( version != 1.0 ) 
        alert('Invalid API version');
}

var set_image = function(ressource) {
    var ie = document.getElementsByName("myimage")[0];
    ie.src = ressource;
}
</script>
</head>
<body>
<h2>Testing application images in gICAPI Web Component</h2>
<img name="myimage" />
</body>
</html>
```

**Example 4: Color picker**

**Introduction**

This topic describes the different steps to implement a color picker with a gICAPI-based web component.

In this example, we will implement a simple color picker, that will allow the user the select a color from a predefined set. Colors are drawn as square boxes using SVG graphics, user can change the current selected color with a separate COMBOBOX field, modify the title of the HTML body, and query for the color list with a webcomponent.call front call.

The HTML file is described in detail, and complete code example with program and form file is available at the end of this topic.

**HTML code description**

As any HTML source code, the file starts with the typical HTML tags:

```html
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="content-type" content="text/html" charset="utf-8" />
<meta name='viewport' content='initial-scale=1.0, maximum-scale=1.0' />
</head>
<body>
</body>
</html>
```

**Note:** The "viewport" meta is provided to adjust the viewport size for mobile devices.
The JavaScript code needs to be enclosed in a `<script>` element:

```html
<script language="JavaScript">
```

Global variables are defined to hold information that must be persistent during the web component life:

```javascript
var current_color;
var wanted_color;
var has_focus;
```

The global function `onICHostReady()` will be called by the front-end, when the web component interface is ready. The version passed as parameter allows you to check that your component code is compatible with the current gICAPI framework, and to define and assign the `gICAPI.on*` callback methods (these will be defined in the body of the `onICHostReady()` function:

```javascript
var onICHostReady = function(version) {
    if ( version != "1.0") {
        alert('Invalid API version');
    }
    ...
    some initialization code ...
    gICAPI.onProperty = function(properties) {
        ...
    }
    gICAPI.onData = function(data) {
        ...
    }
    gICAPI.onFocus = function(polarity) {
        ...
    }
};
```

At this point, the gICAPI interface is ready and the `gICAPI` object can be used.

The `gICAPI.onProperty()` method is called when a web component property changes (properties will be initialized at form creation, or changed during form usage). In this code example, when the property "title" is changed by the program, the element with id="title" is updated with the new title:

```javascript
gICAPI.onProperty = function(properties) {
    try{
        var ps = JSON.parse(properties);
        document.getElementById("title").innerHTML = ps.title;
    } catch (err){
        console.error("onProperty(): Invalid JSON string");
    }
};
```

**Note:** The **ON ACTION change_title** in the dialog code will change the title property after the form initialization, to show that the `gICAPI.onProperty()` function can also be invoked after the web component field creation.
The `showFocusRectangle()` function shows a border around the specified color item (SVG element), following the color identifier passed as parameter and the focus status (focus can be true, false or -1, to keep the current border color and just modify the position of the border):

```javascript
var showFocusRectangle = function(color, focus) {
    // See complete code example for details
}
```

The `changeColor()` function implements a color change, by registering a field value change with `gICAPI.SetData()`, and by triggering a specific action with `gICAPI.Action()`, to inform the program that a color was selected:

```javascript
var changeColor = function(color) {
    current_color = color;
    showFocusRectangle(current_color, true);
    gICAPI.SetData(current_color);
    gICAPI.Action("color_selected");
}
```

Next lines implement the `gICAPI.onFocus()` function, executed when the web component gets or loses the focus. The code distinguishes the case when the focus is gained (by a mouse click on a color item), selecting a new color with a call to `changeColor()`, and the case when the focus is set to the web component by the runtime system. A blue border will be added to the current color item, when the component gets the focus, and the border color is reset to gray when the focus is lost:

```javascript
gICAPI.onFocus = function(polarity) {
    if ( polarity == true ) {
        has_focus = true;
        if (wanted_color != undefined) {
            changeColor(wanted_color);
            wanted_color = undefined;
        } else {
            showFocusRectangle(current_color, true);
        }
    } else {
        has_focus = false;
        showFocusRectangle(current_color, false);
    }
}
```

The `gICAPI.onData()` function must be implemented to detect web component value changes done in the program, and to acknowledge `gICAPI.SetData()` calls. This will be triggered by assigning the `rec.webcomp` variable in the dialog code, typically in the ON CHANGE color block, when modifying the combobox value. The `showFocusRectangle()` function moves the focus border to the color item corresponding to the color identifier passed as parameter:

```javascript
gICAPI.onData = function(data) {
    current_color = data;
    showFocusRectangle(current_color, -1);
}
```

The `selectColor()` function will be called through the `onclick` event of the `<rect>` SVG elements representing colors. If the web component does not have the focus yet, the function will call `gICAPI.SetFocus()`, in order to ask the runtime system, if the focus can go to the web component field. If the runtime system accepts to set the focus to the web component field, the `gICAPI.onFocus()` method will be called with `true` as parameter, and will handle the requested color change (using `wanted_color`). If the focus cannot be set to the web component, the `onFocus()` method will not be called:

```javascript
var selectColor = function(color) {
```
if (has_focus) {
    changeColor(color);
} else {
    wanted_color = color;
gICAPI.SetFocus();
    // Color item change is done in onFocus(), because
    // VM may refuse to set the focus to the wc field.
}
}

Note: The only way to detect that the focus was gained by the web component field, is when onFocus(true) is called.

End the JavaScript element with the </script> ending tag:

</script>

Close the HTML head element with the </head> ending tag:

</head>

The rest of the HTML page defines the graphical elements for the color picker, with a <h3> title and and an <svg> element containing <rect> element to show clickable color items. Note the <rect> element with id="focus_rectangle", used to show a border for the current color item:

<body height="100%" width="100%">
<h3 id="title">no-title</h3>
<svg id="svg_container" width="230" height="130">
    <rect x="5" y="5" rx="5" ry="5" width="30" height="30"
        id="#FFFFCC"
        style="fill:#FFFFCC;stroke:black;stroke-width:1"
        onclick="selectColor('#FFFFCC')" />
...
    <rect x="178" y="73" rx="7" ry="7" width="34" height="34"
        id="focus_rectangle"
        style="fill:none;stroke:gray;stroke-width:3" />
</svg>
</body>

Complete source code

File color_picker.per:

LAYOUT
GRID
{
    Id:  [f1            
    [f2                     
    [                       
    [                       
    [                       
    [                       
    [                       
    [                       
    Color: [f3            ]
}
File color_picker.4gl:

IMPORT util

MAIN

DEFINE rec RECORD
   id INTEGER,
   webcomp STRING,
   pgcolor STRING,
   info STRING
END RECORD,
   f ui.Form,
   n om.DomNode,
   tmp STRING,
   colors DYNAMIC ARRAY OF STRING

OPTIONS INPUT WRAP

OPEN FORM f1 FROM "color_picker"
DISPLAY FORM f1

LET rec.id = 98344
LET rec.webcomp = "#FF0000"
LET rec.pgcolor = rec.webcomp

INPUT BY NAME rec.* WITHOUT DEFAULTS

ATTRIBUTES(UNBUFFERED)

ON CHANGE pgcolor
   LET rec.webcomp = rec.pgcolor

ON ACTION color_selected ATTRIBUTES( DEFAULTVIEW=NO )
   IF rec.webcomp == "#000000" THEN
      LET rec.webcomp = rec.pgcolor
      LET rec.info = NULL
      ERROR "Black color is denied!"
   ELSE
      LET rec.pgcolor = rec.webcomp
      LET rec.info = "Color selected:", rec.pgcolor
   END IF
ON ACTION change_title ATTRIBUTES(TEXT="Change title")
LET f = DIALOG.getForm()
LET n = f.findNode("Property", "title")
CALL n.setAttribute("value", "New title")
LET rec.info = "Title changed."

ON ACTION get_colors ATTRIBUTES(TEXT="Get colors")
TRY
   CALL ui.Interface.frontCall("webcomponent", "call",
      ["formonly.webcomp", "getColorList"], [tmp] )
   CALL util.JSON.parse(tmp, colors)
   LET rec.info = "Color list: ", tmp
CATCH
   ERROR "Front call failed."
END TRY

END INPUT
END MAIN

File color_picker.html:

<!DOCTYPE html>
<html>
<head>
<meta http-equiv="content-type" content="text/html" charset="utf-8" />
<meta name='viewport' content='initial-scale=1.0, maximum-scale=1.0' />
<script language="JavaScript">
var current_color;
var wanted_color;
var has_focus;

var onICHostReady = function(version) {
   if ( version != "1.0" ) {
      alert('Invalid API version');
   }

   current_color = "#000000";

   gICAPI.onProperty = function(properties) {
      try{
         var ps = JSON.parse(properties);
         document.getElementById("title").innerHTML = ps.title;
      }
      catch (err){
         console.error("onProperty(): Invalid JSON string");
      }
   }  

   gICAPI.onFocus = function(polarity) {
      if ( polarity == true ) {
         has_focus = true;
         if (wanted_color != undefined) {
            changeColor(wanted_color);
            wanted_color = undefined;
         } else {
            showFocusRectangle(current_color, true);
         }
      } else {
         has_focus = false;
      }
   }
</script>
showFocusRectangle(current_color, false);
}
}

gICAPI.onData = function(data) {
    current_color = data;
    showFocusRectangle(current_color, -1);
}

var showFocusRectangle = function(color, focus) {
    var f = document.getElementById("focus_rectangle");
    var e = document.getElementById(color);
    if (e == null) {
        e = document.getElementById("#000000");
    }
    var e_x = e.getAttribute("x") - 2;
    var e_y = e.getAttribute("y") - 2;
    f.setAttribute("x", e_x);
    f.setAttribute("y", e_y);
    if (focus == true) {
        f.style.stroke = "blue";
    } else if (focus == false) {
        f.style.stroke = "gray";
    }
}

var changeColor = function(color) {
    current_color = color;
    showFocusRectangle(current_color, true);
    gICAPI.SetData(current_color);
    gICAPI.Action("color_selected");
}

var selectColor = function(color) {
    if (has_focus) {
        changeColor(color);
    } else {
        wanted_color = color;
        gICAPI.SetFocus();
        // Color item change is done in onFocus(), because
        // VM may refuse to set the focus to the wc field.
    }
}

var getColorList = function() {
    var colors = [];
    var p = document.getElementById("svg_container");
    var items = p.getElementsByTagName("rect");
    for (var i = items.length; i--; ) {
        var c = items[i].getAttribute("id");
        if (c.indexOf("#") == 0) {
            colors.push(c);
        }
    }
    return JSON.stringify(colors);
}

</script>
</head>
<body height="100%" width="100%">
<h3 id="title">no-title</h3>
Building user interface

User interface

1972

Built-in web components
Genero provides a set of ready-to-use web components.

Understanding built-in web components
This section describes the available built-in web components and how to use them.

The Genero web components are located in the $FGLDIR/webcomponents directory. Additional utility libraries are provided in $FGLDIR/src (.4gl sources) and $FGLDIR/lib (compiled .42m modules).

Unlike user-defined web components, there is no need to deploy the built-in web components. The built-in web components are available for any type of front-end, including mobile platforms.

Note: Built-in web components are found automatically by the runtime system. There is no need to add the $FGLDIR/webcomponents path to the FGLIMAGEPATH environment variable.

Related tasks
Using a gICAPI web component on page 1932
This section describes how to add a gICAPI-based web component to your application.

Built-in web components reference
List of all built-in web components
The fglrichtext web component
The fglrichtext built-in web component implements an HTML text editor.

The fglrichtext built-in web component uses the gICAPI web component framework.
**HTML content in the fglrichtext web component**

The fglrichtext web component generates standard HTML, without any “flavour” based on external library or CSS. Therefore, depending HTML rendering engine, the visual result might be slightly different.

For example, when using the fglrichtext web component to create a blog post, you want the blog theme to be used to display this post, and not something completely out of the theme.

Thus, fglrichtext generates as "abstract" HTML as possible (without any decoration), to apply different CSS styles afterwards.

**Using images in rich text content**

The fglrichtext web component allows insertion of image resources into the text. This can be done with the `image` option of the toolbar, by copy/paste, or drag&drop operations.

When inserting an image resource from the local platform (for example, with the `image` option of the toolbar), the complete image content will be inserted as a base64 encoded object in the text.

**Note:** Inserting huge base64 encoded image data can lead to performance issues.

When inserting an image resource from a copy/paste from a web resource (like an HTML page), only the URL of the image resource is inserted in the text. If this URL is no longer available, the image will disappear from the text.

**Defining the fglrichtext web component in the form file**

In your `.per` form definition file, define the HTML text editor as a `WEBCOMPONENT` form item with the `COMPONENTTYPE` attribute set to the "fglrichtext" value.
The fglrichtext web component provides its own built-in scrollbars. Therefore, the SCROLLBARS attribute can be set to NONE.

Use SIZEPOLICY=FIXED and STRETCH=BOTH, to get a text editor that is resized with the parent window.

For example:

```
LAYOUT
GRID
{
[rt                ]
[                          ]
[                          ]
...
ATTRIBUTES
WEBCOMPONENT rt = FORMONLY.richtext,
    COMPONENTTYPE = "fglrichtext",
    PROPERTIES = (
        toolbar = "bold|italic|underline|undo|redo"
    ),
    SCROLLBARS = NONE,
    SIZEPOLICY = FIXED,
    STRETCH = BOTH;
...
```

**Properties of the fglrichtext**

The fglrichtext web component supports the following PROPERTIES:

1. toolbar: Defines the toolbar elements layout.
2. autoflush / autoflush_interval: Defines an action and timer for auto flushing the value.
3. spellcheck: Defines the spell checking in the fglrichtext.
4. noteditable: Defines if the fglrichtext is read-only.
5. Properties to define the default font:
   - font_family: Defines the default font family.
   - font_size: Defines the default font size.
   
   For details, see Specifying the default font on page 1977.
6. Properties to localize fglrichtext labels:
   - labels_toolbar_tooltips: Defines the texts to be displayed for toolbar tooltips.
   - labels_toolbar_link: Defines the labels to be displayed for the link popup dialog.
   - labels_toolbar_linespacing: Defines the texts for line spacing options.
   - labels_toolbar_formatselect: Defines the texts to be format options.
   
   For details, see Localizing rich text component strings on page 1977.

**Configuring the toolbar**

Define the controls to be used in the editor toolbar with the toolbar property in the PROPERTIES attribute.

Toolbar items must be separated with a pipe:

```
WEBCOMPONENT rt = FORMONLY.richtext,
...
PROPERTIES = (
    toolbar = "bold|italic|underline|undo|redo|emoji"
),
...
```

The fglrichtext web component supports the following toolbar items:
Table 397: fglrichtext toolbar options

<table>
<thead>
<tr>
<th>Option name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aligncenter</td>
<td>Centers the current block or image.</td>
</tr>
<tr>
<td>alignjustify</td>
<td>Justifies the current block or image.</td>
</tr>
<tr>
<td>alignleft</td>
<td>Left aligns the current block or image.</td>
</tr>
<tr>
<td>alignnone</td>
<td>Removes the alignment of the current block or image and left aligns it.</td>
</tr>
<tr>
<td>alignright</td>
<td>Right aligns the current block or image.</td>
</tr>
<tr>
<td>bold</td>
<td>Applies the bold format to the current selection.</td>
</tr>
<tr>
<td>bullist</td>
<td>Formats the current selection as a bullet list.</td>
</tr>
<tr>
<td>color</td>
<td>Applies text color to selection. This opens a color picker pop-up, allowing you to choose text and/or background color.</td>
</tr>
<tr>
<td>emoji</td>
<td>Opens a picker for selecting an emoji to insert into the field. Important: Emoji are not simple images, they are Unicode characters. They may look different depending on the display device, and may not be supported on some systems. If your device does not handle emoji, you may need to install fonts that handle emoji characters.</td>
</tr>
<tr>
<td>fontselect</td>
<td>Dropdown list with font families to apply to selection.</td>
</tr>
<tr>
<td>fontsizeselect</td>
<td>Dropdown list with font sizes to apply to selection.</td>
</tr>
<tr>
<td>formatselect</td>
<td>Dropdown list with block formats to apply to selection.</td>
</tr>
<tr>
<td>indent</td>
<td>Indents the current list item or block element.</td>
</tr>
<tr>
<td>image</td>
<td>Inserts an image resource found on the local platform.</td>
</tr>
<tr>
<td>insert</td>
<td>Create and delete links.</td>
</tr>
<tr>
<td>italic</td>
<td>Applies the italic format to the current selection.</td>
</tr>
<tr>
<td>linespacing</td>
<td>Defines the line spacing of a selected text.</td>
</tr>
<tr>
<td>numlist</td>
<td>Formats the current selection as a numbered list.</td>
</tr>
<tr>
<td>outdent</td>
<td>Outdents the current list item or block element.</td>
</tr>
<tr>
<td>redo</td>
<td>Redoes the last undo operation.</td>
</tr>
<tr>
<td>removeformat</td>
<td>Removes the formatting from the current selection.</td>
</tr>
<tr>
<td>strikethrough</td>
<td>Applies strike though format to the current selection.</td>
</tr>
<tr>
<td>underline</td>
<td>Applies the underline format to the current selection.</td>
</tr>
</tbody>
</table>
Using the spell checker

The fglrichtext web component can be configured for spell checking with the spellcheck property of the PROPERTIES attribute:

```plaintext
WEBCOMPONENT rt = FORMONLY.richtext,
    ...
    PROPERTIES = (
        spellcheck = "browser"
    ),
    ...
```

The spellcheck property can be set to the following values:

- "browser" (default): This will enable the intrinsic spell checking of the browser.
- "none": No spell checking is required for this rich text field.

**Important:** The spellcheck property is supported by the GDC and GBC. It is not supported by GMI or GMA.

**Note:** GDC has specific features regarding spell checking in web components. For example, in GDC 3.10 there is a limited set of supported languages, and if spellcheck is disabled in the web component, enabling the feature in the context menu will have no effect. For more details, see the Spellchecking in Web Components topic in the Genero Desktop Client User Guide.

Auto-flushing the rich text content

By default, the text value of an fglrichtext field is only flushed to the runtime system, when focus leaves the field or when firing an action.

The autoflush property of the PROPERTIES attribute allows you to define a user action to be fired every autoflush_interval seconds, when the text is changed. The default for autoflush_interval is 10 seconds.

**Note:** When the user enters text, the autoflush action will be fired every autoflush_interval seconds. After the autoflush action is sent, if the user does not modify the text, no action will be fired until the user starts again to type text. This way, the network traffic is optimized.

```plaintext
WEBCOMPONENT rt = FORMONLY.richtext,
    ...
    PROPERTIES = (
        autoflush = "save_text",
        autoflush_interval = 5  -- Flush every 5 seconds
    ),
    ...
```

In the program code, implement an ON ACTION handler using the action name that matches the autoflush property of the WEBCOMPONENT field. When the action is fired, the rich text will be available in the WEBCOMPONENT field value. Use the UNBUFFERED dialog mode to have the text automatically copied into the program variable bound to the form field:

```plaintext
INPUT BY NAME rec.richtext, -- Variable bound to the WEBCOMPONENT form field ...
    ...
    ATTRIBUTES(UNBUFFERED) -- To get the input buffer copied into the variable
    ...
    ON ACTION save_text ATTRIBUTES(DEFAULTVIEW = NO)
        UPDATE ord_info SET oi_comment = rec.richtext
```
WHERE oi_pkey = curr_pkey

Auto-flush can for example be used to implement automatic backup of the text content.

**Defining the rich text field as read-only**

Like with the TEXTEDIT NOTEDITABLE attribute, it is possible to make the fglrichtext web component read-only, by defining the noteditable property:

```plaintext
WEBCOMPONENT rt = FORMONLY.richtext,
    ...
    PROPERTIES = (noteditable = true),
    ...
```

**Specifying the default font**

Default font attributes can be selected for the fglrichtext web component by using the following properties:

**Table 398: Default font properties**

<table>
<thead>
<tr>
<th>Property name</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>font_family</td>
<td>Monospace, Sans serif, Serif, Roboto</td>
</tr>
<tr>
<td>font_size</td>
<td>nnnpx</td>
</tr>
</tbody>
</table>

For example:

```plaintext
WEBCOMPONENT rt = FORMONLY.richtext,
    ...
    PROPERTIES = (font_family = "Monospace",
                  font_size = "25px"),
    ...
```

The font_size defines the px unit, otherwise, the property will be ignored.

If the toolbar items fontselect/fontsizeselect are used, they will display the font family and font size defined by the properties, when the fglrichtext initializes.

If the user moves the edit cursor to a text element that uses a different font to the one specified with these attributes, the richtext will reset the current font settings to match that text element.

**Localizing rich text component strings**

The toolbar button tooltips, combobox items, and popup dialog labels of the fglrichtext web component can be localized with dedicated properties.

**Note:** Consider using localized strings to define the texts in external string ressource files.

For each context, the texts must be provided as a set of JSON formatted key/value pairs, where the key identifies the element to get the text:

```plaintext
label-localization-property = '{ "item-name" : "item-label" [,....] }'
```
Table 399: Properties for fglrichtext string localization and related item identifiers

<table>
<thead>
<tr>
<th>Property name</th>
<th>Items/Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>labels_toolbar_link</td>
<td>&quot;save&quot;, &quot;label&quot;, &quot;edit&quot;, &quot;visit&quot;, &quot;remove&quot;</td>
</tr>
<tr>
<td>labels_toolbar_linespacing</td>
<td>&quot;small&quot;, &quot;normal&quot;, &quot;large&quot;, &quot;huge&quot;</td>
</tr>
<tr>
<td>labels_toolbar_formatselect</td>
<td>&quot;heading&quot;, &quot;normal&quot;</td>
</tr>
</tbody>
</table>

**Note:** For combobox items like with `labels_toolbar_linespacing`, the item/key names must be specified in lowercase ("small": "Petit", not "Small": "Petit")

For example (not using localized strings):

```plaintext
WEBCOMPONENT rt = FORMONLY.richtext,
...
PROPERTIES = {
    labels_toolbar_tooltips = '{ "aligncenter": "Centré", "alignjustify": "Justifié", ... }',
    labels_toolbar_link = '{ "save": "OK", "label": "Entrez un lien:" }',
    labels_toolbar_linespacing = '{ "small": "Petit", "normal": "Normal", "large": "Grand", ... }',
    labels_toolbar_formatselect = '{ "heading": "Titre", "normal": "Normal" }',
},
...
```

Example using localized strings:

```plaintext
labels_toolbar_tooltips = "%rtlabels.toolbar.tooltips"
```

and then in the .str file:

```plaintext
rtlabels.toolbar.tooltips = '{ "aligncenter": "Centré", "alignjustify": "Justifié", ... }'
```

Examples:
- fglrichtext built-in web component usage examples.

Example 1: Simple INPUT with fglrichtext

Form definition file `simple_richtext.per`:

```plaintext
LAYOUT (TEXT="Rich Text", MINWIDTH=80, MINHEIGHT=20)
GRID
{
    [rt]
    [ ]
    [ ]
    [ ]
    [info ]
}
END
```
The fglgallery web component

The fglgallery built-in web component implements a picture gallery, to let the end user select pictures and photos.

The fglgallery built-in web component is a gICAPI web component.

Several rendering modes are supported: Mosaic, List and Thumbnails for basic rendering.
Defining the fglgallery web component in the form file

In your .per form definition file, define the image gallery as a WEBCOMPONENT form item with the COMPONENTTYPE attribute set to the "fglgallery" value.

As the image gallery implements its own navigation controls or scrollbars, the SCROLLBARS attribute can be set to NONE.

Use SIZEPOLICY=FIXED and STRETCH=BOTH, to get an image gallery that resizes with the parent window.

For example:

```
LAYOUT
GRID
{
 [ig        ]
 [          ]
 [          ]
 ...}

ATTRIBUTES
WEBCOMPONENT ig = FORMONLY.gallery_wc,
COMPONENTTYPE = "fglgallery",
PROPERTIES=(selection="image_selection"),
```
Introducing the fglgallery.4gl library

The fglgallery.4gl module implements a set of functions to control an fglgallery web component. The source can be found in $FGLDIR/src/webcomponents/fglgallery.

Customizing the image gallery

The HTML elements created by fglgallery can be customized with CSS styling.

Default CSS styles for fglgallery are provided in $FGLDIR/webcomponents/fglgallery/css/fglgallery.css.

Overwrite or define new style attributes for each fglgallery element type, in a secondary file named $FGLDIR/webcomponents/fglgallery/css/fglgallery-custom.css, and uncomment the link in $FGLDIR/webcomponents/fglgallery/fglgallery.html to load this custom CSS file after fglgallery.css.

For example, to define a background color for the list display mode, add the following lines in fglgallery-custom.css:

```
.list-main {
    background-color: black;
}
```

Note: Do not modify the default CSS styles in fglgallery.css: This file will be overwritten by a new installation. Use the fglgallery-custom.css file name instead.

fglgallery library initialization and finalization

Library initialization and finalization functions are provided to prepare the library before usage, and free resources when the library is not longer needed:

```
CALL fglgallery.initialize()
...
CALL fglgallery.finalize()
```

For more details see the initialize() and finalize() functions.

Creating and destroying an image gallery handle

Before using the image gallery web component, you must create a handle with the create() function:

```
DEFINE id INTEGER
...
LET id = fglgallery.create("formonly.gallery_wc")
```

The create() function takes the WEBCOMPONENT form item name as parameter, and returns the id of the image gallery handle. The image gallery id will be used in subsequent library calls. This technique allows for the manipulation of several image gallery items in the same form, and the use of the library by different callers.

When the image gallery web component is no longer used, destroy the handle with the destroy() function:

```
CALL fglgallery.destroy(id)
```
The *fglgallery.t_struct_value* type

The fglgallery library provides a user-type to define a variable that will be used to send and receive data from the image gallery web component.

The *fglgallery.t_struct_value* type is a RECORD structure, that holds image selection information:

```
PUBLIC TYPE t_struct_value RECORD
  current INTEGER,
  selected DYNAMIC ARRAY OF INTEGER
END RECORD
```

In this structure, each picture is identified by its ordinal position in the image gallery.

Data type for the fglgallery WEBCOMPONENT field

The fglgallery WEBCOMPONENT field value will be used to exchange image selection information in JSON format.

Use STRING as data type for the variable bound to the WEBCOMPONENT field.

The string variable can then be used to map the JSON to the *fglgallery.t_struct_value* record:

```
DEFINE rec RECORD
  ...  
gallery_wc STRING,
  ...
END RECORD
DEFINE ...
INPUT BY NAME rec.* ...
...
ON ACTION image_selection
  CALL util.JSON.parse( rec.gallery_wc, struct_value )
  DISPLAY "Current: ", fglgallery.getPath(id, struct_value.current)
...
```

Filling the image gallery with pictures

In order to add a picture resource to the image gallery, use the *addImage()* function. This function takes three parameters:

1. The id of the image gallery handle,
2. The URL of the picture resource,
3. The text/comment for the picture.

**Important:** When using local application pictures, you need to understand how these pictures are transmitted to the front-end for display inside the WEBCOMPONENT. For more details, see Using image resources with the glICAPI web component on page 1952.

Pictures are identified by their ordinal position in the image gallery.

To add a picture resource that resides on the application server where the program executes, use the `ui.Interface.filenameToURI()` method:

```
CALL fglgallery.addImage(id,  
  ui.Interface.filenameToURI("big_smiley.jpg"),  
  "The big smiley.")
```

To add a picture resources that is available as a URL, from the local network or the internet, provide the URL directly:

```
CALL fglgallery.addImage(id,  
  "https://example.com/smile.jpg")
```
After adding pictures, display the image gallery with `display()`:

```plaintext
CALL fglgallery.display(id, FGLGALLERY_TYPE_MOSAIC, FGLGALLERY_SIZE_NORMAL)
```

Or call the `flush()` function, when the image gallery is already displayed:

```plaintext
CALL fglgallery.flush(id)
```

This flush mechanism allows you to add a set of pictures in one front-end exchange.

**Get the number of pictures in the gallery**

To get the number of pictures that are currently displayed in the gallery, call the `getImageCount()` function:

```plaintext
DISPLAY fglgallery.getImageCount(id)
```

**Cleaning the image gallery**

Use the `clean()` function to remove all pictures from the image gallery:

```plaintext
CALL fglgallery.clean(id)
```

**Displaying the image gallery**

To display the image gallery, use the `display()` function. This function takes the image gallery id as parameter, followed by the gallery type and size options:

```plaintext
CALL fglgallery.display(id, FGLGALLERY_TYPE_MOSAIC, FGLGALLERY_SIZE_NORMAL)
```

**Note:** This function can be called several times, in order to change the type and size of the image gallery.

**Image gallery type (FGLGALLERY_TYPE_*)**

The type of the image gallery defines the layout / rendering of the gallery. This type is defined by the second parameter of the `display()` function.

For possible values, see the `display()` function reference topic.

**Image gallery size (FGLGALLERY_SIZE_*)**

The size of the images displayed in the gallery is defined by the third parameter of the `display()` function. The size is specified with an abstract option (small, normal, large, etc). This size is related to the current font (the "em" unit is used in HTML).

For possible values, see the `display()` function reference topic.

**Image elements aspect ratio**

All image elements in the fglgallery are displayed with the same size, to align properly when the image resources have different sizes.

By default, image elements are displayed with a square (1:1) aspect ratio.

The aspect ratio can be configured with the `setImageAspectRatio()` function.
Get the picture resource path from a position
To get the picture resource name from an image position, use the `getPath()` function:

```plaintext
LET path = fglgallery.getPath(id, struct_value.current)
```

Get the picture title from a position
To get the title of a picture from an image position, use the `getTitle()` function:

```plaintext
LET title = fglgallery.getTitle(id, struct_value.current)
```

Delete pictures from the image gallery
To remove pictures from the image gallery, use the `deleteImages()` function. This function takes a DYNAMIC ARRAY OF INTEGER as parameter, defining the ordinal positions of the pictures to be deleted:

```plaintext
DEFINE todel DYNAMIC ARRAY OF INTEGER
LET todel[1] = 3
LET todel[2] = 6
CALL fglgallery.deleteImages(id, todel)
```

Single and multiple picture selection
By default, the end user can only select a single picture from the image gallery. With the Mosaic, List and Thumbnails display modes, you can enable multiple picture selection.

**Note:** Multiple picture selection is not supported with the Light Gallery display mode.

To enable multiple picture selection, use the `setMultipleSelection()` function:

```plaintext
CALL fglgallery.setMultipleSelection(id, TRUE)
```

Picture selection can then be detected with an ON ACTION handler.

Detecting image selection
To implement picture selection in the image gallery component, define the WEBCOMPONENT form field with a "selection" property in the PROPERTIES attribute:

```plaintext
WEBCOMPONENT ig = FORMONLY.gallery_wc,
...  
    PROPERTIES = (selection="image_selection"),
...  
```

Picture selection is built-in to the fglgallery web component.

In the program code, detect picture selection with an ON ACTION handler defined with an action name that matches the "selection" property of the WEBCOMPONENT field.

When the action is fired, selection information is provided in the WEBCOMPONENT field value as a JSON formatted string similar to: `{"current":5,"selected":[4,5]}

**Tip:** In order to get the selection information directly in the program variable bound to the WEBCOMPONENT field, consider using the UNBUFFERED dialog attribute.

This JSON string can be parsed with the `util.JSON.parse()` method, to fill a variable defined with the `fglgallery.t_struct_value` type:

```plaintext
DEFINE struct_value fglgallery.t_struct_value
...  
```
INPUT BY NAME rec.* ATTRIBUTES (UNBUFFERED, WITHOUT DEFAULTS)
...
ON ACTION image_selection
  CALL util.JSON.parse( rec.gallery_wc, struct_value )
  DISPLAY struct_value.current,
       struct_value.selected.getLength()
...

The selection structure contains a "current" field, to identify the current selected picture, and when the multiple-selection option is enabled, the "selection" field contains the list of picture ids that are selected.

**Define selected pictures by program**

In order to set the current picture, and select or de-select pictures from the program code, use the variable defined with the fglgallery.t_struct_value type and assign it as a JSON string to the web component field:

1. Fill the fglgallery.t_struct_value variable with the picture ordinal positions,
2. Convert to a JSON string with util.JSON.stringify(),
3. Set the WEBCOMPONENT field value with the resulting string.

For example, to set picture #3 as current, and select pictures #2, #7 et #15:

```
DEFINE struct_value fglgallery.t_struct_value
...
LET struct_value.current = 3
CALL struct_value.selected.clear()
LET struct_value.selected[1] = 2
LET struct_value.selected[2] = 7
LET struct_value.selected[3] = 15
LET rec.gallery_wc = util.JSON.stringify(struct_value)
```

**Complete fglgallery API reference**

Check the $FGLDIR/src/webcomponents/fglgallery/fglgallery.4gl source for more details about the image gallery API.

**Related reference**

fglgallery: Image gallery module on page 2322

**Examples**

fglgallery built-in web component usage examples.

Example 1: Simple fglgallery example using free pictures from the web

**Note:** This code example implements an fglgallery web component to display images from the web and images located on the application server. The complete demo with server images is available in $FGLDIR/demo/webcomponents/simple_gallery. To transmit the images located on the server when connecting in direct mode to a front-end, define FGLIMAGEPATH to $PWD/images-private:$PWD/images-public. For more details, see Using image resources with the gICAPI web component on page 1952.

**Form definition file**

```
LAYOUT (TEXT="Image Gallery")
GRID
{
  [f1       |f2       |f3     ]
  Current:[f4     ]
  [wc1                        ]
  [                            ]
  [                            ]
  [                            ]
  [                            ]
`
COMBOBOX f1 = FORMONLY.gallery_type, NOT NULL,
INITIALIZER=display_type_init;
COMBOBOX f2 = FORMONLY.gallery_size, NOT NULL,
INITIALIZER=display_size_init;
COMBOBOX f3 = FORMONLY.aspect_ratio, NOT NULL,
INITIALIZER=aspect_ratio_init;
EDIT f4 = FORMONLY.current;
WEBCOMPONENT wc1 = FORMONLY.gallery_wc,
COMPONENTTYPE = "fglgallery",
PROPERTIES=(selection="image_selection"),
SIZEPOLICY = FIXED,
STRETCH = BOTH;

IMPORT util
IMPORT FGL fglgallery
DEFINE rec RECORD
  gallery_type INTEGER,
  gallery_size INTEGER,
  aspect_ratio DECIMAL(5,2),
  current INTEGER,
  gallery_wc STRING
END RECORD
DEFINe struct_value fglgallery.t_struct_value

MAIN
DEFINE id SMALLINT
OPEN FORM f1 FROM "simple_gallery"
DISPLAY FORM f1
OPTIONS INPUT WRAP, FIELD ORDER FORM
CALL fglgallery.initialize()
LET id = fglgallery.create("formonly.gallery_wc")

-- Image files on the server, to be handled with filenameToURI()/
FGLIMAGEPATH
-- From images-public dir:
CALL fglgallery.addImage(id, image_path("image01.jpg"), "Lake in
mountains")
CALL fglgallery.addImage(id, image_path("image02.jpg"), NULL)
CALL fglgallery.addImage(id, image_path("image03.jpg"), "Lightning")
-- From images-private dir:
CALL fglgallery.addImage(id, image_path("image10.jpg"), "Outdoor cat")
CALL fglgallery.addImage(id, image_path("image11.jpg"), NULL)

-- URLs
CALL fglgallery.addImage(id, "http://freebigpictures.com/wp-content/
uploads/2009/09/mountain-ridge.jpg", "Mountain ridge")
CALL fglgallery.addImage(id, "http://freebigpictures.com/wp-content/
uploads/2009/09/mountain-horse.jpg", "Horse in field")

LET rec.gallery_type = FGLGALLERY_TYPE_MOSAIC
LET rec.gallery_size = FGLGALLERY_SIZE_NORMAL
LET rec.aspect_ratio = 1.0
LET struct_value.current = 1
LET rec.current = struct_value.current
LET rec.gallery_wc = util.JSON.stringify(struct_value)
CALL fglgallery.display(id, rec.gallery_type, rec.gallery_size)

INPUT BY NAME rec.* ATTRIBUTES (UNBUFFERED, WITHOUT DEFAULTS)

ON CHANGE gallery_type
    CALL fglgallery.display(id, rec.gallery_type, rec.gallery_size)

ON CHANGE gallery_size
    CALL fglgallery.display(id, rec.gallery_type, rec.gallery_size)

ON CHANGE aspect_ratio
    CALL fglgallery.setImageAspectRatio(id, rec.aspect_ratio)
    CALL fglgallery.display(id, rec.gallery_type, rec.gallery_size)

ON ACTION set_current ATTRIBUTES(TEXT="Set current")
    LET struct_value.current = rec.current
    LET rec.gallery_wc = util.JSON.stringify(struct_value)

ON ACTION image_selection ATTRIBUTES(DEFAULTVIEW=NO)
    CALL util.JSON.parse( rec.gallery_wc, struct_value )
    LET rec.current = struct_value.current

ON ACTION close
    EXIT INPUT

END INPUT

CALL fglgallery.destroy(id)
CALL fglgallery.finalize()

END MAIN

FUNCTION image_path(path)
    DEFINE path STRING
    RETURN ui.Interface.filenameToURI(path)
END FUNCTION

FUNCTION display_type_init(cb)
    DEFINE cb ui.ComboBox
    CALL cb.addItem(FGLGALLERY_TYPE_MOSAIC, "Mosaic")
    CALL cb.addItem(FGLGALLERY_TYPE_LIST, "List")
    CALL cb.addItem(FGLGALLERY_TYPE_THUMBNAILS, "Thumbnails")
END FUNCTION

FUNCTION display_size_init(cb)
    DEFINE cb ui.ComboBox
    CALL cb.addItem(FGLGALLERY_SIZE_XSMALL, "X-Small")
    CALL cb.addItem(FGLGALLERY_SIZE_SMALL, "Small")
    CALL cb.addItem(FGLGALLERY_SIZE_NORMAL, "Normal")
END FUNCTION
CALL cb.addItem(FGLGALLERY_SIZE_LARGE, "Large")
CALL cb.addItem(FGLGALLERY_SIZE_XLARGE, "X-Large")
END FUNCTION

FUNCTION aspect_ratio_init(cb)
   DEFINE cb ui.ComboBox
   -- Use strings for value to match DECIMAL(5,2) formatting
   CALL cb.addItem("1.00", "1:1")
   CALL cb.addItem("1.77", "16:9")
   CALL cb.addItem("1.50", "3:2")
   CALL cb.addItem("1.33", "4:3")
   CALL cb.addItem("1.25", "5:4")
   CALL cb.addItem("0.56", "9:16")
   CALL cb.addItem("0.80", "4:5")
END FUNCTION

The fglsvgcanvas web component
The fglsvgcanvas built-in web component implements a drawing canvas for Scalable Vector Graphics content.

Note: This documentation does not explain SVG drawing principles. Before starting with fglsvgcanvas, consider learning SVG, with tutorials found on the internet. It is especially mandatory to properly understand the root SVG viewport / viewBox / preserveAspectRatio concepts.

The fglsvgcanvas built-in web component is a gICAPI web component.

Important: Since SVG allows you to specify coordinates and sizes with units such as "10em", and including the % percentage unit like in "50%", some fglsvgcanvas functions use the STRING type for parameters such as x, y, width, height. In SVG, the decimal separator for numeric values must always be a dot. When computing coordinates and sizes with numeric types such as DECIMAL or when passing decimal values directly to fglsvgcanvas functions, pay attention to numeric to string conversion. By default, the DBMONEY/DBFORMAT settings apply and can produce a comma for the decimal separator. As a general advice, when possible, define a large SVG viewBox (like "0 0 1000 1000"), in order to use only integer numbers for coordinates and sizes, or % percentage units.

The fglsvgcanvas web component HTML page is basically a simple HTML container. It is delivered with the utility library $FGLDIR/src/webcomponents/fglsvgcanvas/fglsvgcanvas.4gl, that can be used to produce SVG content.

The programming pattern is based on the built-in om.* API. Create DOM nodes with the utility functions, and construct the root <svg/> element by adding child nodes created from the fglsvgcanvas functions.
Figure 115: fglsvgcanvas web component

Defining the fglsvgcanvas web component in the form file

In the .per form definition file, define the SVG container as a WEBCOMPONENT form item with the COMPONENTTYPE attribute set to the "fglsvgcanvas" value.

Important: The form field name will be used in front calls to identify the SVG canvas.

Since the SVG canvas web component provides built-in scrollbars, the SCROLLBARS attribute can be set to NONE. Use SIZEPOLICY=FIXED and STRETCH=BOTH, to get an SVG canvas that resizes with the parent window.

Additional fglsvgcanvas web component configuration options can be defined with the PROPERTIES attribute (details will be discussed later in this topic).

For example:

```
LAYOUT
GRID
{
[ cv ]
[
 ]
[
 ]
...

ATTRIBUTES
WEBCOMPONENT cv = FORMONLY.canvas,
    COMPONENTTYPE = "fglsvgcanvas",
    PROPERTIES = ( selection="item_selection" ),
    SIZEPOLICY = FIXED,
    STRETCH = BOTH,
    SCROLLBARS = NONE;
...```
The fglsvgcanvas.4gl library

SVG can be used to draw complex content such as an agenda or a graph, with advanced SVG concepts such as CSS styles, patterns, nested <svg/> elements, etc.

To simplify SVG programming, Genero BDL provides the $FGLDIR/src/webcomponents/fglsvgcanvas/fglsvgcanvas.4gl utility library. This library implements a set of functions that produce common SVG elements.

The fglsvgcanvas.4gl library supports following SVG features:

- Attribute sets
- CSS styles
- Patterns
- Masks
- Filters
- Gradients
- Shapes (rect, circle, polygon, etc)
- Simple text, text on path, text tspan
- Animation
- Clickable elements
- Clipping paths
- RGB color utilities (shade, tint)

fglsvgcanvas library initialization and finalization

Library initialization and finalization functions are provided to prepare the library before usage, and free resources when the library is no longer needed:

```gl
CALL fglsvgcanvas.initialize()
...
CALL fglsvgcanvas.finalize()
```

For more details see initialize() on page 2346 and finalize() on page 2344.

Creating an SVG canvas handler

Before creating new SVG element, you need to create the SVG canvas handler, to get an id that will be used in subsequent fglsvgcanvas calls. Define a SMALLINT variable to hold the SVG canvas handler id that is returned by the create() function. This function takes the WEBCOMPONENT field name as attribute, to bind the form field to the SVG canvas handle:

```gl
DEFINE cid SMALLINT
...
LET cid = fglsvgcanvas.create("formonly.canvas")
```

Selecting an SVG canvas handler

The SVG canvas is identified by the id returned by the create() function. After creating an SVG canvas, it is automatically defined as the current canvas, and any subsequent calls to an fglsvgcanvas function will apply to that current canvas handler. If you want to manipulate several SVG canvases, select the current canvas with the setCurrent() function:

```gl
CALL fglsvgcanvas.setCurrent(cid)
```
The root SVG node

The root SVG DOM node is created when calling the `create()` function. However, before drawing your SVG, you need to define essential root SVG attributes.

To define the root SVG attributes, use the `setRootSVGAttributes()` function.

This function returns the root `om.DomNode` of the SVG tree:

```define```
```lang-om```
```define root_svg, n om.DomNode
... 
let root_svg = fglsvgcanvas.setRootSVGAttributes(
"myrootsvg",
"10em", "5em", -- viewport
"0 0 500 200", -- viewBox
"xMidYMid meet" -- preserveAspectRatio
)
...) 
```define```

Specify the following properties to define your root SVG element:

- the viewport defines the viewing area for the SVG image (use NULL, NULL for auto-resize; default unit is px, consider using em unit),
- the viewBox defines the internal coordinate system (with a (0,0,500,200) viewBox, point (250,100) is the middle),
- the parameters to preserve the aspect ratio.

For more details about the `<svg/>` element attributes, see the W3 SVG specification.

Destroying an SVG canvas handler

When the SVG canvas handler is no longer needed (for example, before closing the form/window displaying the corresponding `WEBCOMPONENT`), you can release resources allocated for the SVG handle by calling the `destroy()` function:

```define```
```lang-om```
```define cid SMALLINT
... 
let cid = fglsvgcanvas.create("formonly.canvas")
... 
call fglsvgcanvas.destroy( cid ) 
```define```

After calling the `destroy()` function, the SVG canvas handler that was created before the destroyed handler, will be set as the new current SVG canvas:

```define```
```lang-om```
```define cid1, cid2 SMALLINT
... 
let cid1 = fglsvgcanvas.create("formonly.canvas1") -- current canvas is cid1
... 
let cid2 = fglsvgcanvas.create("formonly.canvas2") -- current canvas is cid2
... 
call fglsvgcanvas.destroy( cid2 ) -- current canvas is cid1
... 
call fglsvgcanvas.destroy( cid1 ) -- no current canvas 
```define```

Building the SVG DOM tree

After creating the root SVG DOM node with the `setRootSVGAttributes()` function, create other DOM element with fglsvgcanvas functions, and append the child nodes to the root element or sub-elements:

```define```
```lang-om```
```define root_svg, n, g om.DomNode
... 
let n = fglsvgcanvas.svg( ... ) -- creates an <svg/> sub-node. 
```define```
CALL root_svg.appendChild( n )
...
LET g = fglsvgcanvas.g( ... ) -- creates a <g/> sub-node.
CALL n.appendChild( g )
...

Cleaning the SVG canvas

To clean up the SVG canvas content, use the clean() function:

CALL fglsvgcanvas.clean( cid )

Defining CSS styles

SVG supports CSS styling. Styles must be defined in the <defs/> element, and can then be referenced in SVG drawing elements by using the class attribute.

To create styles, start by defining a set of SVG attributes with an om.SaxAttributes object. For attribute names, use the predefined SVGATT_* constants available in the fglsvgcanvas library.

Tip: Define a dynamic array of om.SaxAttributes, to define several reusable attribute sets.

CONSTANT COLORS_OCEAN = 1
CONSTANT COLORS_SAHARA = 2
DEFINE attr DYNAMIC ARRAY OF om.SaxAttributes
...
LET attr[COLORS_OCEAN] = om.SaxAttributes.create()
CALL attr[COLORS_OCEAN].addAttribute(SVGATT_FILL, "cyan")
CALL attr[COLORS_OCEAN].addAttribute(SVGATT_FILL_OPACITY, "0.3")
CALL attr[COLORS_OCEAN].addAttribute(SVGATT_STROKE, "blue")
CALL attr[COLORS_OCEAN].addAttribute(SVGATT_STROKE_WIDTH, "5")
CALL attr[COLORS_OCEAN].addAttribute(SVGATT_STROKE_OPACITY, "0.3")

LET attr[COLORS_SAHARA] = om.SaxAttributes.create()
CALL attr[COLORS_SAHARA].addAttribute(SVGATT_FILL, "yellow")
...

An SVG attribute set defined in an om.SaxAttributes object can be used in different manners:

1. To explicitly set attributes in an SVG DOM node, with the setAttributes() function.
2. To define a CSS style with a selector and a list of name:value; pairs, with the styleDefinition() function.
3. To set an inline-style in an DOM node, defining a list of name:value; pairs, with the styleAttributeList() function.

To create a set of CSS styles, create a <defs/> SVG element, containing a <style/> element including your attributes sets.

The <style/> element is created with the styleList() function, and each CSS style string is created with the styleDefinition() function, from the om.SaxAttributes objects:

DEFINE defs om.DomNode
...
LET defs = fglsvgcanvas.defs( NULL )
CALL defs.appendChild( fglsvgcanvas.styleList( 
  fglsvgcanvas.styleDefinition(".style_ocean",attr[COLORS_OCEAN])
  ||
  fglsvgcanvas.styleDefinition(".style_sahara",attr[COLORS_SAHARA])
) )
Then, you can, for example, define a `<rect/>` element that references the style defined with in the `<defs/>` node:

```javascript
DEFINE r om.DomNode
LET r = fglsvgcanvas.rect(10,10,30,40,NULL,NULL)
CALL r.setAttribute(SVGATT_CLASS, "style_ocean")
```

### Displaying the SVG content

After creating the SVG content, it must be sent to the front-end for rendering.

To display the SVG content to the `WEBCOMPONENT` field associated with the SVG canvas handle, use the `display()` function:

```javascript
CALL fglsvgcanvas.display( cid )
```

### Detecting SVG element selection

To enable clickable SVG elements, define the `WEBCOMPONENT` form field with a "selection" property in the `PROPERTIES` attribute. This property defines the action that will be fired when the user clicks on an SVG element. Use lowercase action names:

```javascript
WEBCOMPONENT cv = FORMONLY.canvas,
  ...
  PROPERTIES = (selection="item_selection"),
  ...
```

Define clickable SVG elements by using the `<g/>` SVG grouping element contain child elements.

**Note:** The child elements of the clickable group must not use the `fill:none` style, otherwise they are not clickable.

The group element must have the `SVGATT_ONCLICK"onclick"` attribute set to `SVGVAL_ELEM_CLICKED/"elem_clicked(this)"`: The `elem_clicked(this)` JavaScript function is predefined in `fglsvgcanvas.html` and will be fired the action defined by the `selection` property defined by the `WEBCOMPONENT` form field.

The `<g/>` group element must be defined with an "id" attribute, that will be used to identify the clicked elements:

```javascript
DEFINE root_svg, g, c om.DomNode
  ...
  CALL root_svg.appendChild( g := fglsvgcanvas.g("shape_1") )
  CALL g.setAttribute(SVGATT_ONCLICK,SVGVAL_ELEM_CLICKED)
  CALL g.appendChild( c := fglsvgcanvas.circle(10,20,3) )
  CALL c.setAttribute(SVGATT_STYLE, 'stroke:gray;fill:blue;fill-opacity:0.3' )
  CALL g.appendChild( c := fglsvgcanvas.circle(14,12,2) )
  CALL c.setAttribute(SVGATT_STYLE, 'stroke:gray;fill:yellow;fill-opacity:0.8' )
```

**Tip:** Note that child elements can be created, assigned to a variable with the `:=` operator and the resulting expression can be directly passed as `appendChild()` parameter.

```javascript
CALL root_svg.appendChild( g := fglsvgcanvas.g("shape_1") )
```
In the program code, detect SVG element selection with an **ON ACTION** handler using the action name that matches the "selection" property of the **WEBCOMPONENT** field. When the action is fired, the id of the selected SVG element is provided in the **WEBCOMPONENT** field value:

```plaintext
DEFINE rec RECORD
  canvas STRING, -- Variable bound to the WEBCOMPONENT form field
  ...
END RECORD
...
INPUT BY NAME rec.* ATTRIBUTES(UNBUFFERED)
  ON ACTION item_selection ATTRIBUTES(DEFAULTVIEW = NO)
    DISPLAY "Clicked element: ", rec.canvas
    ...
```

Information about the clicked SVG element is provided in JSON notation, for example:

```plaintext
{"id":"shape_1","source":"action","action":"item_selection"}
```

**Note:** It is also possible to use the `fglsvgcanvas.getItemId()` function to get the id of the clicked SVG element. However, this produces a front call that can be avoided, since the element id is available in the **WEBCOMPONENT** field value. The `getItemId()` function should only be used when the **WEBCOMPONENT** field does not have the focus, for example to handle mouse hovering events with `mouse_event_focus = false`.

For a complete example, see Example 2: Basic SVG shapes with fglsvgcanvas on page 2000.

**Detecting mouse hovering SVG events**

SVG elements can be defined with the mouse hovering events "onmouseover" and "onmouseout", respectively defined as the `SVGATT_ONMOUSEOVER` and `SVGATT_ONMOUSEOUT` constants in `fglsvgcanvas.4gl` for convenience and code readability.

**Tip:** If you want to implement display effects only, consider using well-known SVG + JavaScript techniques to modify element attributes directly in the web browser context, and thus avoid unnecessary network round-trips. For example, to produce some responsive rendering effect when the mouse goes over an SVG element, you can do following:

```plaintext
DEFINE n om.DomNode
LET n = fglsvgcanvas.rect(50,50,20,30,2,2)
CALL n.setAttribute(SVGATT_ONMOUSEOVER,  "evt.target.setAttribute('opacity', '0.5');")
CALL n.setAttribute(SVGATT_ONMOUSEOUT,  "evt.target.setAttribute('opacity', '1.0');")
```

The fglsvgcanvas web component can be configured to bind the `onmouseover/onmouseout` SVG mouse hovering events to **ON ACTION** handlers and trigger code, when the mouse goes over the SVG elements.

**Important:** When SVG element selection is enabled, the corresponding action has a higher importance than mouse hovering actions: Depending on the mouse hovering timeout (`mouse_event_timeout`), a click on an SVG element may cancel the mouse over / mouse out actions, even if corresponding SVG `onmouseover` and `onmouseout` events occurred respectively before and after the `onclick` SVG event.

The next code example defines the web component field properties to handle mouse hovering SVG events:

```plaintext
WEBCOMPONENT cv=FORMONLY.canvas,
  ...
  PROPERTIES=
    ...
    mouse_over = "item_mouse_over",
    mouse_out = "item_mouse_out",
    mouse_event_timeout = 600,
    mouse_event_focus = false
  ),
```
Mouse hovering properties description:

1. The `mouse_over = "action-name"` property identifies the ON ACTION `action-name` handler to be triggered, when the `onmouseover` SVG event occurs. Use lowercase to define mouse hovering action names. This property is mandatory to catch mouse hovering events.

2. The `mouse_out = "action-name"` property identifies the ON ACTION `action-name` handler to be triggered, when the `onmouseout` SVG event occurs. Use lowercase to define mouse hovering action names. The property is typically used to reset mouse hovering management in the program code (the mouse goes away from that SVG element).

3. The `mouse_event_timeout = milliseconds` is used to avoid network clogging: This property defines the number of milliseconds to wait before sending the dialog action, after an SVG mouse event has occurred. A JavaScript `setTimeout()` timer is created with this value. The timer is re-initialized each time the event occurs. If not defined, default is 500 milliseconds.

4. The `mouse_event_focus = [true | false]` specifies if the web component field must have the focus, in order to fire the mouse hovering actions. If not defined, default is `true`.

When creating your SVG elements, bind mouse hovering SVG events to predefined JavaScript function corresponding to the SVG event. The functions take the current SVG element (`this`) as parameter.

- The `SVGATT_ONMOUSEOVER/"onmouseover"` SVG event must be bound to `SVGVAL_ELEM_MOUSE_OVER/"elem_mouse_over(this)"`
- The `SVGATT_ONMOUSEOUT/"onmouseout"` SVG event must be bound to `SVGVAL_ELEM_MOUSE_OUT/"elem_mouse_out(this)"

For example:

```javascript
DEFINE n om.DomNode
...
LET n = fglsvgcanvas.rect(x, y, w, h, NULL, NULL)
CALL n.setAttribute(SVGATT_ONMOUSEOVER, SVGVAL_ELEM_MOUSE_OVER)
CALL n.setAttribute(SVGATT_ONMOUSEOUT, SVGVAL_ELEM_MOUSE_OUT)
```

In the dialog code, implement the ON ACTION handlers to execute code, when the actions are fired.

**Note:** If the mouse hovering action requires field focus (`mouse_event_focus=true`), the involved SVG element id is available in the field value, like for the item selection action. If the mouse hovering action can be fired when the web component field does not have the focus, use the `fglsvgcanvas.getItemId()` function to get the id of the SVG element.

```javascript
...  
ON ACTION item_mouse_over ATTRIBUTES(DEFAULTVIEW = NO)
MESSAGE SFMT("%1 : mouse over item : %2",
   CURRENT HOUR TO FRACTION(5),
   fglsvgcanvas.getItemId(cid) )

ON ACTION item_mouse_out ATTRIBUTES(DEFAULTVIEW = NO)
MESSAGE ""
...
```

For a complete example, see Example 2: Basic SVG shapes with fglsvgcanvas on page 2000.

**Getting the bounding box of an SVG element**

After rendering your SVG, it is possible to get the bounding box of an element.

With elements such as SVG text using a specific font, it is difficult to compute the bounding box of the text, until it has been rendered.
First define a variable with the `fglsvgcanvas.t_svg_rect` type, then (after displaying the SVG with `fglsvgcanvas.display(cid)`), call the `fglsvgcanvas.getBBox(cid, element-id)` function, to get the bounding box of the element identified by `element-id`:

```plaintext
DEFINE rect fglsvgcanvas.t_svg_rect
...
ON ACTION get_bbox
    CALL fglsvgcanvas.getBBox(cid, "label_23") RETURNING rect.*
    DISPLAY rect.x, rect.y, rect.width, rect.height
```

The bounding box coordinates and size are returned in the current user space.

**Note:**

If the bounding box is required to place and size SVG element based on the position and size of other elements, consider using SVG scripting instead of the `getBBox()` function: this will avoid several network roundtrips to render the final SVG image.

For example:

```xml
<svg version="1.1" baseProfile="full"
     xmlns="http://www.w3.org/2000/svg"
     width="500px" height="500px" viewBox="0 0 2000 2000"
     onload="setup()"
>
    <script type="text/ecmascript">
        // <![CDATA[
        function setup_rect(box,bbox){
            box.setAttributeNS(null, "x", bbox.x - 2);
            box.setAttributeNS(null, "y", bbox.y - 2);
            box.setAttributeNS(null, "width", bbox.width + 4);
            box.setAttributeNS(null, "height", bbox.height + 4);
        }
        function setup(evt){
            setup_rect( document.getElementById("label1Box"), label1.getBBox() );
        }
        // ]]>
    </script>

    <g id="label1">
        <text x="150" y="250" font-family="Verdana" font-size="55">Hello everybody!</text>
    </g>

    <rect id="label1Box" stroke="red" stroke-width="3px" fill="none"/>
</svg>
```

**Related reference**

- **fglsvgcanvas: SVG drawing module** on page 2332

**Examples**

- fglsvgcanvas built-in web component usage examples.
- **Example 1: Analog clock with fglsvgcanvas**
  
  Example using the fglsvgcanvas web component to display an analog clock.
Form definition file `svgclock.per`:

```
LAYOUT
GRID
{
  [cv                      ]
  [               ]
  [               ]
  [               ]
  [               ]
  [               ]
  [               ]
  [               ]
}
END
END

ATTRIBUTES

WEBCOMPONENT cv=FORMONLY.canvas,
    COMPONENTTYPE="fglsvgcanvas",
    SIZEPOLICY=FIXED,
    STRETCH=BOTH,
    SCROLLBARS=NONE
;
END
```

Program file `svgclock.4gl`:

```
IMPORT FGL fglsvgcanvas

MAIN
    CONSTANT HAND_H = 1
    CONSTANT HAND_M = 2
    CONSTANT HAND_S = 3
    CONSTANT FACE_1 = 4
```
CONSTANT FACE_2 = 5
CONSTANT FACE_3 = 6
CONSTANT FACE_4 = 7

DEFINE cid SMALLINT,
  root_svg om.DomNode,
  attr DYNAMIC ARRAY OF om.SaxAttributes,
  h,th,m,tm,s,ts SMALLINT,
  defs, gr, g, g2, g3, n om.DomNode

OPEN FORM f1 FROM "svgclock"
DISPLAY FORM f1

CALL fglsvgcanvas.initialize()
LET cid = fglsvgcanvas.create("formonly.canvas")
LET root_svg = fglsvgcanvas.setRootSVGAttributes( NULL,
  Null, Null,
  "0 0 270 270",
  "xMidYMid meet"
)
CALL root_svg.setAttribute(SVGATT_CLASS,"root_svg")

LET h = CURRENT HOUR TO HOUR || " "
IF h > 12 THEN LET h = h - 12 END IF
LET m = CURRENT MINUTE TO MINUTE || " "
LET s = CURRENT SECOND TO SECOND || " "
LET ts = 6*s
LET tm = (m+s/60)*6
LET th = (h+m/60+s/3600)*30

LET attr[HAND_H] = om.SaxAttributes.create()
CALL attr[HAND_H].addAttribute(SVGATT_STROKE, "blue")
CALL attr[HAND_H].addAttribute(SVGATT_STROKE_WIDTH, "5")
CALL attr[HAND_H].addAttribute(SVGATT_STROKE_LINECAP, "round")

LET attr[HAND_M] = om.SaxAttributes.create()
CALL attr[HAND_M].addAttribute(SVGATT_STROKE, "navy")
CALL attr[HAND_M].addAttribute(SVGATT_STROKE_WIDTH, "4")
CALL attr[HAND_M].addAttribute(SVGATT_STROKE_LINECAP, "round")

LET attr[HAND_S] = om.SaxAttributes.create()
CALL attr[HAND_S].addAttribute(SVGATT_STROKE, "red")
CALL attr[HAND_S].addAttribute(SVGATT_STROKE_WIDTH, "2")
CALL attr[HAND_S].addAttribute(SVGATT_STROKE_LINECAP, "round")

LET attr[FACE_1] = om.SaxAttributes.create()
CALL attr[FACE_1].addAttribute(SVGATT_FILL, "none")
CALL attr[FACE_1].addAttribute(SVGATT_STROKE, "gray")
CALL attr[FACE_1].addAttribute(SVGATT_STROKE_WIDTH, "1")

LET attr[FACE_2] = om.SaxAttributes.create()
CALL attr[FACE_2].addAttribute(SVGATT_FILL, "none")
CALL attr[FACE_2].addAttribute(SVGATT_STROKE, "gray")
CALL attr[FACE_2].addAttribute(SVGATT_STROKE_WIDTH, "4")

LET attr[FACE_3] = om.SaxAttributes.create()
CALL attr[FACE_3].addAttribute(SVGATT_FILL, "none")
CALL attr[FACE_3].addAttribute(SVGATT_STROKE, "lightBlue")
CALL attr[FACE_3].addAttribute(SVGATT_STROKE_WIDTH, "5")
CALL attr[FACE_3].addAttribute(SVGATT_STROKE_DASHARRAY, "2,8.471976")
CALL attr[FACE_3].addAttribute(SVGATT_TRANSFORM, "rotate(-.873)")

LET attr[FACE_4] = om.SaxAttributes.create()
CALL attr[FACE_4].addAttribute(SVGATT_FILL, "none")
CALL attr[FACE_4].addAttribute(SVGATT_STROKE, "blue")
CALL attr[FACE_4].addAttribute(SVGATT_STROKE_WIDTH, "11")
CALL attr[FACE_4].addAttribute(SVGATT_STROKE_DASHARRAY, "4,46.789082")
CALL attr[FACE_4].addAttribute(SVGATT_TRANSFORM, "rotate(-1.5)"

LET defs = fglsvgcanvas.defs( NULL )
CALL root_svg.appendChild( defs )

LET gr = fglsvgcanvas.radialGradient( "gradient_1", NULL, NULL, "5\%", "5\%", "65\%", "pad", NULL, NULL )
CALL gr.appendChild( fglsvgcanvas.stop( "0\%", "gray", 0.4 ) )
CALL gr.appendChild( fglsvgcanvas.stop( "100\%", "navy", 0.7 ) )
CALL defs.appendChild( gr )

LET g = fglsvgcanvas.g( "clock" )
CALL g.setAttribute(SVGATT_TRANSFORM,"translate(135,135)"")
CALL g.appendChild( fglsvgcanvas.title("Time goes and goes...") )
CALL root_svg.appendChild( g )
LET g2 = fglsvgcanvas.g( NULL )
CALL g.appendChild( g2 )
CALL g2.appendChild( n:=fglsvgcanvas.circle(0,0,110) )
CALL n.setAttribute(SVGATT_STYLE, "fill:url(#gradient_1);")
CALL g2.appendChild( n:=fglsvgcanvas.circle(0,0,115) )
CALL n.setAttribute(SVGATT_STYLE, fglsvgcanvas.styleAttributeList(attr[FACE_1]))
CALL g2.appendChild( n:=fglsvgcanvas.circle(0,0,108) )
CALL n.setAttribute(SVGATT_STYLE, fglsvgcanvas.styleAttributeList(attr[FACE_2]))
CALL g2.appendChild( n:=fglsvgcanvas.circle(0,0,100) )
CALL n.setAttribute(SVGATT_STYLE, fglsvgcanvas.styleAttributeList(attr[FACE_3]))
CALL g2.appendChild( n:=fglsvgcanvas.circle(0,0, 97) )
CALL n.setAttribute(SVGATT_STYLE, fglsvgcanvas.styleAttributeList(attr[FACE_4]))

LET g2 = fglsvgcanvas.g( NULL )
CALL g2.setAttribute(SVGATT_TRANSFORM,"rotate(180)"")
CALL g.appendChild( g2 )

LET g3 = fglsvgcanvas.g( "hour" )
CALL g3.setAttribute(SVGATT_TRANSFORM, SFMT("rotate(%1)",th))
CALL g2.appendChild( g3 )
CALL g3.appendChild( n:=fglsvgcanvas.line(0,0,0,75) )
CALL n.setAttribute(SVGATT_STYLE, fglsvgcanvas.styleAttributeList(attr[HAND_H]))
CALL g3.appendChild( fglsvgcanvas.animateTransform("transform", "XML", "rotate", NULL, NULL, "360", NULL, "12h", "infinite") )
CALL g3.appendChild( fglsvgcanvas.circle(0,0,7) )

LET g3 = fglsvgcanvas.g( "minute" )
CALL g3.setAttribute(SVGATT_TRANSFORM, SFMT("rotate(%1)",tm))
CALL g2.appendChild( g3 )
CALL g3.appendChild( n:=fglsvgcanvas.line(0,0,0,93) )
CALL n.setAttribute(SVGATT_STYLE, fglsvgcanvas.styleAttributeList(attr[HAND_M]))
CALL g3.appendChild( fglsvgcanvas.animateTransform("transform", "XML", "rotate", NULL, NULL, "360", NULL, "60min", "infinite") )
CALL g3.appendChild( n:=fglsvgcanvas.circle(0,0,6) )
CALL n.setAttribute(SVGATT_STYLE,'fill="red"')
Example 2: Basic SVG shapes with fglsvgcanvas

Example using the fglsvgcanvas web component to draw basic shapes and detect mouse events.

```
LET g3 = fglsvgcanvas.g( "second" )
CALL g3.setAttribute(SVGATT_TRANSFORM, SFMT("rotate(%1)",ts))
CALL g2.appendChild( g3 )
CALL g3.appendChild( n:=fglsvgcanvas.line(0,-20,0,102) )
CALL n.setAttribute(SVGATT_STYLE, fglsvgcanvas.styleAttributeList(attr[HAND_S]))
CALL g3.appendChild( fglsvgcanvas.animateTransform("transform", "XML", "rotate", NULL, NULL, "360", NULL, "60s", "indefinite") )
CALL g3.appendChild( n:=fglsvgcanvas.circle(0,0,4) )
CALL n.setAttribute(SVGATT_STYLE,'fill="blue"')
CALL fglsvgcanvas.display(cid)

MENU "test"
  ON ACTION quit ATTRIBUTES(TEXT="Quit")
    EXIT MENU
END MENU

CALL fglsvgcanvas.destroy(cid)
CALL fglsvgcanvas.finalize()
END MAIN
```

Figure 117: fglsvgcanvas web component - basics

Form definition file svgbasics.per:

```
LAYOUT
GRID
{
 [cv ]
 [ ]
 [ ]
 [ ]
}
```
IMPORT util
IMPORT FGL fglsvgcanvas

MAIN

CONSTANT CB = 1
CONSTANT CY = 2
CONSTANT CG = 3
CONSTANT T1 = 4

DEFINE rec RECORD
    canvas STRING,
    info STRING
END RECORD

DEFINE cid SMALLINT,
    root_svg om.DomNode,
    attr DYNAMIC ARRAY OF om.SaxAttributes,
    defs, g, n om.DomNode

OPEN FORM f1 FROM "svgbasics"
DISPLAY FORM f1

CALL fglsvgcanvas.initialize()
LET cid = fglsvgcanvas.create("formonly.canvas")
LET root_svg = fglsvgcanvas.setRootSVGAttributes( NULL,
    NULL, NULL,
    "0 0 500 500",
    "xMidYMid meet"
)

CALL root_svg.setAttribute(SVGATT_CLASS,"root_svg")

LET attr[CB] = om.SaxAttributes.create()
CALL attr[CB].addAttribute(SVGATT_FILL, "cyan")
CALL attr[CB].addAttribute(SVGATT_FILL_OPACITY, "0.3")
CALL attr[CB].addAttribute(SVGATT_STROKE, "blue")
CALL attr[CB].addAttribute(SVGATT_STROKE_WIDTH, "5")
CALL attr[CB].addAttribute(SVGATT_STROKE_OPACITY, "0.3")

LET attr[CY] = om.SaxAttributes.create()
CALL attr[CY].addAttribute(SVGATT_FILL, "yellow")
CALL attr[CY].addAttribute(SVGATT_FILL_OPACITY, "0.8")
CALL attr[CY].addAttribute(SVGATT_STROKE, "orange")
CALL attr[CY].addAttribute(SVGATT_STROKE_WIDTH, "4")
CALL attr[CY].addAttribute(SVGATT_STROKE_OPACITY, "0.8")

LET attr[CG] = om.SaxAttributes.create()
CALL attr[CG].addAttribute(SVGATT_FILL, "green")
CALL attr[CG].addAttribute(SVGATT_FILL_OPACITY, "0.8")
CALL attr[CG].addAttribute(SVGATT_STROKE, "darkGreen")
CALL attr[CG].addAttribute(SVGATT_STROKE_WIDTH, "4")
CALL attr[CG].addAttribute(SVGATT_STROKE_OPACITY, "0.8")

LET attr[T1] = om.SaxAttributes.create()
CALL attr[T1].addAttribute(SVGATT_STROKE, "gray")
CALL attr[T1].addAttribute(SVGATT_STROKE_WIDTH, "1")
CALL attr[T1].addAttribute(SVGATT_STROKE_LINECAP, "round")
CALL attr[T1].addAttribute(SVGATT_FILL, "blue")
CALL attr[T1].addAttribute(SVGATT_FONT_FAMILY, "Sans")
CALL attr[T1].addAttribute(SVGATT_FONT_SIZE, "24px")

LET defs = fglsvgcanvas.defs(NULL)
CALL defs.appendChild(fglsvgcanvas.styleList(
  fglsvgcanvas.styleDefinition(".style_1", attr[CB])
  fglsvgcanvas.styleDefinition(".style_2", attr[CY])
  fglsvgcanvas.styleDefinition(".style_3", attr[CG])
  fglsvgcanvas.styleDefinition(".style_4", attr[T1])
))
CALL root_svg.appendChild(defs)

CALL root_svg.appendChild(fglsvgcanvas.text(30, 40, "Basic fglsvgcanvas example...", "style_4"))

CALL root_svg.appendChild( n:=fglsvgcanvas.rect(200,200,350,150,10,10) )
CALL n.setAttribute("id","R1")
CALL n.appendChild(fglsvgcanvas.title("Rectangle 1"))
CALL n.setAttribute(SVGATT_STYLE,'stroke:black;fill:orange')

CALL root_svg.appendChild( n:=fglsvgcanvas.circle(40,120,50) )
CALL n.setAttribute("id","C1")
CALL n.appendChild( fglsvgcanvas.title("Circle 1 (with mouse hovering effects)") )
CALL n.setAttribute(SVGATT_CLASS, "style_1")
CALL n.setAttribute(SVGATT_ONMOUSEOVER,
  "evt.target.setAttribute('opacity', '0.5');")
CALL n.setAttribute(SVGATT_ONMOUSEOUT,
  "evt.target.setAttribute('opacity', '1.0');")

CALL root_svg.appendChild( n:=fglsvgcanvas.ellipse(50,400,50,30) )
CALL n.setAttribute("id","E1")
CALL n.appendChild( fglsvgcanvas.title("Ellipse 1 (clickable)") )
CALL n.setAttribute(SVGATT_CLASS, "style_3")
CALL n.setAttribute(SVGATT_ONCLICK, SVGVAL_ELEM_CLICKED)
CALL root_svg.appendChild( n:=fglsvgcanvas.rect(400,100,150,50,5,10) )
CALL n.setAttribute("id","R2")
CALL n.appendChild( fglsvgcanvas.title("Rectangle 2 (with mouse hovering actions)"))
CALL n.setAttribute(SVGATT_CLASS, "style_2")
CALL n.setAttribute(SVGATT_ONMOUSEOVER, SVGVAL_ELEM_MOUSE_OVER)
CALL n.setAttribute(SVGATT_ONMOUSEOUT, SVGVAL_ELEM_MOUSE_OUT)

CALL root_svg.appendChild( g:=fglsvgcanvas.g( "G1" ) )
CALL g.appendChild( fglsvgcanvas.title("Group of elements 1") )
CALL g.setAttribute(SVGATT_ONCLICK, SVGVAL_ELEM_CLICKED)
CALL g.setAttribute(SVGATT_ONMOUSEOVER, SVGVAL_ELEM_MOUSE_OVER)
CALL g.setAttribute(SVGATT_ONMOUSEOUT, SVGVAL_ELEM_MOUSE_OUT)
CALL g.appendChild( n:=fglsvgcanvas.rect(380,390,100,100,NULL,NULL) )
CALL n.setAttribute(SVGATT_STYLE,'stroke:black;fill:none' )
CALL g.setAttribute(SVGATT_STYLE,'stroke:black;fill:#AA2244' )
CALL g.setAttribute(SVGATT_ONCLICK, SVGVAL_ELEM_CLICKED)
CALL g.setAttribute(SVGATT_ONMOUSEOVER, SVGVAL_ELEM_MOUSE_OVER)
CALL g.setAttribute(SVGATT_ONMOUSEOUT, SVGVAL_ELEM_MOUSE_OUT)
CALL g.appendChild( n:=fglsvgcanvas.circle(420,450,30) )
CALL n.setAttribute(SVGATT_CLASS, "style_3")

CALL root_svg.appendChild( n:=fglsvgcanvas.rect(100,150,50,80,20,20) )
CALL n.setAttribute("id","R3")
CALL n.appendChild( fglsvgcanvas.title("Rectangle 3") )
CALL n.setAttribute(SVGATT_STYLE,'stroke:black;fill:#AA2244' )
CALL n.setAttribute(SVGATT_ONCLICK, SVGVAL_ELEM_CLICKED)
CALL n.setAttribute(SVGATT_ONMOUSEOVER, SVGVAL_ELEM_MOUSE_OVER)
CALL n.setAttribute(SVGATT_ONMOUSEOUT, SVGVAL_ELEM_MOUSE_OUT)
CALL n.appendChild( fglsvgcanvas.animateTransform("transform", "XML", "rotate", 0 100 200, 360 200 150, NULL, "0s", "20s", "indefinite") )

CALL fglsvgcanvas.display(cid)

INPUT BY NAME rec.* ATTRIBUTES(UNBUFFERED)

ON ACTION item_selection ATTRIBUTES(DEFAULTVIEW=NO)
LET rec.info = SFMT("Selected item: %1", rec.canvas)

ON ACTION item_mouse_over ATTRIBUTES(DEFAULTVIEW = NO)
LET rec.info = SFMT("%1 : mouse over item : %2", CURRENT HOUR TO FRACTION(5), fglsvgcanvas.getItemId(cid) )

ON ACTION item_mouse_out ATTRIBUTES(DEFAULTVIEW = NO)
LET rec.info = NULL

END INPUT

CALL fglsvgcanvas.destroy(cid)
CALL fglsvgcanvas.finalize()

END MAIN
Dynamic Dialogs

Dialogs can be created at runtime with the `ui.Dialog` class.

**Understanding dynamic dialogs**

This section provides basics about dynamic dialogs.

The `ui.Dialog` class can create dialog objects at runtime, to implement generic code controlling forms that are created at runtime, when the data structure is not known at compile time.

Unlike static dialog instructions, dynamic dialogs do not require a data model (that is a program variables containing the values for fields). Dynamic dialogs are created with a list for field definitions that is built at runtime, no static (RECORD) structure is required to instantiate a dynamic dialog.

**Important:** Dynamic dialogs are provided to resolve specific needs, like implementing a generic zoom window to select a record in a list, and control forms generated at runtime. This feature is not a replacement for regular "static" dialog instructions, used to control the forms defined in form specification files.

Dynamic dialogs use by default the `UNBUFFERED` mode and `WITHOUT DEFAULTS`. For more details, see *Controlling field values* on page 2007.

Dynamic dialogs can be used in conjunction with `base.SqlHandle` objects, to get database table column information in order to build forms dynamically and define the field list.

**Create forms dynamically**

Dynamic dialogs are typically used with forms that are generated at runtime.

**Purpose of dynamic forms**

Before you instantiate a new `ui.Dialog` object, you must load an existing compiled `.42f` form, or create a new form dynamically in your program.

Since dynamic dialogs are build at runtime (because form fields are not known at compiled time), the corresponding form is also created (or completed) at runtime.

**Note:** The current form (in the current window) is automatically attached to the new created dialog.

**Dynamic form creation with `createForm()`**

Forms built at runtime must be created with the `ui.Window.createForm()` method, and must contain a valid definition with layout containers, form fields, and screen records.

**Note:** See Genero BDL demos for a complete example of form creation at runtime.

The `createForm()` method will be invoked by using the current window. For the main form of the program, use directly the (empty) SCREEN window. For child windows, create the windows without a form by using following syntax:

```
OPEN WINDOW w1 WITH 1 ROWS, 1 COLUMNS
```

Assuming that there is a current empty window, you can then create the `ui.Form` object, to finally get the `om.DomNode` object to build your form:

```
DEFINE w ui.Window,
   f ui.Form,
   n om.DomNode
LET w = ui.Window.getCurrent()
LET f = w.createForm("myform")
LET n = f.getNode()
...
Use om classes, to build your form dynamically. A good practice in creating dynamic forms is to write first a .per file, that implements a static version of one of the forms you want to build at runtime. Compile the .per to a .42f and inspect the generated XML file, to understand the structure of the form file.

**COMBOBOX initializers**

Combobox initializers are called when executing the DISPLAY FORM or OPEN WINDOW WITH FORM instructions.

When creating a form dynamically with COMBOBOX fields, the initialization functions are not called. The combobox items must be created as part of the combobox node.

**Related concepts**
The om package on page 2577

These topics cover the built-in classes of the om package

**Instantiate a dynamic dialog**
The dynamic dialogs needs to be created with specific ui.Dialog methods.

**Defining the dialog object variable**

To reference the dialog object, first declare a variable with the type ui.Dialog:

```plaintext
DEFINE d ui.Dialog
```

**Defining field names and types**

The dynamic dialog creation methods take the list of field definitions as parameter, as a dynamic array with a record structure using two members to define the field name and data type.

Define a dynamic array with the following structure:

```plaintext
DEFINE fields DYNAMIC ARRAY OF RECORD
  name STRING,
  type STRING
END RECORD
```

The field definition array will identify form fields and the data types to be used to store the values.

The data types are provided as strings, using the same syntax as a regular Genero type:

```plaintext
LET fields[1].name = "formonly.cust_id"
LET fields[1].type = "INTEGER"
LET fields[2].name = "formonly.cust_name"
LET fields[2].type = "VARCHAR(50)"
LET fields[3].name = "formonly.cust_modts"
LET fields[3].type = "DATETIME YEAR TO FRACTION(5)"
```

**Note:** The type names used by the dynamic dialog API is the same as the type names returned by the base.SqlHandle.getResultType() method.

**Instantiate the dynamic dialog object**

When the list of field definition is complete, the next step is to create the dynamic dialog object.

When instantiating a dynamic dialog, the current form is automatically attached to it.

- To create a dynamic dialog handling simple record input (like INPUT BY NAME):

  ```plaintext
  LET d = ui.Dialog.createInputByName(fields)
  ```
For more details, see `ui.Dialog.createInputByName` on page 2517.

- To create a dynamic dialog handling query by example (like `CONSTRUCT`):

```plaintext
LET d = ui.Dialog.createConstructByName(fields)
```

For more details, see `ui.Dialog.createConstructByName` on page 2515.

- To create a dynamic dialog handling a read-only list (like `DISPLAY ARRAY`):

```plaintext
LET d = ui.Dialog.createDisplayArrayTo(fields, "sr_custlist")
```

**Note:** The `createDisplayArrayTo()` method requires the name of the screen record used to group form fields, as defined in the `INSTRUCTIONS` section of the `.per` form file.

For more details, see `ui.Dialog.createDisplayArrayTo` on page 2515.

- To create a dynamic dialog handling an editable list (like `INPUT ARRAY FROM`):

```plaintext
LET d = ui.Dialog.createInputArrayFrom(fields, "sr_custlist")
```

**Note:** The `createInputArrayFrom()` method requires the name of the screen record used to group form fields, as defined in the `INSTRUCTIONS` section of the `.per` form file.

For more details, see `ui.Dialog.createInputArrayFrom` on page 2516.

- To create a dynamic dialog handling a multiple dialog (like `DIALOG / END DIALOG`):

```plaintext
LET d = ui.Dialog.createMultipleDialog()
```

Then add sub-dialogs with methods such as `ui.Dialog.addInputByName`.

For more details, see `ui.Dialog.createMultipleDialog` on page 2518.

**Defining dialog triggers**

Dialog triggers must be implemented in a `WHILE` loop.

**Dialog trigger identification**

The code implementing dynamic dialogs must be generic. Triggers like `ON ACTION myaction` are identified as a simple string.

**Implicit standard triggers**

Standard dialog triggers such as `BEFORE INPUT`, `BEFORE FIELD field-name`, `AFTER ROW` do not need to be added when creating a dynamic dialog: These triggers are implicitly declared. To see a reaction to standard triggers, add a test for the trigger in the event loop.

**Adding user-defined triggers**

Dynamic dialogs can be configured with user-defined triggers, for example to execute code when a specific action is fired.

After creating the dialog object, add user-defined triggers with the `ui.Dialog.addTrigger()` method:

```plaintext
DEFINE d ui.Dialog
...;
CALL d.addTrigger("ON ACTION print")
CALL d.addTrigger("ON DELETE")
...;
```

Note that some triggers must be identified with the user-defined action name, as in "ON ACTION print".

User-defined triggers will then be handled in the dynamic dialog loop, when the event occurs.
Handling dialog events

To implement the "body" of a dynamic dialog, mix a WHILE loop with the `ui.Dialog.nextEvent()` method, to handle dialog events.

The WHILE loop will act as the main event handler of your dynamic dialog, and will loop, waiting for dialog events until you explicitly exit the loop with an EXIT WHILE instruction.

The `nextEvent()` method can return NULL in case of dialog error or when the dialog is terminated.

```define d ui.Dialog,
  t STRING
  ...
WHILE (t := d.nextEvent()) IS NOT NULL
  CASE t
    WHEN "BEFORE DISPLAY"
      ...
    WHEN "ON ACTION print"
      ...
    WHEN "ON DELETE"
      ...
    WHEN "AFTER DISPLAY"
      ...
END WHILE
```

The standard triggers such as "BEFORE ROW" and "AFTER FIELD field-name" are equivalent to the trigger found in static dialog control blocks. These allow you to control the behavior of the dynamic dialog.

The user-defined triggers that have been added with the `addTrigger()` method must also be handled in the dynamic dialog loop.

Inside the WHILE loop, control the behavior of the dialog with the methods provided in the `ui.Dialog` class. For example, to jump to a different field when the "jump" action is fired:

```.
  WHEN "ON ACTION jump"
    CALL d.nextField("customer.cust_name")
  ...
```

BEFORE/AFTER FIELD handlers must be identified with the field name (without the table/formonly prefix):

```.
  WHEN "AFTER FIELD cust_name"
    IF LENGTH(d.getFieldValue("customer.cust_name")) < 3 THEN
      ERROR "Customer name is too short"
      CALL d.nextField("customer.cust_name")
    END IF
  ...
```

For more details, see the `ui.Dialog.nextEvent()` method reference.

Controlling field values

Fields values in dynamic dialogs can be manipulated dynamically.

Unbuffered mode is the default

Dynamic dialogs do not use program variables and thus behave by default as static dialogs using the UNBUFFERED mode: When an action is fired, and the corresponding trigger handler is executed, the field is validated and the value is available with the `ui.Dialog.getFieldValue` on page 2533 method. Changing the value by program with `ui.Dialog.setFieldValue` on page 2550 is automatically displayed to the corresponding form-field and visible when the control goes back to the end user.
For more details see The buffered and unbuffered modes on page 1720.

**Default form-field values**

A dynamic input dialog created with `ui.Dialog.createInputByName` on page 2517 behaves like a static INPUT dialog using the WITHOUT DEFAULTS option: The DEFAULT attribute of the form-field is not used.

A dynamic input array dialog created with `ui.Dialog.createInputArrayFrom` on page 2516 behaves like a static INPUT ARRAY using the WITHOUT DEFAULTS option: The values set in the internal rows before starting the dialog will be used. However, like with a static INPUT ARRAY, when adding a new row, the DEFAULT attributes of the form-fields are used.

For more details about the WITHOUT DEFAULTS clause, see Form field initialization on page 1722.

**Setting an getting field values**

A dynamic dialog stores field values in internal buffers based on the field definitions provided in the creation method. Access to these values is required, to implement the dynamic dialog.

For example, to set default values before entering the dialog loop, modifying and/or querying values during the dialog loop, and to get the entered values after dialog termination when accepted by the user.

To set or get values of fields controlled by a dynamic dialog, use respectively the `ui.Dialog.setFieldValue()` and `ui.Dialog.getFieldValue()` methods.

**Note:** These methods take a form field name as parameter, that can be provided in different notations. See Identifying fields in ui.Dialog methods on page 2556 for more details.

When implementing a display array or input array dynamic dialog handling a record list, the set/get field value methods apply to the current row. If you want to set or get field values of a particular row, first move to the row with the `ui.Dialog.setCurrentRow()` method.

This example copies the values from the fields in the current row of a display array dynamic dialog (`d_list`), to the field buffers of a record input dynamic dialog (`d_rec`):

```plaintext
CALL d_list.setCurrentRow("sr_custlist", index)
FOR i=1 TO fields.getLength()
    CALL d_rec.setFieldValue( fields[i].name, 
    d_list.getFieldValue(fields[i].name) 
)
END FOR
```

**Getting query conditions for a field**

A dynamic dialog created with `ui.Dialog.createConstructByName()` handles query by example input (like CONSTRUCT).

To generate the SQL condition from the search value entered in a construct field, use the `ui.Dialog.getQueryFromField()` method, by passing the field name as parameter:

```plaintext
LET field_condition = DIALOG.getQueryFromField("customer.cust_name")
```

To build the complete WHERE part for the SELECT statement, iterate through all form fields and concatenate the form field condition by separating with the AND or with the OR operator:

```plaintext
FOR i=1 TO fields.getLength()
    LET field_condition = d.getQueryFromField(fields[i].name)
    IF field_condition IS NOT NULL THEN
        IF where_clause IS NOT NULL THEN 
            LET where_clause = where_clause, " AND "
        END IF 
        LET where_clause = where_clause, field_condition
    END IF
```
Ending dynamic dialogs
Describes how to terminate dynamic dialogs.

Implementing the accept and cancel actions
Regular static dialog instructions implement the accept and cancel actions, to respectively validate or abort the dialog. These actions are created automatically for static dialogs, but must be created by hand for dynamic dialogs.

In the case of cancel, you can mimic the behavior of static dialogs by setting the INT_FLAG register to TRUE and then directly leave the WHILE loop with an EXIT WHILE.

For the accept action, call the ui.Dialog.accept() method to validate field input and leave the dialog, and put the EXIT WHILE in the termination event, to leave the dialog events loop.

For example, to implement the accept and cancel actions for a simple record input:

```plaintext
DEFINE d ui.Dialog,
  t STRING
...
LET d = ui.Dialog.createInputByName(fields)
CALL d.addTrigger("ON ACTION cancel")
CALL d.addTrigger("ON ACTION accept")
...
WHILE (t := d.nextEvent()) IS NOT NULL
  CASE t
    WHEN "ON ACTION cancel"
      LET int_flag = TRUE
      EXIT WHILE
    WHEN "ON ACTION accept"
      CALL d.accept()
    WHEN "AFTER INPUT"
      EXIT WHILE
  END CASE
END WHILE
```

For a multiple dynamic dialog, the termination event is "AFTER DIALOG", and the EXIT WHILE must be done when that event occurs (for the accept action):

```plaintext
...
WHILE (t := d.nextEvent()) IS NOT NULL
  CASE t
    ...
    WHEN "AFTER DIALOG"
      EXIT WHILE
  END CASE
END WHILE
```

Note: In multiple dialogs, the AFTER INPUT, AFTER DISPLAY, AFTER CONSTRUCT triggers are used to indicate that the focus leaves the corresponding sub-dialog. Do not perform EXIT WHILE in these triggers for a multiple dialog.

Terminating the dialog
Some synchronization code needs to be implemented to properly destroy the dynamic dialog.
A dialog needs to be destroyed before closing its corresponding window/form.
In order to terminate a dialog, call the `close()` dialog method and assign `NULL` to the `ui.Dialog` variable referencing the dialog object. This will close the dialog and destroy the corresponding object, if no other variables references it.

When the dialog object is terminated, the corresponding window can also be closed:

```plaintext
... WHEN "ON ACTION cancel"
  EXIT WHILE
END WHILE
CALL d.close()
LET d = NULL
CLOSE WINDOW w1
```

**Dynamic multiple dialog**
Dynamic dialogs allows you to create the equivalent of a `DIALOG` procedural dialog.

**Steps to implement a dynamic multiple dialog**
The order in which dialog methods are called defines the structure of the dynamic multiple dialog.

To build a dynamic multiple dialog:

1. Create the multiple dialog with `createMultipleDialog()`.
2. Add global dialog triggers with `addTrigger()`.
3. Define the fields for a sub-dialog with a `DYNAMIC ARRAY OF RECORD` with `name` and `type` members.
4. Add the sub-dialog with one of the following methods:
   - `addConstructByName()`
   - `addDisplayArrayTo()`
   - `addInputArrayFrom()`
   - `addInputByName()`
5. Add sub-dialog triggers with `addTrigger()`. When calling this method after a sub-dialog item is added, it will create the trigger in the sub-dialog context.
6. Implement the `WHILE` loop using the `nextEvent()` method, as for a simple dynamic dialog. However, sub-dialog triggers must be identified with the sub-dialog name.

**Identifying global and sub-dialog actions**
Sub-dialog actions are added with `addTrigger("ON ACTION action-name")` in the context of a sub-dialog definition, after calling a method such as `addDisplayArrayTo()`.

In the event loop, the `nextEvent()` method returns sub-dialog actions events with the sub-dialog name prefix, using the "ON ACTION sub-dialog-name.action-name" text form:

```plaintext
... CALL d.addDisplayArrayTo(, "sr_cust")
CALL d.addTrigger("ON ACTION refresh")
...
WHILE (t := d.nextEvent()) IS NOT NULL
  CASE t
    ...
      WHEN "ON ACTION sr_cust.refresh"
      ...
```
Multiple dialog initialization and termination triggers

As in a static multiple dialog, the BEFORE DIALOG and AFTER DIALOG events indicate respectively the initialization and termination of the dynamic multiple dialog:

```
WHILE (t := d.nextEvent()) IS NOT NULL
    CASE t
        ... WHEN "BEFORE DIALOG"
        ...
        WHEN "AFTER DIALOG"
        ...
```

Identifying sub-dialog triggers

Sub-dialog triggers such as BEFORE DISPLAY, BEFORE INPUT can be identified in the event loop with the sub-dialog identifier. The `nextEvent()` method returns the event text in the form "trigger-name sub-dialog-name".

For example, when you define the DISPLAY ARRAY-style sub-dialog with the screen record name "sr_cust", the trigger "BEFORE DISPLAY sr_cust" will be generated when the sr_cust list gets the focus (as in a static DIALOG / END DIALOG multiple dialog block):

```
... CALL d.addDisplayArrayTo(, "sc_cust")
... CALL d.addDisplayArrayTo(, "sc_prod")
... WHILE (t := d.nextEvent()) IS NOT NULL
    CASE t
        ...
        WHEN "BEFORE DISPLAY sc_cust"
        ...
        WHEN "AFTER DISPLAY sc_cust"
        ...
        WHEN "BEFORE DISPLAY sc_prod"
        ...
        WHEN "AFTER DISPLAY sc_prod"
        ...
```

Example

This example defines a dynamic multiple dialog with a DISPLAY ARRAY and CONSTRUCT BY NAME sub-dialogs, and adds ON ACTION triggers at global and sub-dialog levels:

```
TYPE t_fields DYNAMIC ARRAY OF RECORD
    name STRING,
    type STRING
END RECORD
DEFINE d ui.Dialog,
    list_fields t_fields,
    qbe_fields t_fields
...
OPEN WINDOW w1 WITH FORM "myform"
...
LET d = ui.Dialog.createMultipleDialog()
-- add global triggers
CALL d.addTrigger("ON ACTION accept")
CALL d.addTrigger("ON ACTION cancel")
...
-- Add a DISPLAY ARRAY sub-dialog with name "sr_cust" (screen array)
```
CALL d.addDisplayArrayTo(list_fields, "sr_cust")
CALL d.addTrigger("ON ACTION refresh")
...
-- Add CONSTRUCT BY NAME sub-dialog with name "qr_cust"
CALL d.addConstructByName(qbe_fields, "qr_cust")
CALL d.addTrigger("ON ACTION query")
...
WHILE (t := d.nextEvent()) IS NOT NULL
CASE t
    ...
    WHEN "ON ACTION sr_cust.refresh"
        CALL refresh_list(d)
    ...
    WHEN "ON ACTION qr_cust.query"
        CALL exec_query(d)
    ...
    WHEN "ON ACTION accept"
        CALL d.accept()
    WHEN "ON ACTION close"
        EXIT WHILE
    WHEN "AFTER DIALOG" -- after d.accept()
        EXIT WHILE
END CASE
END WHILE
CALL d.close()
LET d = NULL
CLOSE WINDOW w1

Related concepts
Sub-dialog actions in procedural DIALOG blocks on page 1765
This topic describes how action are differentiated with handlers defined in a procedural DIALOG block.

Using dynamic cursors
Implementing a dynamic dialog based on the database schema.

To write generic code accessing a database, implement the dynamic dialog with field names and types coming from a base.SqlHandle cursor.

The following code example builds a list of fields based on the database table passed as first parameter.

The function scans the result set column names and types of the base.SqlHandle cursor, to build the list of field definitions, that can then be used for the dynamic dialog creation:

FUNCTION build_field_list(dbtable, fields)
    DEFINE dbtable STRING,
        fields DYNAMIC ARRAY OF RECORD
        name STRING,
        type STRING
    END RECORD
    DEFINE h base.SqlHandle,
        i INT
    LET h = base.SqlHandle.create()
    CALL h.prepare("SELECT * FROM " || dbtable)
    CALL h.open()
    CALL h.fetch()
    CALL fields.clear()
    FOR i=1 TO h.getResultCount()
        LET fields[i].name = h.getResultName(i)
        LET fields[i].type = h.getResultType(i)
    END FOR
END FUNCTION
Additional features
Miscellaneous features of dynamic dialogs.

Defining cell attributes
In order to display array cells with colors in a list controlled by a dynamic dialog, you can define a two-dimensional dynamic array to hold cell attributes:

```prolog
DEFINE attrs DYNAMIC ARRAY OF DIMENSION 2 OF STRING
...
LET attrs[row,col] = "red reverse"
...
CALL d.setArrayAttributes("custlist", attrs)
```

Note: If all cells of the array must get the same display attributes, use a simple DYNAMIC ARRAY OF STRING.

Canvases
Canvases are form drawing areas.

Understanding canvases
This is an introduction to CANVAS drawing.

A canvas element defines a drawing area in a form, to show basic colored shapes.

Important: This feature is not supported on mobile platforms.

Canvas can draw lines, rectangles, ovals, circles, texts, arcs, and polygons. Keys can be bound to graphical elements for selection with a right or left mouse click.

In programs, you select a given canvas area by name and you create the shapes in the abstract user interface tree by using the built-in DOM API, or helper functions.

The painted canvas is automatically displayed on the front-end when an interactive instruction is executed, such as MENU or INPUT.

Each canvas element is identified by a unique number (id). You can use this identifier to bind mouse clicks to canvas elements.

Note: Consider using Web Components for specific drawing needs (charts, graphics). For example, Genero BDL provides the build-in Web Component fglsvgcanvas.

Related concepts
CANVAS item definition on page 1334
The CANVAS form item defines an area in which you can draw shapes, in a grid-based layout.

User interface basics on page 1105
This section introduces to the foundation of the Genero user interface.

CANVAS item definition
The CANVAS form item defines an area in which you can draw shapes, in a grid-based layout.

Syntax
Important: This feature is deprecated, its use is discouraged although not prohibited.

```prolog
CANVAS item-tag: item-name ↓, attribute-list ↓;
```

1. *item-tag* is an identifier that defines the name of the item tag in the layout section.
2. *item-name* identifies the form item.
3. *attribute-list* defines the aspect and behavior of the form item.
Form attributes

COMMENT, HIDDEN, TAG.

Style attributes

Common: backgroundColor, border, fontFamily, fontSize, fontStyle, fontWeight, localAccelerators, showAcceleratorInToolTip, textColor, textDecoration.

Class-specific: none.

Usage

Define the rendering and behavior of a canvas drawing area item tag, with a CANVAS element in the ATTRIBUTES section.

Note: The CANVAS feature is deprecated, consider using the built-in fglsvgcanvas WEBCOMPONENT.

Example

LAYOUT
GRID
{
[cvs1         ]
[             ]
[             ]
...           
}
END
END

ATTRIBUTES
CANVAS cvs1: canvas1;
...

Related concepts

Canvases on page 2013
Canvases are form drawing areas.

Syntax of canvas nodes

The AUI tree contains the XML representation of the canvas, with a variety of elements defining shapes.

Canvas areas are defined in forms with the following XML syntax:

```xml
<Canvas colName="name" >
  | <CanvasArc canvainternal-attribute="value" [...] />
  | <CanvasCircle canvainternal-attribute="value" [...] />
  | <CanvasLine canvainternal-attribute="value" [...] />
  | <CanvasOval canvainternal-attribute="value" [...] />
  | <CanvasPolygon canvainternal-attribute="value" [...] />
  | <CanvasRectangle canvainternal-attribute="value" [...] />
  | <CanvasText canvainternal-attribute="value" [...] />
  [...]
</Canvas>
[...]```
### Table 400: Types of canvas element

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanvasArc</td>
<td>Arc defined by the bounding square top left point, a diameter, a start angle, an end angle, and a fill color.</td>
</tr>
<tr>
<td>CanvasCircle</td>
<td>Circle defined by the bounding square top left point, a diameter, and a fill color.</td>
</tr>
<tr>
<td>CanvasLine</td>
<td>Line defined by a start point, an end point, width, and a fill color.</td>
</tr>
<tr>
<td>CanvasOval</td>
<td>Oval defined by rectangle (with start point and endpoint), and a fill color.</td>
</tr>
<tr>
<td>CanvasPolygon</td>
<td>Polygon defined by a list of points, and a fill color.</td>
</tr>
<tr>
<td>CanvasRectangle</td>
<td>Rectangle defined by a start point, an end point, and a fill color.</td>
</tr>
<tr>
<td>CanvasText</td>
<td>Text defined by a start point, an anchor hint, the text, and a fill color.</td>
</tr>
</tbody>
</table>

### Table 401: Attributes of canvas elements

<table>
<thead>
<tr>
<th>Name</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>startX</td>
<td>INTEGER (0-&gt;1000)</td>
<td>X position of starting point.</td>
</tr>
<tr>
<td>startY</td>
<td>INTEGER (0-&gt;1000)</td>
<td>Y position of starting point.</td>
</tr>
<tr>
<td>endX</td>
<td>INTEGER (0-&gt;1000)</td>
<td>X position of ending point.</td>
</tr>
<tr>
<td>endY</td>
<td>INTEGER (0-&gt;1000)</td>
<td>Y position of ending point.</td>
</tr>
<tr>
<td>xyList</td>
<td>STRING</td>
<td>Space-separated list of XY coordinates. For example: &quot;23 45 56 78&quot; defines (x=23, y=45) (x=56, y=78).</td>
</tr>
<tr>
<td>width</td>
<td>INTEGER</td>
<td>Width of the shape.</td>
</tr>
<tr>
<td>height</td>
<td>INTEGER</td>
<td>Height of the shape.</td>
</tr>
<tr>
<td>diameter</td>
<td>INTEGER</td>
<td>Diameter for circles and arcs.</td>
</tr>
<tr>
<td>startDegrees</td>
<td>INTEGER</td>
<td>Beginning of the angular range occupied by an arc.</td>
</tr>
<tr>
<td>extentDegrees</td>
<td>INTEGER</td>
<td>Size of the angular range occupied by an arc.</td>
</tr>
<tr>
<td>text</td>
<td>STRING</td>
<td>The text to draw.</td>
</tr>
<tr>
<td>anchor</td>
<td>&quot;n&quot;, &quot;e&quot;, &quot;w&quot;, &quot;s&quot;</td>
<td>Anchor hint to give the draw direction for texts.</td>
</tr>
<tr>
<td>fillColor</td>
<td>STRING</td>
<td>Name of the color to be used for the element.</td>
</tr>
<tr>
<td>acceleratorKey1</td>
<td>STRING</td>
<td>Name of the key associated to a left button click.</td>
</tr>
<tr>
<td>acceleratorKey3</td>
<td>STRING</td>
<td>Name of the key associated to a right button click.</td>
</tr>
</tbody>
</table>
**Canvas drawing area**

The canvas area defines a two-dimensional coordinate system for drawing elements.

The canvas area represents an abstract drawing page where you define size and location of shapes with coordinates from (0,0) to (1000,1000).

The origin point (0,0), is on the bottom-left of the drawing area.

---

Figure 118: Canvas area diagram

The drawing area is defined in the form file with a `CANVAS` form item. At runtime, you draw the content of canvas areas in the Abstract User Interface tree. In a form defining canvas areas, the Abstract User Interface tree contains empty `<Canvas>` nodes that you can fill with canvas items.

A canvas node is identified in the program by the `name` attribute. You can get the canvas node by name with the `Window.getElement(name)` method.

You cannot drop canvas area nodes, as they are read-only in a form definition.

**Related concepts**

- **CANVAS item definition** on page 1334
- The `CANVAS` form item defines an area in which you can draw shapes, in a grid-based layout.

**Canvas drawing functions**

This table describes the helper functions provided to ease canvas usage. Use these functions or use the DOM API to directly create canvas elements in the form. The helper functions are implemented in `$FGLDIR/src/fgldraw.4gl`. See the source file for more details.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>drawInit()</code></td>
<td>Initializes the drawing API. It is mandatory to call this function at the beginning of your program, before the first display instruction.</td>
</tr>
<tr>
<td><code>drawSelect()</code></td>
<td>Selects a canvas area for drawing.</td>
</tr>
<tr>
<td><code>drawDisableColorLines()</code></td>
<td>By default, simple lines drawn with <code>drawLine()</code> are colored by <code>drawFillColor()</code>. Pass TRUE to the function to get black lines.</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>drawFillColor()</td>
<td>Defines the fill color for shapes and lines. Color value are named colors like &quot;red&quot;, &quot;green&quot;, &quot;blue&quot;, etc.</td>
</tr>
<tr>
<td>drawLineWidth()</td>
<td>Defines the width of lines.</td>
</tr>
<tr>
<td>drawAnchor()</td>
<td>Defines the anchor hint for texts.</td>
</tr>
<tr>
<td>drawLine()</td>
<td>Draws a line in the selected canvas.</td>
</tr>
<tr>
<td>drawCircle()</td>
<td>Draws a circle in the selected canvas.</td>
</tr>
<tr>
<td>drawArc()</td>
<td>Draws an arc in the selected canvas.</td>
</tr>
<tr>
<td>drawRectangle()</td>
<td>Draws a rectangle in the selected canvas.</td>
</tr>
<tr>
<td>drawOval()</td>
<td>Draws an oval in the selected canvas.</td>
</tr>
<tr>
<td>drawText()</td>
<td>Draws a text in the selected canvas.</td>
</tr>
<tr>
<td>drawPolygon()</td>
<td>Draws a polygon in the selected canvas.</td>
</tr>
<tr>
<td>drawClear()</td>
<td>Clears the selected canvas.</td>
</tr>
<tr>
<td>drawButtonLeft()</td>
<td>Enables left mouse click on a canvas element.</td>
</tr>
<tr>
<td>drawButtonRight()</td>
<td>Enables right mouse click on a canvas element.</td>
</tr>
<tr>
<td>drawClearButton()</td>
<td>Disables all mouse clicks on a canvas element.</td>
</tr>
<tr>
<td>drawGetClickedItemId()</td>
<td>Returns the id of the last clicked canvas element.</td>
</tr>
</tbody>
</table>

**Examples**

Canvas usage examples.

**Example 1: Simple canvas**

This topic describes the steps to draw elements in a canvas.

First define a drawing area in the form file with the CANVAS form item type. In this example, the name of the canvas field is 'canvas01'. This field name identifies the drawing area:

```plaintext
DATABASE FORMONLY
LAYOUT
GRID
{
  Canvas example:
  [ca01                         
  [                             
  [                             
  [                             
  [                             
  [                             
  ]]
}
END
END
ATTRIBUTES
CANVAS ca01: canvas01;
END
```

In programs, you draw canvas shapes by creating canvas nodes in the abstract user interface tree with the DOM API utilities.

Define a variable to hold the DOM node of the canvas and a second to handle children created for shapes:

```plaintext
DEFINE c, s om.DomNode
```
Define a window object variable; open a window with the form containing the canvas area; get the current window object, and then get the canvas DOM node:

```
DEFINE w ui.Window
OPEN WINDOW w1 WITH FORM "form1"
LET w = ui.Window.getCurrent()
LET c = w.findNode("Canvas","canvas01")
```

Create a child node with a specific type defining the shape:

```
LET s = c.createChild("CanvasRectangle")
```

Set attributes to complete the shape definition:

```
CALL s.setAttribute( "fillColor", "red" )
CALL s.setAttribute( "startX", 10 )
CALL s.setAttribute( "startY", 20 )
CALL s.setAttribute( "endX", 100 )
CALL s.setAttribute( "endY", 150 )
```

It is possible to bind keys / actions to Canvas items in order to let the end user select elements with a mouse click. You can assign a function key for left-button mouse clicks with the `acceleratorKey1` attribute, while `acceleratorKey3` is used to detect right-button mouse clicks. The function keys you can bind are F1 to F255. If the user clicks on a Canvas item bound to key actions, the corresponding action handler will be executed in the current dialog. Several canvas items can be bound to the same action keys; in order to identify which items have been selected by a mouse click, you can use the `drawGetClickedItemId()` function of `fgldraw.4gl`. This method will return the AUI tree node id of the Canvas item that was selected (`s.getId()`).

```
... Create the Canvas item with s node variable ...
CALL s.setAttribute( "acceleratorKey1", "F50" )
MENU "test"
  COMMAND KEY (F50)
    IF drawGetClickedItemId() = s.getId() THEN
      ...
    END IF
... 
END MENU
```

To clear a given shape in the canvas, remove the element in the canvas node:

```
CALL c.removeChild(s)
```

To clear the drawing area completely, remove all children of the canvas node:

```
LET s=c.getFirstChild()
WHILE s IS NOT NULL
  CALL c.removeChild(s)
  LET s=c.getFirstChild()
END WHILE
```

**Related concepts**

- [The DomNode class](/UserInterface/Chapter22_DomNodeClass) on page 2584
The `omDOMNode` class provides methods to manipulate a DOM node of a data tree.

**Start menus**

Start menus define a tree of application programs that can be started.

**Understanding start menus**

This is an introduction to start menus.

The start menu defines a tree of commands that start programs on the application server where the runtime system executes.

**Important:** This feature is not supported on mobile platforms.

It is recommended that you create a specific program dedicated to running the start menu. This program must create (or load) a start menu, and then perform an interactive instruction to enter the interaction loop.

The start menu must be defined in the abstract user interface tree under the "UserInterface" root node.

The start menu is unique for a program and cannot be redefined.

When a start menu command is selected by the user, the runtime system automatically starts a child process with the command specified in the command attribute.

**Note:** By default, the start menu is not displayed: Use presentation style attributes to control the rendering of a start menu.

**Related concepts**

The abstract user interface tree on page 1106

The abstract user interface tree is the XML representation of the application forms displayed to the end user.

**Syntax of start menu files (.4sm)**

A start menu file contains a tree of XML elements defining the application menu to start programs.

Start menus are defined in a `.4sm` file with the following XML syntax:

```xml
<StartMenu startmenu-attribute="value" [...]
  group[...]
</StartMenu>
```

where `group` is:

```xml
<StartMenuGroup group-attribute="value"
  [...]>[ <StartMenuSeparator/>
  <StartMenuCommand command-attribute="value"
    [...]/
  group]
  [...]
</StartMenuGroup>
```

1. `startmenu-attribute` defines a property of the `StartMenu`.
2. `command-attribute` defines a property of a `StartMenuCommand`.
3. `group-attribute` defines a property of a `StartMenuGroup`.

**Table 403: Attributes of the StartMenu node**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>STRING</td>
<td>Identifies the StartMenu, can be omitted.</td>
</tr>
<tr>
<td>text</td>
<td>STRING</td>
<td>Defines the text to be displayed as title.</td>
</tr>
</tbody>
</table>
### Table 404: Attributes of the StartMenuGroup node

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>disabled</td>
<td>INTEGER</td>
<td>Indicates if the group must be disabled (grayed, cannot be selected).</td>
</tr>
<tr>
<td>hidden</td>
<td>INTEGER</td>
<td>Indicates if the group is hidden or visible.</td>
</tr>
<tr>
<td>image</td>
<td>STRING</td>
<td>Defines the icon to be used for this group.</td>
</tr>
<tr>
<td>name</td>
<td>STRING</td>
<td>Identifies the start menu group, can be omitted.</td>
</tr>
<tr>
<td>text</td>
<td>STRING</td>
<td>Defines the text to be displayed for this group.</td>
</tr>
</tbody>
</table>

### Table 405: Attributes of the StartMenuCommand node

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>disabled</td>
<td>INTEGER</td>
<td>Indicates if the item must be disabled (grayed, cannot be selected).</td>
</tr>
<tr>
<td>comment</td>
<td>STRING</td>
<td>Specifies the comment to be shown for this command.</td>
</tr>
<tr>
<td>exec</td>
<td>STRING</td>
<td>Defines the command to be executed when the user selects this command.</td>
</tr>
<tr>
<td>hidden</td>
<td>INTEGER</td>
<td>Indicates if the command is hidden or visible.</td>
</tr>
<tr>
<td>image</td>
<td>STRING</td>
<td>Defines the icon to be used for this command.</td>
</tr>
<tr>
<td>name</td>
<td>STRING</td>
<td>Identifies the StartMenu item, can be omitted.</td>
</tr>
<tr>
<td>text</td>
<td>STRING</td>
<td>Defines the text to be displayed for this command.</td>
</tr>
<tr>
<td>waiting</td>
<td>INTEGER</td>
<td>Defines if the command must be started without waiting (0, default) or waiting (1).</td>
</tr>
</tbody>
</table>

### Table 406: Attributes of the StartMenuSeparator node

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>STRING</td>
<td>Identifies the StartMenu separator, can be omitted.</td>
</tr>
</tbody>
</table>

### Loading a start menu from an XML file

Start menu XML definition files can be loaded at runtime.

To load a start menu definition file, use the utility method provided by the `ui.Interface` built-in class:

```
CALL ui.Interface.loadStartMenu("standard")
```
Creating the start menu dynamically

Start menu can be created dynamically with the om.DomNode class.

First, get the abstract user interface root node:

```om
DEFINE aui om.DomNode
LET aui = ui.Interface.getRootNode()
```

Next, create a node with the "StartMenu" tag name:

```om
DEFINE sm om.DomNode
LET sm = aui.createChild("StartMenu")
```

Next, create a "StartMenuGroup" node to group a couple of command nodes:

```om
DEFINE smg om.DomNode
LET smg = sm.createChild("StartMenuGroup")
CALL smg.setAttribute("text","Programs")
```

Then, create "StartMenuCommand" nodes for each program and, if needed, add "StartMenuSeparator" nodes to separate entries:

```om
DEFINE smc, sms om.DomNode
LET smc = smg.createChild("StartMenuCommand")
CALL smc.setAttribute("text","Orders")
CALL smc.setAttribute("exec","fglrun orders.42r")
LET smc = smg.createChild("StartMenuCommand")
CALL smc.setAttribute("text","Customers")
CALL smc.setAttribute("exec","fglrun customers.42r")
LET sms = smg.createChild("StartMenuSeparator")
LET smc = smg.createChild("StartMenuCommand")
CALL smc.setAttribute("text","Items")
CALL smc.setAttribute("exec","fglrun items.42r")
```

Related concepts

- [The DomNode class](#) on page 2584
  The om.DomNode class provides methods to manipulate a DOM node of a data tree.
- [The Interface class](#) on page 2470
  The ui.Interface class provides methods to manipulate the user interface.
- [The abstract user interface tree](#) on page 1106
  The abstract user interface tree is the XML representation of the application forms displayed to the end user.

Rendering of a start menu

Start menus can be displayed in different ways.

By default, start menus are not rendered: You need to define the startMenuPosition style attribute for the current window, to a value different from "none".

Secondary style attributes can be used to specify the decoration and behavior of the start menu:

- startMenuSize,
- startMenuShortcut,
- startMenuAccelerator,
- startMenuExecShortcut2
In the next example, the presentation styles file defines the start menu attributes to render the start menu as a large tree:

```
<Style name="Window.container">
  <StyleAttribute name="windowType" value="normal" />
  <StyleAttribute name="startMenuPosition" value="tree" />
  <StyleAttribute name="startMenuSize" value="large" />
</Style>
```

You can then reference this style in the form definition with `WINDOWSTYLE` attribute:

```
LAYOUT (WINDOWSTYLE="tsm")
...
```

Or in the program, when creating the window, with the `STYLE` attribute of `CREATE WINDOW`:

```
OPEN WINDOW w1 WITH FORM "f1" ATTRIBUTES(STYLE="tsm")
```

**Related concepts**

**Presentation styles** on page 1165

Use presentation styles to specify decoration attributes for window and form elements.

**Examples**

Start menu usage examples.

**Example 1: Start menu in XML format**

```
<StartMenu>
  <StartMenuGroup text="Ordering">
    <StartMenuCommand text="Orders" exec="fglrun orders" disabled="1" />
    <StartMenuCommand text="Customers" exec="fglrun custs" image="smiley" />
    <StartMenuCommand text="Items" exec="fglrun items" waiting="1" />
    <StartMenuCommand text="Reports" exec="fglrun reports" comment="Run reports" />
  </StartMenuGroup>
  <StartMenuGroup text="Configuration">
    <StartMenuCommand text="Database" exec="fglrun dbseconf" />
    <StartMenuCommand text="Users" exec="fglrun userconf" />
    <StartMenuCommand text="Printers" exec="fglrun prntconf" />
  </StartMenuGroup>
</StartMenu>
```

**Example 2: Start menu created dynamically**

```
MAIN
  DEFINE aui om.DomNode
  DEFINE sm om.DomNode
  DEFINE smg om.DomNode
  DEFINE smc om.DomNode
  LET aui = ui.Interface.getRootNode()
  LET sm = aui.createChild("StartMenu")
  LET smg = createStartMenuGroup(sm,"Ordering")
  LET smc = createStartMenuCommand(smg,"Orders","fglrun orders",NULL)
  LET smc = createStartMenuCommand(smg,"Customers","fglrun custs",NULL)
  LET smc = createStartMenuCommand(smg,"Items","fglrun items",NULL)
  LET smc = createStartMenuCommand(smg,"Reports","fglrun reports",NULL)
```

LET smg = createStartMenuGroup(sm,"Configuration")
LET smc = createStartMenuCommand(smg,"Database","fglrun dbseconf",NULL)
LET smc = createStartMenuCommand(smg,"Users","fglrun userconf",NULL)
LET smc = createStartMenuCommand(smg,"Printers","fglrun prntconf",NULL)

MENU "Example"
  COMMAND "Quit"
  EXIT PROGRAM
END MENU

END MAIN

FUNCTION createStartMenuGroup(p,t)
  DEFINE p om.DomNode
  DEFINE t STRING
  DEFINE s om.DomNode
  LET s = p.createChild("StartMenuGroup")
  CALL s.setAttribute("text",t)
  RETURN s
END FUNCTION

FUNCTION createStartMenuCommand(p,t,c,i)
  DEFINE p om.DomNode
  DEFINE t,c,i STRING
  DEFINE s om.DomNode
  LET s = p.createChild("StartMenuCommand")
  CALL s.setAttribute("text",t)
  CALL s.setAttribute("exec",c)
  CALL s.setAttribute("image",i)
  RETURN s
END FUNCTION

Window containers (WCI)

WCI containers define window containers to group several programs in a parent multiple document interface presentation.

Understanding the Window Container Interface
This is an introduction to WCI programming.

By default, application windows are displayed independently in separate windows on the front-end window manager. This mode is known as SDI, "Single Document Interface".

The user interface can be configured to group program windows in a parent container. This is known as MDI, "Multiple Document Interface". In Genero, Multiple Document Interface is called WCI: Window Container Interface.

Important: The Window Container Interface is a desktop application feature (for GDC), and is not supported on other front-ends (web and mobile). This feature is also denied when using Universal Rendering in GDC.

The WCI can be used to group several programs together in a parent window. The parent program is the container for the other programs, defined as children of the container. The container program can have its own windows, but this makes sense only for temporary modal windows (with style="dialog").

WCI configuration is done dynamically at the beginning of programs, with methods of the ui.Interface built-in class.

Related concepts
Windows and forms on page 1131
The section describes the concept of windows and forms in the language.

**Configuration of WCI parent programs**

Programs acting as parent WCI containers must perform WCI configuration calls when they start.

**Important:** The Window Container Interface is a desktop application feature (for GDC), and is not supported on other front-ends (web and mobile). This feature is also denied when using Universal Rendering in GDC.

The WCI container program is a separate program of a special type, dedicated to contain other program windows. On the front-end, container programs automatically display a parent window that will hold all child program windows that attach to the container.

The WCI container program must indicate that its type is special (ui.Interface.setType() method), and must identify itself (ui.Interface.setName() method):

```plaintext
MAIN
 CALL ui.Interface.setName("parent1")
 CALL ui.Interface.setContainer("container")
 CALL ui.Interface.setText("SoftStore Manager")
 CALL ui.Interface.setSize("600px","1000px")
 CALL ui.Interface.loadStartMenu("mystartmenu")
 MENU "Main"
  COMMAND "Help" CALL help()
  COMMAND "About" CALL aboutbox()
  COMMAND "Exit" EXIT MENU
 END MENU
END MAIN
```

You can define the initial size of the parent container window with the `ui.Interface.setSize(height,width)` method.

When the program is identified as a container, a global window is automatically displayed as a container window. The default toolbar and the default topmenu are displayed and a startmenu can be used. Other windows created by this kind of program can be displayed, inside the container or as dialog windows respectively with the `windowType="normal" or windowType="modal"` style attribute. Other Window styles can be applied to the parent window by using the default style specification (name="Window.main").

**Related concepts**

- **The Interface class** on page 2470
  The `ui.Interface` class provides methods to manipulate the user interface.

- **Toolbars** on page 1436
  Toolbars define a bar of buttons that appears at the top of application forms.

- **Topmenus** on page 1444
  Topmenus define typical pull-down menus that appear at the top of application forms.

- **Start menus** on page 2019
  Start menus define a tree of application programs that can be started.

**Configuration of WCI child programs**

WCI child programs attach to a parent container by referencing the WCI container by name.

**Important:** The Window Container Interface is a desktop application feature (for GDC), and is not supported on other front-ends (web and mobile). This feature is also denied when using Universal Rendering in GDC.

WCI children programs must attach to a parent container by giving the name of the container program:

```plaintext
MAIN
 CALL ui.Interface.setName("custapp")
 CALL ui.Interface.setContainerType("child")
 CALL ui.Interface.setText("Customers")
 CALL ui.Interface.setContainer("parent1")
 ...
```
Multiple container programs can be used to group programs by application modules.

The client displays a system error and the program stops when:

- A child program is started, but the parent container is not
- A container program is started twice

When the parent container program is stopped, other applications are automatically stopped by the front-end. This will result in a runtime error -6313 on the application server side. To avoid this, ensure that there are no more running child programs before terminating the parent container program. The WCI container program can query for the existence of children with the ui.Interface.getChildCount() and ui.Interface.getChildInstances() methods:

```plaintext
MAIN
    CALL ui.Interface.setName("parent1")
    CALL ui.Interface.setType("container")
    CALL ui.Interface.setText("SoftStore Manager")
    CALL ui.Interface.setSize("600px","1000px")
    CALL ui.Interface.loadStartMenu("mystartmenu")
    MENU "Main"
        COMMAND "Help" CALL help()
        COMMAND "About" CALL aboutbox()
        COMMAND "Exit"
            IF ui.Interface.getChildCount()>0 THEN
                ERROR "You must first exit the child programs."
            ELSE
                EXIT MENU
            END IF
    END MENU
END MAIN
```

**Related concepts**

The Interface class on page 2470

The `ui.Interface` class provides methods to manipulate the user interface.

### Implement tabbed WCI containers

WCI container can display child programs in a folder tab.

**Important:** The Window Container Interface is a desktop application feature (for GDC), and is not supported on other front-ends (web and mobile). This feature is also denied when using Universal Rendering in GDC.

To display the child programs in a folder tab, define the presentation style attribute `tabbedContainer` to yes.

With a tabbed window container, the style attribute `tabbedContainerCloseMethod` defines how to close the current page.

Values can be:

- "container" (default), the container has a close button on the top right corner, which closes the current tab.
- "page", each page has its own close button.
- "both", each page and the container have a close button.
- "none", no close button is shown.

The close button is enabled depending on the window style attribute.
Understanding reports

A report can arrange and format the data depending on the instructions used and display the output on the screen, send it to a printer, or store it as a file for future use.

To implement a report, a program must include two distinct components:

- The report driver specifies what data the report includes.
- The report routine formats the data for output.

The report driver retrieves the specified rows from a database, stores their values in program variables, and sends these - one input record at a time - to the report routine. After the last input record is received and formatted, the runtime system calculates any aggregate values based on all the data and sends the entire report to some output device.

Figure 119: Report driver and database cursor

By separating the two tasks of data retrieval and data formatting, the runtime system simplifies the production of recurrent reports and makes it easy to apply the same report format to different data sets.

The report engine supports the following features:

- The option to display report output to the screen, to a printer, to a file, or to a SAX handler to transform the output following XML standards. When using the GDC, it is possible to send to report to the local printer configured in GDC, by setting the DBPRINT environment variable to the value "FGLSERVER".
- Full control of page layout, including first page header and generic page headers, page trailers, columnar presentation, and row grouping.
- Facilities for creating the report either from the rows returned by a cursor or from input records assembled from any other source, such as output from several different SELECT statements through the report driver.
- Control blocks to manipulate data from a database cursor on a row-by-row basis, either before or after the row is formatted by the report.
- Aggregate functions that can calculate frequencies, percentages, sums, averages, minimum, and maximum values.
- The USING operator and other built-in functions and operators for formatting and displaying information in output from the report.
The \texttt{WORDWRAP} operator to format long character strings that occupy multiple lines of output from the report.

The option to execute other language statements while generating a report.

Stopping a report in the report definition code, with \texttt{EXIT REPORT} or \texttt{TERMINATE REPORT}.

The report engine supports one-pass reports and two-pass reports. The one-pass requires sorted data to be produced by the report driver in order to handle row grouping with the \texttt{BEFORE GROUP / AFTER GROUP} blocks. The two-pass record handles sort automatically and does not need sorted data from the report driver. During the first pass, the report engine sorts the data and stores the sorted values in a temporary file in the database. During the second pass, it calculates any aggregate values and produces output from data in the temporary files.

\textbf{Related concepts}

\textit{The report driver} on page 2033

The report \textit{driver} retrieves data, starts the report engine and sends the data (as input records) to be formatted by the \texttt{REPORT} routine.

\textit{The report routine} on page 2039

The report \textit{routine} implements the body of a report, with formatting instructions.

\textit{XML output for reports} on page 2027

For better integration with external tools based on XML standards, reports can produce XML output.

\section*{XML output for reports}

For better integration with external tools based on XML standards, reports can produce XML output.

The purpose of XML-based reports is to sort and group data, not to decorate. Data decoration and formatting can be done by external tools, or you can redirect the XML report output to a SAX document handler object to process the output and generate for example HTML pages.

\section*{Writing an XML report driver and routine}

\subsection*{Generating XML output}

To produce an XML report, initiate the report with the \texttt{START REPORT} instruction followed by the \texttt{TO XML HANDLER} clause, to specify the \texttt{SAX document handler} that will process the XML report output:

\begin{verbatim}
START REPORT order_report
   TO XML HANDLER om.XmlWriter.createFileWriter("orders.xml")
\end{verbatim}

In the report routine, you must use the \texttt{PRINTX} statement to generate XML output:

\begin{verbatim}
REPORT order_report(rec)
   ...
   FORMAT ON EVERY ROW
      PRINTX NAME = order rec.*
   ...
END REPORT
\end{verbatim}

The \texttt{PRINTX} instruction takes an optional \texttt{NAME} argument to define the name of the XML node.

\subsection*{Nested XML reports}

If a new report is started with \texttt{START REPORT} instruction inside a \texttt{REPORT} routine producing XML, and if there is no destination specified in the \texttt{START REPORT} instruction, the sub-report inherits the XML output target of the parent, and sub-report nodes will be merged into the parent XML output:

\begin{verbatim}
REPORT order_report(rec)
   ...
\end{verbatim}
FORMAT
ON EVERY ROW
    PRINTX NAME = order rec.*
    -- Merges sub-report output to parent report XML handler
START REPORT sub_report
FOR ... 
    OUTPUT TO REPORT sub_report(...) 
END FOR 
FINISH REPORT sub_report
...
END REPORT

API for global XML handler

The \texttt{fgl\_report\_set\_document\_handler()} built-in function can be used to specify a general XML handler, for \texttt{START REPORT} instructions which do not use the \texttt{TO XML HANDLER} clause:

\begin{verbatim}
MAIN
...
    CALL
    fgl_report_set_document_handler( om.XmlWriter.createFileWriter("orders.xml") )
...
END MAIN
\end{verbatim}

\textbf{Note:} The \texttt{fgl\_report\_set\_document\_handler()} function is supported for backward compatibility, it is recommended to use \texttt{START REPORT ... TO XML HANDLER} instead.

\textbf{Related concepts}

The \texttt{SaxDocumentHandler} class on page 2617

The \texttt{om.SaxDocumentHandler} class provides an interface to write an XML filter with events.

Structure of XML report output

The generated XML output contains the structure of the formatted pages, with page header, page trailer, and group sections.

Every \texttt{PRINTX} instruction will generate a \texttt{<Print>} node with a list of \texttt{<Item>} nodes containing the data. The XML processor can use this structure to format and render the output as needed.

The output of an XML report will have the following node structure:

\begin{verbatim}
<Report ...>
    <PageHeader pageNo="...">
        ...
    </PageHeader>
    <Group>
        <BeforeGroup>
            <Print name="...">
                <Item name="...." type="...." value="...." isoValue="...." />
                <Item name="...." type="...." value="...." isoValue="...." />
                ...
            </Print>
            ...
        </BeforeGroup>
        <OnEveryRow>
            <Print name="...">
                <Item name="...." type="...." value="...." isoValue="...." />
                <Item name="...." type="...." value="...." isoValue="...." />
                ...
            </Print>
        </OnEveryRow>
    </Group>
</Report>
\end{verbatim}
Related concepts

**START REPORT** on page 2034

The **START REPORT** instruction initializes a report execution.

### Conditional statements in XML output

**Principle**

If **PRINTX** commands are used inside program flow control instructions like **IF**, **CASE**, **FOR**, **FOREACH** and **WHILE**, the XML output will contain additional nodes to identify such conditional print instructions.

That information can be useful to process an XML report output.

#### FOR ... END FOR

```xml
<For>
  <ForItem>
    <Print name="...">
      <Item name="..." type="..." value="..." isoValue="..." />
      <Item name="..." type="..." value="..." isoValue="..." />
    </Print>
    ...
  </ForItem>
  ...
</For>
```

#### WHILE ... END WHILE

```xml
<While>
  <WhileItem>
    <Print name="...">
      <Item name="..." type="..." value="..." isoValue="..." />
      <Item name="..." type="..." value="..." isoValue="..." />
    </Print>
    ...
  </WhileItem>
  ...
</While>
```
FOREACH ... END FOREACH

```xml
<Foreach>
  <ForeachItem>
    <Print name="...">
      <Item name="..." type="..." value="..." isoValue="..." />
    </Print>
    ...
  </ForeachItem>
  ...
</Foreach>
```

CASE ... END CASE

```xml
<Case>
  <When id="position">
    <Print name="...">
      <Item name="..." type="..." value="..." isoValue="..." />
    </Print>
    ...
  </When>
  ...
</Case>
```

IF ... THEN ... ELSE ... END IF

```xml
<If>
  <IfThen>
    <Print name="...">
      <Item name="..." type="..." value="..." isoValue="..." />
    </Print>
    ...
  </IfThen>
  <IfElse>
    <Print name="...">
      <Item name="..." type="..." value="..." isoValue="..." />
    </Print>
    ...
  </IfElse>
</If>
```

Examples

**Example 1: Nested XML report**

This code example produces XML output from a main and sub-report:

```plaintext
TYPE t_cust RECORD
  cust_id INTEGER,
  cust_name VARCHAR(30)
END RECORD

TYPE t_ord RECORD
  ord_id INTEGER,
  ord_cust INTEGER,
  ord_date DATE
END RECORD

MAIN
  CALL create_database()
```
CALL run_cust_report()
END MAIN

FUNCTION create_database()
DEFINE d DATE
CONNECT TO ":memory:+driver='dbmsqt'"
CREATE TABLE cust (  
cust_id INTEGER PRIMARY KEY,  
cust_name VARCHAR(30)  
)
INSERT INTO cust VALUES ( 101, "Mike Potter" )
INSERT INTO cust VALUES ( 102, "John Callaghan" )
CREATE TABLE ord (  
ord_id INTEGER PRIMARY KEY,  
ord_cust INTEGER REFERENCES cust(cust_id),  
ord_date DATE  
)
LET d = TODAY - 10
INSERT INTO ord VALUES ( 1001, 101, d )
LET d = TODAY - 1
INSERT INTO ord VALUES ( 1002, 101, d )
LET d = TODAY
INSERT INTO ord VALUES ( 1003, 102, d )
END FUNCTION

FUNCTION run_cust_report()
DEFINE r_cust t_cust
START REPORT cust_report
TO XML HANDLER om.XmlWriter.createFileWriter("output.xml")
DECLARE c_cust CURSOR FOR SELECT * FROM cust ORDER BY cust_id
FOREACH c_cust INTO r_cust.*
OUTPUT TO REPORT cust_report(r_cust.*)
END FOREACH
FINISH REPORT cust_report
END FUNCTION

REPORT cust_report(r_cust)
DEFINE r_cust t_cust
DEFINE r_ord t_ord
OUTPUT
TOP MARGIN 0 BOTTOM MARGIN 0
LEFT MARGIN 0 RIGHT MARGIN 0
PAGE LENGTH 1
FORMAT
ON EVERY ROW
PRINTX NAME = customer r_cust.*
START REPORT ord_report
DECLARE c_ord CURSOR FOR
SELECT * FROM ord WHERE ord_cust = r_cust.cust_id ORDER BY ord_date
FOREACH c_ord INTO r_ord.*
OUTPUT TO REPORT ord_report(r_ord.*)
END FOREACH
FINISH REPORT ord_report
END REPORT

REPORT ord_report(r_ord)
DEFINE r_ord t_ord
OUTPUT
TOP MARGIN 0 BOTTOM MARGIN 0
LEFT MARGIN 0 RIGHT MARGIN 0
PAGE LENGTH 1
FORMAT
ON EVERY ROW
PRINTX NAME = order r_ord.*
<?xml version='1.0' encoding='ASCII'?>
<Report name="main.cust_report" headerLength="0" trailerLength="0"
pageLength="1" topMargin="0" bottomMargin="0" leftMargin="0"
rightMargin="0">
  <PageHeader pageNo="1"/>
  <OnEveryRow>
    <Print name="customer">
      <Item name="r_cust.cust_id" type="INTEGER" value="        101" isoValue="101"/>
      <Item name="r_cust.cust_name" type="VARCHAR(30)" value="Mike Potter"/>
    </Print>
  </OnEveryRow>
  <PageTrailer/>
</Report>

<?xml version='1.0' encoding='ASCII'?>
<Report name="main.ord_report" headerLength="0" trailerLength="0"
pageLength="1" topMargin="0" bottomMargin="0" leftMargin="0"
rightMargin="0">
  <PageHeader pageNo="1"/>
  <OnEveryRow>
    <Print name="order">
      <Item name="r_ord.ord_id" type="INTEGER" value="       1001" isoValue="1001"/>
      <Item name="r_ord.ord_cust" type="INTEGER" value="        101" isoValue="101"/>
      <Item name="r_ord.ord_date" type="DATE" value="11/21/2017" isoValue="2017-11-21"/>
    </Print>
  </OnEveryRow>
  <PageTrailer/>
</Report>

<?xml version='1.0' encoding='ASCII'?>
<Report name="main.cust_report" headerLength="0" trailerLength="0"
pageLength="1" topMargin="0" bottomMargin="0" leftMargin="0"
rightMargin="0">
  <PageHeader pageNo="2"/>
  <OnEveryRow>
    <Print name="customer">
      <Item name="r_cust.cust_id" type="INTEGER" value="        102" isoValue="102"/>
      <Item name="r_cust.cust_name" type="VARCHAR(30)" value="John Callaghan"/>
    </Print>
  </OnEveryRow>
  <PageTrailer/>
</Report>

<?xml version='1.0' encoding='ASCII'?>
<Report name="main.ord_report" headerLength="0" trailerLength="0"
pageLength="1" topMargin="0" bottomMargin="0" leftMargin="0"
rightMargin="0">
  <PageHeader pageNo="1"/>
  <OnEveryRow>
    <Print name="order">
      <Item name="r_ord.ord_id" type="INTEGER" value="       1002" isoValue="1002"/>
      <Item name="r_ord.ord_cust" type="INTEGER" value="        101" isoValue="101"/>
      <Item name="r_ord.ord_date" type="DATE" value="11/30/2017" isoValue="2017-11-30"/>
    </Print>
  </OnEveryRow>
  <PageTrailer/>
</Report>

<?xml version='1.0' encoding='ASCII'?>
<Report name="main.cust_report" headerLength="0" trailerLength="0"
pageLength="1" topMargin="0" bottomMargin="0" leftMargin="0"
rightMargin="0">
  <PageHeader pageNo="2"/>
  <OnEveryRow>
    <Print name="customer">
      <Item name="r_cust.cust_id" type="INTEGER" value="        102" isoValue="102"/>
      <Item name="r_cust.cust_name" type="VARCHAR(30)" value="John Callaghan"/>
    </Print>
  </OnEveryRow>
  <PageTrailer/>
</Report>

<Report name="main.ord_report" headerLength="0" trailerLength="0"
pageLength="1" topMargin="0" bottomMargin="0" leftMargin="0"
rightMargin="0">
  <PageHeader pageNo="1"/>
  <OnEveryRow>
    <Print name="order">
      <Item name="r_ord.ord_id" type="INTEGER" value="       1003" isoValue="1003"/>
    </Print>
  </OnEveryRow>
</Report>
The report driver

The report driver retrieves data, starts the report engine and sends the data (as input records) to be formatted by the REPORT routine.

Usage

A report driver can be part of the MAIN program block, or it can be in one or more functions.

The report driver typically consists of a loop (such as WHILE, FOR, or FOREACH) with the following statements to process the report:

Table 407: Report driver statements

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>START REPORT</td>
<td>This statement is required to instantiate the report driver.</td>
</tr>
<tr>
<td>OUTPUT TO REPORT</td>
<td>Provide data for one row to the report driver.</td>
</tr>
<tr>
<td>FINISH REPORT</td>
<td>Normal termination of the report.</td>
</tr>
<tr>
<td>TERMINATE REPORT</td>
<td>Cancels the processing of the report.</td>
</tr>
</tbody>
</table>

A report driver is started by the START REPORT instruction. Once started, data can be provided to the report driver through the OUTPUT TO REPORT statement. To instruct the report engine to terminate output processing, use the FINISH REPORT instruction. To cancel a report from outside the report routine, use TERMINATE REPORT (from inside the report routine, you cancel the report with EXIT REPORT).

In order to handle report interruption, the report driver can check if the INT_FLAG variable is TRUE in order to stop the loop if the user interrupts the report execution.

It is possible to execute several report drivers at the same time. It is even possible to invoke a report driver inside a REPORT routine, which is different from the current driver.

The programmer must make sure that the runtime system will always execute these instructions in the following order:

1. START REPORT
2. OUTPUT TO REPORT
3. FINISH REPORT
Example

```
SCHEMA stores7
MAIN
  DEFINE rcust RECORD LIKE customer.*
  DATABASE stores7
  DECLARE cul CURSOR FOR SELECT * FROM customer
  LET int_flag = FALSE
  START REPORT myrep
    FOREACH cul INTO rcust.*
      IF int_flag THEN EXIT FOREACH END IF
      OUTPUT TO REPORT myrep(rcust.*)
    END FOREACH
    IF int_flag THEN
      TERMINATE REPORT myrep
    ELSE
      FINISH REPORT myrep
    END IF
  END MAIN
```

Related concepts

**INT_FLAG** on page 569
INT_FLAG is a predefined variable set to TRUE when an interruption event is detected.

**START REPORT**

The **START REPORT** instruction initializes a report execution.

**Syntax**

```
START REPORT report-routine
  [ TO to-clause ]
  [ WITH dimension-option [, . . . ] ]
where to-clause is one of:

  SCREEN
  PRINTER
  [FILE] filename
  PIPE program [ IN FORM MODE | IN LINE MODE ]
  XML HANDLER sax-handler-object
  OUTPUT destination-expr [ DESTINATION { program | filename } ]
```

where **dimension-option** is one of:

```
  LEFT MARGIN = m-left
  RIGHT MARGIN = m-right
  TOP MARGIN = m-top
  BOTTOM MARGIN = m-bottom
  PAGE LENGTH = m-length
  TOP OF PAGE = c-top
```

1. **report-routine** is the name of the REPORT routine.
2. **filename** is a string expression specifying the file that receives report output.
3. **program** is a string expression specifying a program, a shell script, or a command line to receive report output.
4. **destination-expr** is a string expression that specifies one of: SCREEN, PRINTER, FILE, PIPE, PIPE IN LINE MODE, PIPE IN FORM MODE.
5. **sax-handler-object** is a variable referencing an `om.SaxDocumentHandler` instance.
6. *m-left* is the left margin in number of characters. The default is 5.
7. *m-right* is the right margin in number of characters. The default is 132.
8. *m-top* is the top margin in number of lines. The default is 3.
9. *m-bottom* is the bottom margin in number of lines. The default is 3.
10. *m-length* is the total number of lines on a report page. The default page length is 66 lines.
11. *c-top* is a string that defines the page-eject character sequence.

**Usage**

The `START REPORT` statement initializes a report. The instruction allows you to specify the report output destination, the page dimension, and margins.

`START REPORT` typically precedes a loop instruction such as `FOR`, `FOREACH`, or `WHILE` in which `OUTPUT TO REPORT` feeds the report routine with data. After the loop terminates, `FINISH REPORT` completes the processing of the output.

```sql
DEFINE file_name VARCHAR(200), page_size INTEGER
...
START REPORT myrep
  TO FILE file_name
  WITH PAGE LENGTH = page_size
```

If a `START REPORT` statement references a report that is already running, the report is re-initialized; any output might be unpredictable.

**Output specification**

The `TO` clause can be used to specify a destination for output. If you omit the `TO` clause, the Genero runtime system sends report output to the destination specified in the report routine definition. If the report routine does not define an `OUTPUT` clause, the report output is sent by default to the report viewer when in GUI mode, or to the screen when in TUI mode.

Report output can be specified dynamically as follows:

- The `TO FILE` option can specify the `filename` as a character variable that is assigned at runtime.
- The `TO PIPE` option can specify the `program` as a character variable that is assigned at runtime.
- The `TO OUTPUT` option can specify the report output with a string expression, described later in detail.

The `SCREEN` option specifies that output is to the report window. The way the report is displayed to the end user depends on whether you are in TUI mode or GUI mode. In TUI mode, the report output displays to the terminal screen. In GUI mode, the report output displays in a dedicated pop-up window called the Report Viewer. When using `REPORT TO SCREEN` in TUI mode, set a `PAGE LENGTH` no larger than the terminal can display, and include `PAUSE` statements in the `FORMAT` section, to let the end user see the output on the screen.

The `PRINTER` option instructs the runtime system to output the report to the device or program defined by the `DBPRINT` environment variable.

When using the `TO PRINTER` clause, if the `DBPRINT` environment variable is set to the value "FGLSERVER", the document is sent to the printer configured in the Genero Desktop Client (GDC).

When using the `FILE` option, you can specify a file name as the report destination. Output will be sent to the specified file. If the file exists, its content will be overwritten by the new report output. The `FILE` keyword is optional, but it's best to include it to make your code more readable.

The `PIPE` option defines a program, shell script, or command line to which the report output must be sent, using the standard input channel. When using the TUI mode, you can use the `IN [LINE|FORM] MODE` option to specify whether the program is in line mode or in formatted mode when report output is sent to a pipe. For more details about the `IN LINE MODE / IN FORM MODE` clause, see `RUN` on page 490.

The `TO OUTPUT` option allows you to specify one of the output options dynamically at runtime. The character string expression must be one of: "SCREEN", "PRINTER", "FILE", "PIPE", "PIPE IN LINE MODE", " PIPE
IN FORM MODE". If the expression specifies "FILE" or "PIPE", you can also specify a filename or program in a character variable following the DESTINATION keyword.

The XML HANDLER option indicates that the report output will be generated as XML and redirected to a SAX-document handler. When using XML output, the report result can be shown in the Genero Report Engine installed on the front-end workstation. See XML output for more details.

When START REPORT without a destination is used inside a REPORT routine producing XML, the sub-report inherits the XML output target of the parent, and sub-report nodes will be merged into the parent XML output.

Page dimensions specification

The WITH clause defines the dimensions of each report page and the left, top, right and bottom margins. The values corresponding to a margin and page length must be valid integer expressions. The margins can be defined in any order, but a comma "," is required to separate two page dimensions options.

- The LEFT MARGIN clause defines the number of blank spaces to include at the start of each new line of output. The default is 5.
- The RIGHT MARGIN clause defines the total number of characters in each line of output, including the left margin. If you omit this but specify FORMAT EVERY ROW, the default is 132.
- The TOP MARGIN clause specifies how many blank lines appear above the first line of text on each page of output. The default is 3.
- The BOTTOM MARGIN clause specifies how many blank lines follow the last line of output on each page. The default is 3.
- The PAGE LENGTH clause specifies the total number of lines on each page, including data, the margins, and any page headers or page trailers from the FORMAT section. The default page length is 66 lines.

In addition to the page dimension options, the TOP OF PAGE clause can specify a page-eject sequence for a printer. On some systems, specifying this value can reduce the time required for a large report to produce output, because SKIP TO TOP OF PAGE can substitute this value for multiple line feeds.

Tip: To produce reports with no page size limit, set margins to zero and use a page length of 1.

Related concepts

The report routine on page 2039
The report routine implements the body of a report, with formatting instructions.

Two-pass reports on page 2051
The report engine supports two-pass reports, to order rows automatically.

XML output for reports on page 2027
For better integration with external tools based on XML standards, reports can produce XML output.

The SaxDocumentHandler class on page 2617
The om.SaxDocumentHandler class provides an interface to write an XML filter with events.

OUTPUT TO REPORT

The OUTPUT TO REPORT instruction provides a data row to the report execution.

Syntax

OUTPUT TO REPORT report-name ( parameters )

1. report-name is the name of the report to which the parameters are sent.
2. parameters is the data that needs to be sent to the report.

Usage

The OUTPUT TO REPORT instruction feeds the report routine with a single set of data values (called an input record), which corresponds usually to one printed line in the report output.
An input record is the ordered set of values returned by the expressions that you list between the parentheses following the report name in the `OUTPUT TO REPORT` statement. The specified values are passed to the report routine, as part of the input record. The input record typically corresponds to a retrieved row from the database.

The set of values is usually grouped in a `RECORD` variable and best practice is to define a user defined type (`TYPE`) in order to ease the variable definitions required in the code implementing the report driver and the report routine definition, for example:

```plaintext
SCHEMA stores
TYPE t_cust RECORD LIKE customer.*
...
DEFINE r_cust t_cust
...
  OUTPUT TO REPORT cust_report(r_cust. *)
...
REPORT cust_report (r)
  DEFINE r t_cust
...
```

The `OUTPUT TO REPORT` statement is included within a `WHILE` on page 389, `FOR` on page 384, or `FOREACH` loop, so that the program passes data to the report one input record at a time. The example uses a `FOREACH` loop to fetch data from the database and pass it as input record to a report:

```plaintext
SCHEMA stores
DEFINE o LIKE orders.*
...
DECLARE order_c CURSOR FOR
  SELECT orders.*
  FROM orders ORDER BY ord_cust
START REPORT order_list
FOREACH order_c INTO o.*
  OUTPUT TO REPORT order_list(o. *)
END FOREACH
FINISH REPORT order_list
...
```

Special consideration is needed when using `OUTPUT TO REPORT` with row ordering. For example if the report groups rows with `BEFORE GROUP OF` or `AFTER GROUP OF` sections, the rows must be ordered by the column specified in these sections, preferably by the report driver to avoid two-pass reports.

If `OUTPUT TO REPORT` is not executed, none of the control blocks of the report routine are executed, even if the program also includes the `START REPORT` on page 2034 and `FINISH REPORT` on page 2038 statements.

The members of the input record that you specify in the expression list of the `OUTPUT TO REPORT` statement must correspond to elements of the formal argument list in the report definition; in their number and their position. They must be also of compatible data types. At compile time, the number of parameters passed with the `OUTPUT TO REPORT` instruction is not checked against the `DEFINE section in REPORT` on page 2042 of the report routine. This is known behavior in the language.

Arguments of the `TEXT` and `BYTE` data types are passed by reference rather than by value; arguments of other data types are passed by value. A report can use the `WORDWRAP` on page 2059 operator with the `PRINT` on page 2052 statement to display `TEXT` values. A report cannot display `BYTE` values; the character string `<byte value>` in output from the report indicates a `BYTE` value.

**Related concepts**

- The report routine on page 2039
- The report routine implements the body of a report, with formatting instructions.
- Two-pass reports on page 2051
- The report engine supports two-pass reports, to order rows automatically.
- PRINTX on page 2055
Prints an XML formatted row of data in a report, with an additional identifier for XML outputs.

**Primitive Data types** on page 289
Selecting the correct data type assists you in the input, storage, and display of your data.

**FINISH REPORT**
The **FINISH REPORT** instruction finalizes a report execution.

**Syntax**
```
FINISH REPORT report-name
```
1. *report-name* is the name of the report to be ended.

**Usage**
**FINISH REPORT** closes the report driver. Therefore, it must be the last statement in the report driver and must follow a **START REPORT** statement that specifies the name of the same report.
**FINISH REPORT** must be the last statement in the report driver.
**FINISH REPORT** does the following:
1. Completes the second pass, if report is a two-pass report. These 'second pass' activities handle the calculation and output of any aggregate values that are based on all the input records in the report, such as `COUNT (*)` or `PERCENT (*)` with no `GROUP` qualifier.
2. Executes any **AFTER GROUP OF** control blocks.
3. Executes any **PAGE HEADER, ON LAST ROW**, and **PAGE TRAILER** control blocks to complete the report.
4. Copies data from the output buffers of the report to the destination.
5. Closes the Select cursor on any temporary table that was created to order the input records or to perform aggregate calculations.

**Related concepts**
**TERMINATE REPORT** on page 2038
The **TERMINATE REPORT** instruction cancels a report execution.

**EXIT REPORT** on page 2052
Cancels the report processing.

**Two-pass reports** on page 2051
The report engine supports *two-pass reports*, to order rows automatically.

**TERMINATE REPORT**
The **TERMINATE REPORT** instruction cancels a report execution.

**Syntax**
```
TERMINATE REPORT report-name
```
1. *report-name* is the name of the report to be canceled.

**Usage**
**TERMINATE REPORT** cancels the report processing. It is typically used when the program (or the user) becomes aware that a problem prevents the report from producing part of its intended output, or when the user interrupted the report processing.
**TERMINATE REPORT** has the following effects:
- Terminates the processing of the current report.
• Deletes any intermediate files or temporary tables that were created in processing the report.

The **EXIT REPORT** instruction has the same effect, except that it can be used inside the report definition.

**Related concepts**

- **FINISH REPORT** on page 2038
  - The **FINISH REPORT** instruction finalizes a report execution.
- **EXIT REPORT** on page 2052
  - Cancels the report processing.

### The report routine

The *report routine* implements the body of a report, with formatting instructions.

**Syntax 1 (legacy syntax):**

```
[PUBLIC|PRIVATE] REPORT report-name ( parameter-name [, ...] )

  | define-section |
  | output-section |
  | sort-section |
  | format-section |
END REPORT
```

**Syntax 2 (full typed):**

```
[PUBLIC|PRIVATE] REPORT report-name (parameter-name data-type
  attributes ( attribute [ = "value" [, ...] )
  , ... )

  | define-section |
  | output-section |
  | sort-section |
  | format-section |
END REPORT
```

where *define-section* is:

```
DEFINE variable-definition [, ...]
```

where *output-section* is:

```
OUTPUT
  REPORT TO
  | SCREEN |
  | PRINTER |
  | FILE \ filename |
  | PIPE program \ IN FORM MODE \ IN LINE MODE |
  |
  \ attributes ( attribute [ = "value" [, ...] )
  , ... )
  |
  | LEFT MARGIN m-left |
  | RIGHT MARGIN m-right |
  | TOP MARGIN m-top |
  | BOTTOM MARGIN m-bottom |
  | PAGE LENGTH m-length |
```
where `sort-section` is:

```
ORDER [ EXTERNAL ] BY report-variable [, ... ]
```

where `format-section` is:

```
FORMAT EVERY ROW
```

or:

```
FORMAT
  control-block
  [ report-only-fgl-statement
  [ sql-statement
  [ report-statement ]
  [ ... ]
  [ ... ]
```

where `control-block` can be one of:

```
[ FIRST ] PAGE HEADER
  ON EVERY ROW
  BEFORE GROUP OF report-variable
  AFTER GROUP OF report-variable
  PAGE TRAILER
  ON LAST ROW
```

1. `variable-definition` follows the `DEFINE` instruction syntax and declares `report-variables`.
2. `report-variable` is the name of a variable declared in the `DEFINE` section.
3. `report-only-fgl-statement` is a subset of all the regular language statements.
4. `sql-statement` is a valid static SQL statement.

**Usage**

The report definition formats input records. Like the `FUNCTION` or `MAIN` statement, it is a program block that can have the scope of local variables. It is not, however, a function; it is not reentrant, and `CALL` cannot invoke it. The report definition receives data from its driver in sets called input records. These records can include program records, but other data types are also supported. Each input record is formatted and printed as specified by control blocks and statements within the report definition. Most statements and functions can be included in a report definition, and certain specialized statements and operators for formatting output can appear only in a report definition.

Like `MAIN` or `FUNCTION`, the report definition must appear outside any other program block. It must begin with the `REPORT` statement and must end with the `END REPORT` keywords.

Some statements are prohibited in a `REPORT` routine control block. For example, it is not possible to use `CONSTRUCT`, `INPUT`, `DEFER`, `DEFINE`, `REPORT`, `RETURN` instructions in a control block of a report.

By default, report routines are public; they can be called by any other module of the program. If a report routine is only used by the current module, you may want to hide that routine from other modules, to make sure that it will not be called by mistake. To keep a report routine local to the module, add the `PRIVATE` keyword before the report header. Private report routines are only hidden to external modules; all functions of the current module can still call local private report routines.
The **define section** declares the data types of local variables used within the report, and of any variables (the input records) that are passed as arguments to the report by the calling statement. Reports without arguments or local variables do not require a **DEFINE** section.

The **output-section** can set margin and page size values, and can also specify where to send the formatted output. Output from the report consists of successive pages, each containing a fixed number of lines whose margins and maximum number of characters are fixed.

The **sort-section** specifies how the rows have to be sorted. The specified sort order determines the order in which the runtime system processes any **GROUP** of control blocks in the **FORMAT** section.

The **format-section** is required. It specifies the appearance of the report, including page headers, page trailers, and aggregate functions of the data. It can also contain control blocks that specify actions to take before or after specific groups of rows are processed. (Alternatively, it can produce a default report by only specifying **FORMAT EVERY ROW**).

**Related concepts**

- **OUTPUT section in REPORT** on page 2043
  Specifies report destination and page format options.
- **ORDER BY section in REPORT** on page 2044
  Forces a sort order of unsorted data rows in reports.
- **FORMAT section in REPORT** on page 2045
  Defines the formatting directives inside a report routine.

**The report prototype**

When defining a report routine, the report name must immediately follow the **REPORT** keyword. The name must be unique among function and report names within the program. Its scope is the entire program.

The list of formal arguments of the report must be enclosed in parentheses and separated by commas. These are local variables that store values that the calling routine passes to the report. The compiler issues an error unless you declare their data types in the subsequent **DEFINE** section of the report routine. You can include a program record in the formal argument list, but you cannot append the .* symbols to the name of the record. Arguments can be of any data type except ARRAY, or a record with an ARRAY member.

When you call a report, the formal arguments are assigned values from the argument list of the **OUTPUT TO REPORT** statement. These actual arguments that you pass must match, in number and position, the formal arguments of the **REPORT** routine. The data types must be compatible, but they need not be identical. The runtime system can perform some conversions between compatible data types.

The names of the actual arguments and the formal arguments do not have to match.

You must include the following items in the list of formal arguments:

- All the values for each row sent to the report in the following cases:
  - If you include an **ORDER BY** section or **GROUP PERCENT** (*) function
  - If you use a global aggregate function (one over all rows of the report) anywhere in the report, except in the **ON LAST ROW** control block
  - If you specify the **FORMAT EVERY ROW** default format
  - Any variables referenced in the following group control blocks:
    - **AFTER GROUP OF**
    - **BEFORE GROUP OF**

**Related concepts**

- **Primitive Data types** on page 289
Selecting the correct data type assists you in the input, storage, and display of your data.

**DEFINE section in REPORT**
Defines report parameters and local variables.

**Syntax**
The syntax of the report DEFINE section is the same as for the DEFINE statement, except that you cannot define arrays or array members for records.

**Usage**
This section declares a data type for each formal argument in the REPORT prototype and for any additional local variables that can be referenced only within the REPORT program block. The DEFINE section is required if you pass arguments to the report or if you reference local variables in the report.

For declaring local report variables, the same rules apply to the DEFINE section as to the DEFINE statement in MAIN and FUNCTION program blocks. There are some differences and exceptions, however:

- Report arguments cannot be of type ARRAY.
- Report arguments cannot be records that include ARRAY members.
- Report local variables are not allocated on the stack at every OUTPUT TO REPORT call. The scope of the variables in the DEFINE section is local to the report routine, but the lifetime is the duration of the program, like module or global variables. In other words, report variables persist across OUTPUT TO REPORT calls.

Data types of local variables that are not formal arguments are unrestricted. You must include arguments in the report prototype and declare them in the DEFINE section, if any of the following conditions is true:

- If you specify FORMAT EVERY ROW to create a default report, you must pass all the values for each record of the report.
- If an ORDER BY section is included, you must pass all the values that ORDER BY references for each input record of the report.
- If you use the AFTER GROUP OF control block, you must pass at least the arguments that are named in that control block.
- If an aggregate that depends on all records of the report appears anywhere except in the ON LAST ROW control block, you must pass each of the records of the report through the argument list.

Aggregates dependent on all records include:

- GROUP PERCENT(*) (anywhere in a report).
- Any aggregate without the GROUP keyword (anywhere outside the ON LAST ROW control block).

If your report calls an aggregate function, an error might result if any argument of an aggregate function is not also a format argument of the report. You can, however, use global or module variables as arguments of aggregates if the value of the variable does not change while the report is executing.

A report can reference variables of global or module scope that are not declared in the DEFINE section. Their values can be printed, but they can cause problems in aggregates and in BEFORE GROUP OF and AFTER GROUP OF clauses. Any references to non-local variables can produce unexpected results, however, if their values change while a two-pass report is executing.

**Related concepts**
Two-pass reports on page 2051
The report engine supports two-pass reports, to order rows automatically.

OUTPUT section in REPORT

Specifies report destination and page format options.

Syntax

```
OUTPUT
  REPORT TO
    SCREEN
    PRINTER
    FILE filename
    PIPE IN FORM MODE IN LINE MODE program
  LEFT MARGIN m-left
  RIGHT MARGIN m-right
  TOP MARGIN m-top
  BOTTOM MARGIN m-bottom
  PAGE LENGTH m-length
  TOP OF PAGE c-top
```

1. `program` defines the name of a program, shell script, command receiving the output.
2. `filename` defines the file which receives the output of the report.
3. `m-left` is the left margin in number of characters. The default is 5.
4. `m-right` is the right margin in number of characters. The default is 132.
5. `m-top` is the top margin in number of lines. The default is 3.
6. `m-bottom` is the bottom margin in number of lines. The default is 3.
7. `m-length` is the total number of lines on a report page. The default page length is 66 lines.
8. `c-top` is a string that defines the page-eject character sequence.

Usage

The OUTPUT section can specify the destination and dimensions for output from the report and the page-eject sequence for the printer. If you omit the OUTPUT section, the report uses default values to format each page. This section is superseded by any corresponding START REPORT specifications.

The OUTPUT section can direct the output from the report to a printer, file, or pipe, and can initialize the page dimensions and margins of report output. If PRINTER is specified, the DBPRINT environment variable specifies which printer.

The START REPORT statement of the report driver can override all of these specifications by assigning another destination in its TO clause or by assigning other dimensions, margins, or another page-eject sequence in the WITH clause.

As the size specifications for the dimensions and margins of a page of report output that the OUTPUT section can specify must be literal integers, consider defining page dimensions in the START REPORT statement, where you can use variables to assign these values dynamically at runtime.

**Tip:** To produce reports with not page size limit, set margins to zero and use a page length of 1.

Related concepts

START REPORT on page 2034
The START REPORT instruction initializes a report execution.

**ORDER BY section in REPORT**
Forces a sort order of unsorted data rows in reports.

**Syntax**

```
ORDER [ EXTERNAL ] BY report-variable [ DESC | ASC ] [, ... ]
```

1. *report-variable* identifies one of the variables passed to the report routine to be used for sorting rows.

**Usage**

When grouping rows in a report, values that the report definition receives from the report driver are significant in determining how `BEFORE GROUP OF / AFTER GROUP OF` control blocks will process the data in the formatted report output.

The `ORDER BY` section defines how the variables of the input records are to be sorted. It is required if the report driver does not send sorted data to the report. The specified sort order determines the order in which the runtime system processes any `GROUP OF` control blocks in the `FORMAT` section.

If you omit the `ORDER BY` section, the runtime system processes input records in the order received from the report driver and processes any `GROUP OF` control blocks in their order of appearance in the `FORMAT` section. If records are not sorted in the report driver, the `GROUP OF` control blocks might be executed at random intervals (that is, after any input record) because unsorted values tend to change from record to record.

If you specify only one variable in the `GROUP OF` control blocks, and the input records are already sorted in sequence on that variable by the `SELECT` statement, you do not need to include an `ORDER BY` section in the report.

Specify `ORDER EXTERNAL` BY if the input records have already been sorted by the `SELECT` statement used by the report driver. The list of variables after the keywords `ORDER EXTERNAL` BY control the execution order of `GROUP BY` control blocks.

Without the `EXTERNAL` keyword, the report becomes a two-pass report, meaning that the report engine processes the set of input records twice. During the first pass, the report engine sorts the data and stores the sorted values in a temporary table in the database. During the second pass, it calculates any aggregate values and produces output from data in the temporary files.

With the `EXTERNAL` keyword, the report engine only needs to make a single pass through the data: it does not need to build the temporary table in the database for sorting the data. However, if the report routine contains aggregations functions such as `GROUP PERCENT (*)`, the report will become a two-pass report because such aggregation function needs all rows to compute the value.

The `DESC` or `ASC` clause defines the sort order.

**Related concepts**

- [Two-pass reports](#) on page 2051
  The report engine supports two-pass reports, to order rows automatically.

- [OUTPUT section in REPORT](#) on page 2043
  Specifies report destination and page format options.

- [The report driver](#) on page 2033
The *report driver* retrieves data, starts the report engine and sends the data (as input records) to be formatted by the REPORT routine.

**FORMAT section in REPORT**

Defines the formatting directives inside a report routine.

**Syntax**

Default format:

```
FORMAT EVERY ROW
```

Custom format:

```
FORMAT
  control-block
    report-statement
    report-only-fgl-statement
    sql-statement
  [...]
  [...]
```

where *control-block* can be one of:

```
FIRST PAGE HEADER
ON EVERY ROW
BEFORE GROUP OF report-variable
AFTER GROUP OF report-variable
PAGE TRAILER
ON LAST ROW
```

1. *report-statement* is any report-specific instruction.
2. *report-only-fgl-statement* is any language instruction supported in the report routine.
3. *sql-statement* is any SQL statement supported by the language.
4. *report-variable* is the name of a variable declared in the DEFINE section.

**Usage**

A report definition must contain a FORMAT section.

The FORMAT section determines how the output from the report will look. It works with the values that are passed to the REPORT program block through the argument list or with global or module variables in each record of the report. In a source file, the FORMAT section begins with the FORMAT keyword and ends with the END REPORT keywords.

The FORMAT section is made up of the following control blocks:

- **FIRST PAGE HEADER**
- **PAGE HEADER**
- **PAGE TRAILER**
- **BEFORE GROUP OF**
- **AFTER GROUP OF**
- **ON EVERY ROW**
- **ON LAST ROW**

If you use the FORMAT EVERY ROW, no other statements or control blocks are valid. The EVERY ROW keywords specify a default output format, including every input record that is passed to the report.
Control blocks define the structure of a report by specifying one or more statements to be executed when specific parts of the report are processed.

If a report driver includes START REPORT and FINISH REPORT statements, but no data records are passed to the report, no control blocks are executed. That is, unless the report executes an OUTPUT TO REPORT statement that passes at least one input record to the report; then neither the FIRST PAGE HEADER control block nor any other control block is executed.

Apart from BEFORE GROUP OF and AFTER GROUP OF, each control block must appear only once.

More complex FORMAT sections can contain control blocks like ON EVERY ROW or BEFORE GROUP OF, which contain statements to execute while the report is being processed. Control blocks can contain report execution statements and other executable statements.

A control block may invoke most language statements, except those listed in prohibited statements.

The BEFORE/AFTER GROUP OF control blocks can include aggregate functions to instruct the report engine to automatically compute such values.

A report-statement is a statement specially designed for the report format section. It cannot be used in any other part of the program.

The sequence in which the BEFORE GROUP OF and AFTER GROUP OF control blocks are executed depends on the sort list in the ORDER BY section, regardless of the physical sequence in which these control blocks appear within the FORMAT section.

Related concepts
Prohibited report routine statements on page 2051

FORMAT EVERY ROW
Default format specification of a report.

A report routine written with FORMAT EVERY ROW formats the report in a simple default format, containing only the values that are passed to the REPORT program block through its arguments, and the names of the arguments. You cannot modify the EVERY ROW statement with any of the statements listed in report execution statements, and neither can you include any control blocks in the FORMAT section.

The report engine uses as column headings the names of the variables that the report driver passes as arguments at runtime. If all fields of each input record can fit horizontally on a single line, the default report prints the names across the top of each page and the values beneath. Otherwise, it formats the report with the names down the left side of the page and the values to the right, as in the previous example. When a variable contains a null value, the default report prints only the name of the variable, with nothing for the value.

The following example is a brief report specification that uses FORMAT EVERY ROW. We assume here that the cursor that retrieved the input records for this report was declared with an ORDER BY clause, so that no ORDER BY section is needed in this report definition:

```plaintext
DATABASE stores7

REPORT simple( order_num, customer_num, order_date )

DEFINE order_num LIKE orders.order_num,
     customer_num LIKE orders.customer_num,
     order_date LIKE orders.order_date

FORMAT EVERY ROW

END REPORT
```

The example would produce the following output:

```
order_num  customer_num  order_date
1001       104         01/20/1993
```
### FIRST PAGE HEADER

Defines the printing commands for the first page of a report.

This control block specifies the action that the runtime system takes before it begins processing the first input record. You can use it, for example, to specify what appears near the top of the first page of output from the report.

As the runtime system executes the FIRST PAGE HEADER control block before generating any output, you can use this control block to initialize variables that you use in the FORMAT section.

If a report driver includes START REPORT and FINISH REPORT statements, but no data records are passed to the report, this control block is not executed. That is, unless the report executes an OUTPUT TO REPORT statement that passes at least one input record to the report, neither the FIRST PAGE HEADER control block nor any other control block is executed.

As its name implies, you can also use a FIRST PAGE HEADER control block to produce a title page as well as column headings. On the first page of a report, this control block overrides any PAGE HEADER control block. That is, if both a FIRST PAGE HEADER and a PAGE HEADER control block exist, output from the first appears at the beginning of the first page, and output from the second begins all subsequent pages.

The TOP MARGIN (set in the OUTPUT section) determines how close the header appears to the top of the page.

Consider the following notes when programming the FIRST PAGE HEADER control block:

1. You cannot include a SKIP integer LINES statement inside a loop within this control block.
2. The NEED statement is not valid within this control block.
3. If you use an IF…THEN…ELSE statement within this control block, the number of lines displayed by any PRINT statements following the THEN keyword must be equal to the number of lines displayed by any PRINT statements following the ELSE keyword.
4. If you use a CASE, FOR, or WHILE statement that contains a PRINT statement within this control block, you must terminate the PRINT statement with a semicolon (;). The semicolon suppresses any LINEFEED characters in the loop, keeping the number of lines in the header constant from page to page.
5. You cannot use a PRINT file name statement to read and display text from a file within this control block.

Corresponding restrictions also apply to CASE, FOR, IF, NEED, SKIP, PRINT, and WHILE statements in PAGE HEADER and PAGE TRAILER control blocks.

**Related concepts**

Report instructions on page 2052

The report instruction listed in this section can appear only in control blocks of the FORMAT section of a report routine.

### PAGE HEADER

Defines the printing commands for the top of all pages of a report.

This control block is executed whenever a new page is added to the report. The PAGE HEADER control block specifies the action that the runtime takes before it begins processing each page of the report. It can specify what information, if any, appears at the top of each new page of output from the report.

The TOP MARGIN specification (in the OUTPUT section) affects how many blank lines appear above the output produced by statements in the PAGE HEADER control block.

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1002</td>
<td>101</td>
<td>06/01/1993</td>
</tr>
<tr>
<td>1003</td>
<td>104</td>
<td>10/12/1993</td>
</tr>
<tr>
<td>1004</td>
<td>106</td>
<td>04/12/1993</td>
</tr>
<tr>
<td>1005</td>
<td>116</td>
<td>12/04/1993</td>
</tr>
<tr>
<td>1006</td>
<td>112</td>
<td>09/19/1993</td>
</tr>
<tr>
<td>1007</td>
<td>117</td>
<td>03/25/1993</td>
</tr>
<tr>
<td>1008</td>
<td>110</td>
<td>11/17/1993</td>
</tr>
<tr>
<td>1009</td>
<td>111</td>
<td>02/14/1993</td>
</tr>
<tr>
<td>1010</td>
<td>115</td>
<td>05/29/1993</td>
</tr>
<tr>
<td>1011</td>
<td>104</td>
<td>03/23/1993</td>
</tr>
<tr>
<td>1012</td>
<td>117</td>
<td>06/05/1993</td>
</tr>
</tbody>
</table>
```
You can use the `PAGENO` operator in a `PRINT` statement within a `PAGE HEADER` control block to automatically display the current page number at the top of every page.

The `FIRST PAGE HEADER` control block overrides this control block on the first page of a report.

New group values can appear in the `PAGE HEADER` control block when this control block is executed after a simultaneous end-of-group and end-of-page situation.

The runtime system delays the processing of the `PAGE HEADER` control block until it encounters the first `PRINT`, `SKIP`, or `NEED` statement in the `ON EVERY ROW`, `BEFORE GROUP OF`, or `AFTER GROUP OF` control block. This order guarantees that any group columns printed in the `PAGE HEADER` control block have the same values as the columns printed in the `ON EVERY ROW` control block.

**Important:** The restrictions that apply to `FIRST PAGE HEADER` also apply to `PAGE HEADER`.

**Related concepts**

- **PAGE TRAILER** on page 2048
  Defines the printing commands for the tail of all pages of a report.

- **Report operators** on page 2058
  Report operators can be used to print dynamic report information.

**PAGE TRAILER**

Defines the printing commands for the tail of all pages of a report.

The `PAGE TRAILER` control block specifies what information, if any, appears at the bottom of each page of output from the report.

The runtime system executes the statements in the `PAGE TRAILER` control block before the `PAGE HEADER` control block when a new page is needed. New pages can be initiated by any of the following conditions:

- `PRINT` attempts to print on a page that is already full.
- `SKIP TO TOP OF PAGE` is executed.
- `SKIP n LINES` specifies more lines than are available on the current page.
- `NEED` specifies more lines than are available on the current page.

You can use the `PAGENO` operator in a `PRINT` statement within a `PAGE TRAILER` control block to automatically display the page number at the bottom of every page, as in this example:

```
PAGE TRAILER
  PRINT COLUMN 28, PAGENO USING "page <<<<"
```

The `BOTTOM MARGIN` specification (in the `OUTPUT` section) affects how close to the bottom of the page the output displays the page trailer.

**Important:** The restrictions that apply to `FIRST PAGE HEADER` also apply to `PAGE TRAILER`.

**Related concepts**

- **PAGE HEADER** on page 2047
  Defines the printing commands for the top of all pages of a report.

- **Report operators** on page 2058
  Report operators can be used to print dynamic report information.

**BEFORE/AFTER GROUP OF**

Defines printing commands of row grouping sections within a report.

The `BEFORE/AFTER GROUP OF` control blocks specify what action the runtime system takes respectively before or after it processes a group of input records. Group hierarchy is determined by the `ORDER BY` specification in the `SELECT` statement or in the report definition.

A group of records is all of the input records that contain the same value for the variable whose name follows the `AFTER GROUP OF` keywords. This group variable must be passed through the report arguments. A report can include no more than one `AFTER GROUP OF` control block for any group variable.
When the runtime system executes the statements in a BEFORE/AFTER GROUP OF control block, the report variables have the values from the first / last record of the new group. From this perspective, the BEFORE/AFTER GROUP OF control block can be thought of as the "on first / last record of group" control block.

Each BEFORE GROUP OF block is executed in order, from highest to lowest priority, at the start of a report (after any FIRST PAGE HEADER or PAGE HEADER control blocks, but before processing the first record) and on these occasions:

- Whenever the value of the group variable changes (after any AFTER GROUP OF block for the old value completes execution)
- Whenever the value of a higher-priority variable in the sort list changes (after any AFTER GROUP OF block for the old value completes execution)

The runtime system executes the AFTER GROUP OF control block on these occasions:

- Whenever the value of the group variable changes.
- Whenever the value of a higher-priority variable in the sort list changes.
- At the end of the report (after processing the last input record but before the runtime system executes any ON LAST ROW or PAGE TRAILER control blocks). In this case, each AFTER GROUP OF control block is executed in ascending priority.

How often the value of the group variable changes depends in part on whether the input records have been sorted by the SELECT statement:

- If records are already sorted, the BEFORE/AFTER GROUP OF block executes before the runtime system processes the first record of the group.
- If records are not sorted, the BEFORE GROUP OF block might be executed after any record because the value of the group variable can change with each record. If no ORDER BY section is specified, all BEFORE/AFTER GROUP OF control blocks are executed in the same order in which they appear in the FORMAT section. The BEFORE/AFTER GROUP OF control blocks are designed to work with sorted data.

You can sort the records by specifying a sort list in either of the following areas:

- An ORDER BY section in the report definition
- The ORDER BY clause of the SELECT statement in the report driver

To sort data in the report definition (with an ORDER BY section), make sure that the name of the group variable appears in both the ORDER BY section and in the BEFORE GROUP OF control block.

To sort data in the ORDER BY clause of a SELECT statement, perform the following tasks:

- Use the column name in the ORDER BY clause of the SELECT statement as the group variable in the BEFORE GROUP OF control block.
- If the report contains BEFORE or AFTER GROUP OF control blocks, make sure that you include an ORDER EXTERNAL BY section in the report to specify the precedence of variables in the sort list.

If you specify sort lists in both the report driver and the report definition, the sort list in the ORDER BY section of the REPORT takes precedence. When the runtime system starts to generate a report, it first executes the BEFORE GROUP OF control blocks in descending order of priority before it executes the ON EVERY ROW control block. If the report is not already at the top of the page, the SKIP TO TOP OF PAGE statement in a BEFORE GROUP OF control block causes the output for each group to start at the top of a page.

If the sort list includes more than one variable, the runtime system sorts the records by values in the first variable (highest priority). Records that have the same value for the first variable are then ordered by the second variable and so on until records that have the same values for all other variables are ordered by the last variable (lowest priority) in the sort list.

The ORDER BY section determines the order in which the runtime system processes BEFORE GROUP OF and AFTER GROUP OF control blocks. If you omit the ORDER BY section, the runtime system processes any GROUP OF control blocks in the lexical order of their appearance within the FORMAT section.

If you include an ORDER BY section, and the FORMAT section contains more than one BEFORE GROUP OF or AFTER GROUP OF control block, the order in which these control blocks are executed is determined by the sort list.
in the ORDER BY section. In this case, their order within the FORMAT section is not significant because the sort list overrides their lexical order.

The runtime system processes all the statements in a BEFORE GROUP OF or AFTER GROUP OF control block on these occasions:

- Each time the value of the current group variable changes.
- Each time the value of a higher-priority variable changes.

How often the value of the group variable changes depends in part on whether the input records have been sorted. If the records are sorted, AFTER GROUP OF executes after the runtime system processes the last record of the group of records; BEFORE GROUP OF executes before the runtime system processes the first records with the same value for the group variable. If the records are not sorted, the BEFORE GROUP OF and AFTER GROUP OF control blocks might be executed before and after each record because the value of the group variable might change with each record. All the AFTER GROUP OF and BEFORE GROUP OF control blocks are executed in the same lexical order in which they appear in the FORMAT section.

In the AFTER GROUP OF control block, you can include the GROUP keyword to qualify aggregate report functions like AVG(), SUM(), MIN(), or MAX():

```
AFTER GROUP OF r.order_num
  PRINT r.order_date, 7 SPACES,
        r.order_num USING"###&", 8 SPACES,
        r.ship_date, ", "
        GROUP SUM(r.total_price) USING"$$$$,$$$,$$$.&"
AFTER GROUP OF r.customer_num
  PRINT 42 SPACES, "-------------------"
  PRINT 42 SPACES, GROUP SUM(r.total_price) USING"$$$$,$$$,$$$.&"
```

Using the GROUP keyword to qualify an aggregate function is only valid within the AFTER GROUP OF control block. It is not valid, for example, in the BEFORE GROUP OF control block.

After the last input record is processed, the runtime system executes the AFTER GROUP OF control blocks before it executes the ON LAST ROW control block.

**Related concepts**

- **Two-pass reports** on page 2051
  The report engine supports two-pass reports, to order rows automatically.

- **PRINT** on page 2052
  Formats and prints a row of data in a report routine.

- **PRINTX** on page 2055
  Prints an XML formatted row of data in a report, with an additional identifier for XML outputs.

**ON EVERY ROW**

Defines printing commands for each row in a report.

The ON EVERY ROW control block specifies the action to be taken by the runtime system for every input record that is passed to the report definition.

The runtime system executes the statements within the ON EVERY ROW control block for each new input record that is passed to the report. The following example is from a report that lists all the customers, their addresses, and their telephone numbers across the page:

```
ON EVERY ROW
  PRINT r.fname, ", " r.lname, ", "
        r.address1, ", " r.cust_phone
```

The runtime system delays processing the PAGE HEADER control block (or the FIRST PAGE HEADER control block, if it exists) until it encounters the first PRINT, SKIP, or NEED statement in the ON EVERY ROW control block.
If a **BEFORE GROUP OF** control block is triggered by a change in the value of a variable, the runtime system executes all appropriate **BEFORE GROUP OF** control blocks (in the order of their priority) before it executes the **ON EVERY ROW** control block. Similarly, if execution of an **AFTER GROUP OF** control block is triggered by a change in the value of a variable, the runtime system executes all appropriate **AFTER GROUP OF** control blocks (in the reverse order of their priority) before it executes the **ON EVERY ROW** control block.

**Related concepts**

**PRINT** on page 2052
Formats and prints a row of data in a report routine.

**PRINTX** on page 2055
Prints an XML formatted row of data in a report, with an additional identifier for XML outputs.

**ON LAST ROW**
Defines the printing commands of the last row in a report.

The **ON LAST ROW** control block specifies the action that the runtime system is to take after it processes the last input record that was passed to the report definition and encounters the **FINISH REPORT** statement.

The statements in the **ON LAST ROW** control block are executed after the statements in the **ON EVERY ROW** and **AFTER GROUP OF** control blocks if these blocks are present.

When the runtime system processes the statements in an **ON LAST ROW** control block, the variables that the report is processing still have the values from the final record that the report processed. The **ON LAST ROW** control block can use aggregate functions to display report totals.

**Related concepts**

**PRINT** on page 2052
Formats and prints a row of data in a report routine.

**PRINTX** on page 2055
Prints an XML formatted row of data in a report, with an additional identifier for XML outputs.

**Prohibited report routine statements**

Language statements that have no meaning inside a report definition routine are prohibited. These statements are some of the statements that are not valid within any control block of the **FORMAT** section of a **REPORT** program block, such as interactive statements (**CONSTRUCT, INPUT, DIALOG, MENU**), program block definitions (**FUNCTION, REPORT**), and some flow control instructions like **RETURN**.

A compile-time error is issued if you attempt to include any of these statements in a control block of a report. You can call a function that includes some of these statements, but this is not recommended.

**Related concepts**

**Report instructions** on page 2052
The report instruction listed in this section can appear only in control blocks of the **FORMAT** section of a report routine.

**Two-pass reports**

The report engine supports **two-pass reports**, to order rows automatically.

The one-pass report requires sorted data to be produced by the report driver in order to handle **BEFORE/AFTER GROUP OF** blocks properly. The two-pass report handles sorts internally and does not need sorted data from the report driver. During the first pass, the report engine sorts the data and stores the sorted values in a temporary file in the database. During the second pass, it calculates any aggregate values and produces output from data in the temporary files.

A report is defined as a two-pass report if it includes any of the following items:

- An **ORDER BY** section without the **EXTERNAL** keyword.
• The `GROUP PERCENT(*)` aggregate function anywhere in the report.
• Any `aggregate` function that has no GROUP keyword in any control block other than `ON LAST ROW`.

Two-pass reports create temporary tables. The `FINISH REPORT` statement uses values from these tables to calculate any global aggregates, and then deletes the tables. Since two-pass reports create temporary tables, the report engine requires a database connection, and the database server must support temporary tables with indexes.

Consider avoiding two-pass reports when a regular report is possible.

**Related concepts**
The report routine on page 2039
The `report routine` implements the body of a report, with formatting instructions.

---

**Report instructions**
The report instruction listed in this section can appear only in control blocks of the `FORMAT` section of a report routine.

**EXIT REPORT**
Cancels the report processing.

**Syntax**

```plaintext
EXIT REPORT
```

**Usage**

EXIT REPORT cancels the report processing. It must appear in the `FORMAT` section of the report definition. It is useful after the program (or the user) becomes aware that a problem prevents the report from producing part of its intended output.

EXIT REPORT has the following effects:
• Terminates the processing of the current report.
• Deletes any intermediate files or temporary tables that were created in processing the report.

**Note:** Do not use the `RETURN` statement as a substitute for `EXIT REPORT`. An error is issued if `RETURN` is encountered within the definition of a report.

**Related concepts**
`FORMAT section in REPORT` on page 2045
Defines the formatting directives inside a report routine.

**PRINT**
Formats and prints a row of data in a report routine.

**Syntax**

```plaintext
PRINT
  \[ expression
  \| COLUMN left-offset
  \| PAGENO
  \| LINENO
  \| num-spaces SPACES
  \| [GROUP] COUNT(*) \| WHERE condition \|
  \| [GROUP] PERCENT(*) \| WHERE condition\|
  \| [GROUP] AVG( variable ) \| WHERE condition\|
```
| REPORTS |

1. `expression` is any legal language expression.
2. `left-offset` is described in `COLUMN`.
3. `num-spaces` is described in `SPACES`.
4. `char-expression` is a string expression or a `TEXT` variable.
5. `file-name` is a string expression, or a quoted string, that specifies the name of a text file to include in the output from the report.

**Usage**

The PRINT instruction is used in a report routine to output a line of data.

The PRINT statement can include character data in the form of an ASCII file, a TEXT variable, or a comma-separated expression list of character expressions in the output of the report. (For TEXT variable or file name, you cannot specify additional output in the same PRINT statement.)

If a BYTE value is used in the PRINT statement, the output will show the "<byte value>" text for this element when the report output is regular text. If the report output is XML, the BYTE value is converted to Base64 before it is written to the output stream.

PRINT statement output begins at the current character position, sometimes called simply the current position. On each page of a report, the initial default character position is the first character position in the first line. This position can be offset horizontally and vertically by margin and header specifications and by executing any of the following statements:

- The SKIP statement moves it down to the left margin of a new line.
- The NEED statement can conditionally move it to a new page.
- The PRINT statement moves it horizontally (and sometimes down).

Unless you use the keyword CLIPPED or USING, values are displayed with widths (including any sign) that depend on their declared data types.

**Table 408: Default print width for data types**

<table>
<thead>
<tr>
<th>Data type</th>
<th>Default Print Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE</td>
<td>N/A</td>
</tr>
<tr>
<td>CHAR</td>
<td>Length of character data type declaration.</td>
</tr>
<tr>
<td>DATE</td>
<td>DBDATE dependent, 10 if DBDATE = &quot;MDY4/&quot;</td>
</tr>
<tr>
<td>DATETIME</td>
<td>From 2 to 25, as implied in the data type declaration.</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>((2 + p + s)), where p is the precision and s is the scale from the data type declaration.</td>
</tr>
<tr>
<td>FLOAT</td>
<td>14</td>
</tr>
<tr>
<td>INTEGER</td>
<td>11</td>
</tr>
<tr>
<td>INTERVAL</td>
<td>From 3 to 25, as implied in the data type declaration.</td>
</tr>
<tr>
<td>MONEY</td>
<td>((2 + c + p + s)), where c is the length of the currency defined by DBMONEY and p is the precision and s is the scale from the data type declaration.</td>
</tr>
</tbody>
</table>
### Data type

<table>
<thead>
<tr>
<th>Data type</th>
<th>Default Print Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>NCHAR</td>
<td>Length of character data type declaration.</td>
</tr>
<tr>
<td>NVARCHAR</td>
<td>Length current value in the variable.</td>
</tr>
<tr>
<td>SMALLFLOAT</td>
<td>14</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>6</td>
</tr>
<tr>
<td>STRING</td>
<td>Length current value in the variable.</td>
</tr>
<tr>
<td>TEXT</td>
<td>Length current value in the variable.</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>Length current value in the variable.</td>
</tr>
</tbody>
</table>

Unless you specify the FILE or WORDWRAP option, each PRINT statement displays output on a single line. For example, this fragment displays output on two lines:

```plaintext
PRINT fname, lname
PRINT city, "", state, " ", zip-code
```

If you terminate a PRINT statement with a semicolon. However, you suppress the implicit LINEFEED character at the end of the line. The following example has the same effect as the PRINT statements in the previous example:

```plaintext
PRINT fname;
PRINT lname
PRINT city, "", state, " ", zip-code
```

The expression list of a PRINT statement returns one or more values that can be displayed as printable characters. The expression list can contain report variables, built-in functions, and operators. Some of these can appear only in a REPORT program block such as PAGENO, LINENO, PERCENT.

If the expression list applies the USING operator to format a DATE or MONEY value, the format string of the USING operator takes precedence over the DBDATE, DBMONEY, and DBFORMAT environment variables.

The PRINT FILE file-name statement reads the contents of the specified file into the report, beginning at the current character position. This statement permits you to insert a multiple-line character string into the output of a report. If file-name stores the value of a TEXT variable, the PRINT FILE file-name statement has the same effect as specifying PRINT text-variable. (But only PRINT variable can include the WORDWRAP operator)

Aggregate report functions summarize data from several records in a report. The syntax and effects of aggregates in a report resemble those of SQL aggregate functions but are not identical.

The expression (in parentheses) that SUM(), AVG(), MIN(), or MAX() takes as an argument is typically of a number or INTERVAL data type; ARRAY, BYTE, RECORD, and TEXT are not valid. The SUM(), AVG(), MIN(), and MAX() aggregates ignore input records for which their arguments have null values, but each returns NULL if every record has a null value for the argument.

The GROUP keyword is an optional keyword that causes the aggregate function to include data only for a group of records that have the same value for a variable that you specify in an AFTER GROUP OF control block. An aggregate function can only include the GROUP keyword within an AFTER GROUP OF control block.

The optional WHERE clause allows you to select among records passed to the report, so that only records for which the boolean expression is TRUE are included.

### Example

The following example is from the FORMAT section of a report definition that displays both quoted strings and values from rows of the customer table:

```plaintext
FIRST PAGE HEADER
  PRINT COLUMN 30, "CUSTOMER LIST"
  SKIP 2 LINES
```
PRINT "Listings for the State of ", thisstate
SKIP 2 LINES
PRINT "NUMBER", COLUMN 12, "NAME", COLUMN 35, "LOCATION",
    COLUMN 57, "ZIP", COLUMN 65, "PHONE"
SKIP 1 LINE
PAGE HEADER
PRINT "NUMBER", COLUMN 12, "NAME", COLUMN 35, "LOCATION",
    COLUMN 57, "ZIP", COLUMN 65, "PHONE"
SKIP 1 LINE
ON EVERY ROW
PRINT customer_num USING "###&", COLUMN 12, fname CLIPPED,
    1 SPACE, lname CLIPPED, COLUMN 35, city CLIPPED, ", ",
    state, COLUMN 57, zip-code, COLUMN 65, phone

Related concepts
PRINTX on page 2055
Prints an XML formatted row of data in a report, with an additional identifier for XML outputs.

COLUMN on page 354
The COLUMN operator generates blanks.

PRINTX
Prints an XML formatted row of data in a report, with an additional identifier for XML outputs.

Syntax

PRINTX [NAME = identifier] expression

1. identifier is the name to be used in the XML node.
2. expression is any legal language expression.

Usage

The PRINTX statement is similar to PRINT, except that when XML is produced by the report, the XML print element will be named as specified. If the NAME clause is omitted or the report is run in non-XML mode, then PRINTX does exactly the same as PRINT.

To generate XML output, you must redirect the report output into a SAX document handler with the TO XML HANDLER clause of START REPORT:

START REPORT orders_report
    TO XML HANDLER om.XmlWriter.createFileWriter("orders.xml")

Note that when using XML output, BYTE values are converted to Base64 before they are printed with the PRINTX instruction.

Example

REPORT (fname, lname, ...)
    DEFINE fname VARCHAR(20),
        lname VARCHAR(20)
    ...
    FORMAT
        ...
        ON EVERY ROW
            PRINTX NAME=customer fname, lname
    ...

With the above code, the variable names will appear in the graphical report designer as "customer.fname" and "customer.lname".
Related concepts
The SaxDocumentHandler class on page 2617
The om.SaxDocumentHandler class provides an interface to write an XML filter with events.
Expressions on page 330
Shows the possible expressions supported in the language.

NEED
Specifies the number of rows needed in a report section.

Syntax

```
NEED num-lines LINE[S]
```

1. num-lines is the number of lines.

Usage
This statement has the effect of a conditional SKIP TO TOP OF PAGE statement, the condition being that the number to which the integer expression evaluates is greater than the number of lines that remain on the current page.

The NEED statement can prevent the report from dividing parts of the output that you want to keep together on a single page. In this example, the NEED statement causes the PRINT statement to send output to the next page unless at least six lines remain on the current page:

```
AFTER GROUP OF r.order_num
  NEED 6 LINES
  PRINT "", r.order_date, "", GROUP SUM(r.total_price)
```

The LINES value specifies how many lines must remain between the line above the current character position and the bottom margin for the next PRINT statement to produce output on the current page. If fewer than LINES remain on the page, the report engine prints both the PAGE TRAILER and the PAGE HEADER.

The NEED statement does not include the BOTTOM MARGIN value when it compares LINES to the number of lines remaining on the current page. NEED is not valid in FIRST PAGE HEADER, PAGE HEADER, or PAGE TRAILER blocks.

Related concepts
SKIP on page 2057
Skips a given number of lines in a report.

PAUSE
Pauses a report displayed to the screen.

Syntax

```
PAUSE ["comment"]
```

1. comment is an optional comment to be displayed.

Usage
Output is sent by default to the screen unless the START REPORT statement or the OUTPUT section specifies a destination for report output.

The PAUSE statement can be executed only if the report sends its output to the screen. It has no effect if you include a TO clause in either of these contexts:

- In the OUTPUT section of the report definition.
In the START REPORT statement of the report driver.

Include the PAUSE statement in the PAGE HEADER or PAGE TRAILER block of the report. For example, the following code causes the runtime system to skip a line and pause at the end of each page of report output displayed on the screen:

```
PAGE TRAILER
  SKIP 1 LINE
  PAUSE "Press return to continue"
```

**Related concepts**

**SKIP**

Skips a given number of lines in a report.

**Syntax**

```
SKIP { num-lines LINE[S] \ TO TOP OF PAGE }
```

1. `num-lines` is the number of lines.

**Usage**

The `SKIP` statement allows you to insert blank lines into report output or to skip to the top of the next page as if you had included an equivalent number of `PRINT` statements without specifying any expression list.

Output from any PAGE HEADER or PAGE TRAILER control block appears in its usual location.

The `SKIP` statement cannot appear within a CASE statement, a FOR loop, or a WHILE loop.

Example

```
FIRST PAGE HEADER
  PRINT "Customer List"
  SKIP 2 LINES
  PRINT "Number        Name               Location"
  SKIP 1 LINE
  PAGE HEADER
    PRINT "Number        Name               Location"
    SKIP 1 LINE
    ON EVERY ROW
      PRINT r.customer_num, r.fname, r.city
```

**Related concepts**

`PRINT` on page 2052
Formats and prints a row of data in a report routine.

**Report operators**

Report operators can be used to print dynamic report information.

**LINENO**

Contains the current line number in a report.

**Syntax**

```
LINENO
```

**Usage**

This operator takes no operand but returns the value of the line number of the report line that is currently printing. The report engine calculates the line number by calculating the number of lines from the top of the current page, including the TOP MARGIN.

**Example**

In this example, a PRINT statement instructs the report to calculate and display the current line number, beginning in the tenth character position after the left margin:

```
ON EVERY ROW
  IF LINENO > 9 THEN
    PRINT COLUMN 10, "Line:", LINENO USING "<<<"
  END IF
```

**Related concepts**

START REPORT on page 2034

The START REPORT instruction initializes a report execution.

**PAGENO**

Contains the current page number in a report.

**Syntax**

```
PAGENO
```

**Usage**

This operator takes no operand but returns the number of the page the report engine is currently printing. You can use PAGENO in the PAGE HEADER or PAGE TRAILER block, or in other control blocks to number the pages of a report sequentially.

**Example**

If you use the SQL aggregate COUNT(*) in the SELECT statement to find how many records are returned by the query, and if the number of records that appear on each page of output is both fixed and known, you can calculate the total number of pages, as in this example:

```
FIRST PAGE HEADER
  SELECT COUNT(*) INTO cnt FROM customer
  LET y = cnt/50 -- Assumes 50 records per page
```
If the calculated number of pages was 20, the first page trailer would be:

Page 1 of 20

PAGENO increments with each page, so the last page trailer would be:

Page 20 of 20

**Related concepts**

**START REPORT** on page 2034
The START REPORT instruction initializes a report execution.

**SPACES**
Generates the given number of blank characters.

**Syntax**

```
num-spaces SPACES
```

1. `num-spaces` is the number of spaces.

**Usage**
This operator returns a string of blanks, equivalent to a quoted string containing the specified number of blank spaces.

In a PRINT statement, these blanks are inserted at the current character position.

Its operand must be an integer expression that returns a positive number, specifying an offset (from the current character position) no greater than the difference (right margin - current position). After PRINT SPACES has executed, the new current character position has moved to the right by the specified number of characters.

Outside PRINT statements, SPACES and its operand must appear within parentheses: `(n SPACES)`.

**Example**

```
ON EVERY ROW
   LET s = (6 SPACES), "=ZIP"
   PRINT r.fname, 2 SPACES, r.lname, s
```

**Related concepts**

**Integer expressions** on page 333
This section covers integer expression evaluation rules.

**WORDWRAP**
Splits a character string to match a given margin limit.

**Syntax**

```
WORDWRAP [ RIGHT MARGIN position ]
```

1. `position` defines the temporary right margin, as a number of characters, counting from the left.
Usage

The **WORDWRAP** operator automatically wraps successive segments of long character strings onto successive lines of report output. Any string value that is too long to fit between the current position and the right margin is divided into segments and displayed between temporary margins:

- The current character position becomes the temporary left margin.
- Unless you specify **RIGHT MARGIN**, the right margin defaults to 132, or to the size value from the **RIGHT MARGIN** clause of the **OUTPUT** section or **START REPORT** instruction.

Specify **WORDWRAP RIGHT MARGIN integer** to set a temporary right margin as a number of characters, counting from the left edge of the page. This value cannot be smaller than the current character position or greater than right margin defined for the report. The current character position becomes the temporary left margin. These temporary values override the specified or default left and right margins of the report.

After the **PRINT** statement has executed, any explicit or default margins defined in the **RIGHT MARGIN** clause of the **OUTPUT** section or **START REPORT** instruction are restored.

The following **PRINT** statement specifies a temporary left margin in column 10 and a temporary right margin in column 70 to display the character string that is stored in the variable called **mynovel**:

```plaintext
PRINT COLUMN 10, mynovel WORDWRAP RIGHT MARGIN 70
```

The data string can include printable ASCII characters. It can also include the **TAB** (ASCII 9), **LINEFEED** (ASCII 10), and **ENTER** (ASCII 13) characters to partition the string into words that consist of substrings of other printable characters. Other non-printable characters might cause runtime errors. If the data string cannot fit between the margins of the current line, the report engine breaks the line at a word division, and pads the line with blanks at the right.

From left to right, the report engine expands any **TAB** character to enough blank spaces to reach the next tab stop. By default, tab stops are in every eighth column, beginning at the left-hand edge of the page. If the next tab stop or a string of blank characters extends beyond the right margin, the report engine takes these actions:

1. Prints blank characters only to the right margin.
2. Discards any remaining blanks from the blank string or tab.
3. Starts a new line at the temporary left margin.
4. Processes the next word.

The report engine starts a new line when a word plus the next blank space cannot fit on the current line. If all words are separated by a single space, this action creates an even left margin. The following rules are applied (in descending order of precedence) to the portion of the data string within the right margin:

- Break at any **LINEFEED**, or **ENTER**, or **LINEFEED, ENTER** pair.
- Break at the last blank (ASCII 32) or **TAB** character before the right margin.
- Break at the right margin, if no character farther to the left is a space, **ENTER**, **TAB**, or **LINEFEED** character.

The report engine maintains page discipline under the **WORDWRAP** option. If the string is too long for the current page, the report engine executes the statements in any page trailer and header control blocks before continuing output onto a new page.

For Japanese locales, a suitable break can also be made between the Japanese characters. However, certain characters must not begin a new line, and some characters must not end a line. This convention creates the need for **KINSOKU** processing, whose purpose is to format the line properly, without any prohibited word at the beginning or ending of a line.

Reports use the wrap-down method for **WORDWRAP** and **KINSOKU** processing. The wrap-down method forces down to the next line characters that are prohibited from ending a line. A character that precedes another that is prohibited from beginning a line can also wrap down to the next line. Characters that are prohibited from beginning or ending a line must be listed in the locale. The runtime system tests for prohibited characters at the beginning and ending of a line, testing the first and last visible characters. The **KINSOKU** processing only happens once for each
line. That is, no further KINSOKU processing occurs, even if prohibited characters are still on the same line after the first KINSOKU processing.

**Report aggregate functions**

Report aggregate functions can be used to compute data.

**COUNT**
Counts a number of rows in a report based on a condition.

**Syntax**

```
[GROUP] COUNT(*) [WHERE condition ]
```

1. *condition* is a boolean expression evaluated to compute the aggregate value.

**Usage**

This aggregate report instruction returns the total number of records qualified by the optional *WHERE* condition. The *WHERE* condition is evaluated after any *OUTPUT TO REPORT* execution. Even if it is typically used in *AFTER GROUP OF* blocks, the aggregate expression is not evaluated in that block. Changing values of the *WHERE* clause in the *AFTER GROUP* context will not have an immediate effect.

Using the *GROUP* keyword causes the aggregate instructions to include only data of the current group of records that have the same value for the variable that you specify in the *AFTER GROUP OF* control block.

**Example**

The following fragment of a report definition uses the *AFTER GROUP OF* control block and *GROUP* keyword to form sets of records depending on how many items are in each order. The last *PRINT* statement calculates the total price of each order, adds a shipping charge, and prints the result. Because no *WHERE* clause is specified here, *GROUP SUM()* combines the *total_price* of every item in the group included in the order.

```
AFTER GROUP OF number
  SKIP 1 LINE
  PRINT 4 SPACES, "Shipping charges for the order: ",
  ship_charge USING "$$$.&&"
  PRINT 4 SPACES, "Count of small orders: ",
  GROUP COUNT(*) WHERE total_price < 200.00 USING "##,###"
  SKIP 1 LINE
  PRINT 5 SPACES, "Total amount for the order: ",
  ship_charge + GROUP SUM(total_price) USING "$$$,$$$,$$$.$&"
```

**Related concepts**

- [Report engine configuration](#) on page 2064
- Report engine behavior can be controlled with FGLPROFILE settings.
- [Two-pass reports](#) on page 2051
  The report engine supports two-pass reports, to order rows automatically.

**PERCENT**
Calculates the percentage of rows matching a condition.

**Syntax**

```
[GROUP] PERCENT(*) [WHERE condition ]
```
1. *condition* is a boolean expression evaluated to compute the aggregate value.

### Usage

This aggregate report instruction returns the percentage of the total number of records qualified by the optional *WHERE* condition.

Using the *GROUP* keyword causes the aggregate instructions to include only data of the current group of records that have the same value for the variable that you specify in the *AFTER GROUP OF* control block.

This aggregate instruction makes a two-pass report when not using the *GROUP* keyword and is used in any control block other than *ON LAST ROW*, or when using the *GROUP PERCENT(*)* anywhere in the report.

### Related concepts

- Report engine configuration on page 2064
- Report engine behavior can be controlled with FGLPROFILE settings.
- Two-pass reports on page 2051
- The report engine supports *two-pass reports*, to order rows automatically.

### SUM

Calculates the total of a report parameter based on a condition.

#### Syntax

```plaintext
[GROUP] SUM(expression) [WHERE condition]
```

1. *expression* is the expression to be computed.
2. *condition* is a boolean expression evaluated to compute the aggregate value.

#### Usage

This aggregate report instruction evaluates as the total of expression among all records or among records qualified by the optional *WHERE* clause and any *GROUP* specification.

Using the *GROUP* keyword causes the aggregate instructions to include only data of the current group of records that have the same value for the variable that you specify in the *AFTER GROUP OF* control block.

Input records for which the *expression* evaluates to *NULL* values are ignored.

By default, if all input record values are *NULL*, the result of the aggregate is *NULL*. However, you can control this behavior and force the runtime system to return zero instead of *NULL* with the `report.aggregateZero` FGLPROFILE parameter.

This aggregate instruction makes a two-pass report when not using the *GROUP* keyword and is used in any control block other than *ON LAST ROW*.

### Related concepts

- Report engine configuration on page 2064
- Report engine behavior can be controlled with FGLPROFILE settings.
- Two-pass reports on page 2051
- The report engine supports *two-pass reports*, to order rows automatically.

### AVG

Calculates the average of a report parameter based on a condition.

#### Syntax

```plaintext
[GROUP] AVG(expression) [WHERE condition]
```

1. *expression* is the expression to be computed.
2. *condition* is a boolean expression evaluated to compute the aggregate value.

#### Usage

This aggregate report instruction evaluates as the total of expression among all records or among records qualified by the optional *WHERE* clause and any *GROUP* specification.

Using the *GROUP* keyword causes the aggregate instructions to include only data of the current group of records that have the same value for the variable that you specify in the *AFTER GROUP OF* control block.

Input records for which the *expression* evaluates to *NULL* values are ignored.

By default, if all input record values are *NULL*, the result of the aggregate is *NULL*. However, you can control this behavior and force the runtime system to return zero instead of *NULL* with the `report.aggregateZero` FGLPROFILE parameter.

This aggregate instruction makes a two-pass report when not using the *GROUP* keyword and is used in any control block other than *ON LAST ROW*.
1. *expression* is the expression to be computed.
2. *condition* is a boolean expression evaluated to compute the aggregate value.

**Usage**

This aggregate report instruction evaluates as the average (that is, the arithmetic mean value) of expression among all records or among records qualified by the optional *WHERE* clause and any *GROUP* specification.

Using the *GROUP* keyword causes the aggregate instructions to include only data of the current group of records that have the same value for the variable that you specify in the *AFTER GROUP OF* control block.

Input records for which the *expression* evaluates to NULL values are ignored.

By default, if all input record values are NULL, the result of the aggregate is NULL. However, you can control this behavior and force the runtime system to return zero instead of NULL with the `report.aggregateZero` FGLPROFILE parameter.

This aggregate instruction makes a two-pass report when not using the *GROUP* keyword and when used in any control block other than *ON LAST ROW*.

**Related concepts**

- Report engine configuration on page 2064
- Report engine behavior can be controlled with FGLPROFILE settings.
- Two-pass reports on page 2051
- The report engine supports *two-pass reports*, to order rows automatically.

**MIN**

Calculates the minimum value of a report parameter based on a condition.

**Syntax**

```plaintext
MIN(expression) [WHERE condition]
```

1. *expression* is the expression to be computed.
2. *condition* is a boolean expression evaluated to compute the aggregate value.

**Usage**

For number, currency, and interval values, `MIN(expression)` aggregate report instruction returns the minimum value for *expression* among all records or among records qualified by the *WHERE* clause and any *GROUP* specification.

For *DATETIME* or *DATE* data values, greater than means later and less than means earlier in time. Character strings are sorted depending on their first character. If your program is executed in the default (U.S. English) locale, for character data types, greater than means after in the ASCII collating sequence, where a > A > 1, and less than means before in the ASCII sequence, where 1 < A < a.

Using the *GROUP* keyword causes the aggregate instructions to include only data of the current group of records that have the same value for the variable that you specify in the *AFTER GROUP OF* control block.

Input records for which the *expression* evaluates to NULL values are ignored.

By default, if all input record values are NULL, the result of the aggregate is NULL. However, you can control this behavior and force the runtime system to return zero instead of NULL with the `report.aggregateZero` FGLPROFILE parameter.

This aggregate instruction makes a two-pass report when not using the *GROUP* keyword and is used in any control block other than *ON LAST ROW*.

**Related concepts**

- Report engine configuration on page 2064
Report engine behavior can be controlled with FGLPROFILE settings.

Two-pass reports on page 2051
The report engine supports two-pass reports, to order rows automatically.

**MAX**
Calculates the maximum value of a report parameter based on a condition.

**Syntax**

\[
\text{GROUP}\  \text{MAX(expression)} \ [\text{WHERE} \ condition]
\]

1. `expression` is the expression to be computed.
2. `condition` is a boolean expression evaluated to compute the aggregate value.

**Usage**
For number, currency, and interval values, the `MAX(expression)` aggregate report instruction returns the maximum value for `expression` among all records or among records qualified by the `WHERE` clause and any `GROUP` specification.

For `DATETIME` or `DATE` data values, greater than means later and less than means earlier in time. Character strings are sorted depending on their first character. If your program is executed in the default (U.S. English) locale, for character data types, greater than means after in the ASCII collating sequence, where a > A > 1, and less than means before in the ASCII sequence, where 1 < A < a.

Using the `GROUP` keyword causes the aggregate instructions to include only data of the current group of records that have the same value for the variable that you specify in the `AFTER GROUP OF` control block.

Input records for which the `expression` evaluates to `NULL` values are ignored.

By default, if all input record values are `NULL`, the result of the aggregate is `NULL`. However, you can control this behavior and force the runtime system to return zero instead of `NULL` with the `report.aggregateZero` FGLPROFILE parameter.

This aggregate instruction makes a two-pass report when not using the `GROUP` keyword and is used in any control block other than `ON LAST ROW`.

**Related concepts**
Report engine configuration on page 2064
Report engine behavior can be controlled with FGLPROFILE settings.

Two-pass reports on page 2051
The report engine supports two-pass reports, to order rows automatically.

**Report engine configuration**
Report engine behavior can be controlled with FGLPROFILE settings.

By default, aggregate instructions such as `SUM()` return a `NULL` value if all input record values are `NULL`.

You can force the report engine to return a zero decimal value with the following FGLPROFILE setting:

\[
\text{Report.aggregateZero} = \{\text{true} | \text{false}\}
\]

When this entry is set to true, the `SUM()`, `AVG()`, `MIN()`, or `MAX()` aggregate functions return zero when all values are `NULL`.

Default value of the configuration parameter is false (this means aggregate functions evaluate to null if all items are null).
When using GROUP aggregates with this entry is set to true, the aggregate instruction will still return NULL in the first AFTER GROUP OF output of the report. Zero values will be returned starting from second group output. This behavior is expected, for backward compatibility with older versions.

It is not recommended to use the Report.aggregateZero entry if you don't need that specific behavior.

**Related concepts**
The FGLPROFILE file(s) on page 255
FGLPROFILE environment variable defines Genero BDL configuration files
NULL on page 555
The NULL constant is provided as the "nil" value.

## Programming tools

These topics cover programming with the Genero Business Development Language.

### Command reference

Command line tools provided for BDL.

**fglr**

The fglrun tool is the runtime system program that executes p-code programs.

**Syntax 1: Executing programs**

```
fglr [exec-options] program [argument [....]]
```

1. *exec-options* can be execution options as well as trace options.
2. *program* is a .42r program, or a .42m p-code module containing the MAIN definition.
3. *argument* is an argument passed to the program.

**Syntax 2: Starting the debugger**

```
fglr -d [exec-options] program
```

1. The -d option starts fglrun in the debug mode. For more details, see Integrated debugger on page 2142.
2. *exec-options* can be execution options (trace options are not supported).
3. *program* is a .42r program, or a .42m p-code module containing the MAIN definition.

**Syntax 3: Linking programs or libraries (used by fgl**

```
fglr -l -o outfile .42r | .42x | module .42m | library .42x
   | pattern
   | argfile
```

1. *link-options* are described in Table 412: Linker options on page 2067.
2. *outfile* is the .42r program (or a .42x library) to produce from the link.
3. *module .42m* is a p-code module compiled with fglcomp.
4. *library .42x* is a name of a library to be used for linking.
5. *pattern* is a MATCHES-style pattern to find files, like `'\[a-z]\*.42m'`.
6. *argfile* defines a file that contains a list of `.42m` or `.42x` files. Each line must specify a filename or a *pattern*.
7. In this form, *fglrun* links a program as described in *Compiling source files* on page 2106.

**Syntax 4: Using diagnostic options**

```
fglrun diagnostic-option
  \(\downarrow\text{\textbackslash\textbackslash} module.42m\)
  \(\downarrow\text{\textbackslash\textbackslash} pattern\)
  \(\downarrow\text{\textbackslash\textbackslash} [\ldots]\)
  \(\downarrow\text{\textbackslash\textbackslash} \@argfile\)

1. *diagnostic-option* is one of the diagnostic options.
2. *module.42m* is a p-code module compiled with *fglcomp*.
3. *pattern* is a MATCHES-style pattern to find files, like `'\[a-z]\*.42m'`.
4. *argfile* defines a file that contains a list of `.42m` or `.42x` files. Each line must specify a filename or a *pattern*.

**Syntax 5: Information options**

```
fglrun info-option
```

1. *info-option* can be any of the informational options.

**Options**

**Table 409: Execution options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-e extfile (\ldots)</td>
<td>Specify a C extension module to be loaded. This option can take a comma-separated list of extensions.</td>
</tr>
<tr>
<td>--java-option=option</td>
<td>Passes Java runtime options when initializing the JNI interface. See Java Interface for more details.</td>
</tr>
</tbody>
</table>

**Table 410: Trace options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-p</td>
<td>Generate profiling information to stderr (UNIX™ only). See Program profiler on page 2165.</td>
</tr>
<tr>
<td>--start-guilog=logfile</td>
<td>Log all GUI protocol exchange in a file. The GUI log file can then be replayed with the --run-guilog option. If the log file contains a %p placeholder, it is replaced by the current process id.</td>
</tr>
<tr>
<td>--run-guilog=logfile</td>
<td>Replays a GUI log created with the --start-guilog option.</td>
</tr>
<tr>
<td>--gui-listen=port</td>
<td>Instructs the runtime system to listen to a TCP port for incoming GUI connections. For more details see Connecting with a front-end on page 1116.</td>
</tr>
<tr>
<td>--trace</td>
<td>Starts the program by printing function call stack trace with parameter and return values. For more details, see Execution trace on page 2171.</td>
</tr>
</tbody>
</table>
### Table 411: Diagnostic options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-b</td>
<td>Displays compiler version information of the module, see Compiling source files on page 2106.</td>
</tr>
<tr>
<td>--print-imports</td>
<td>Loads the specified modules and prints all IMPORT FGL instructions required in each module. See Identifying modules to be imported on page 483.</td>
</tr>
<tr>
<td>--print-missing-imports</td>
<td>Loads the specified modules and prints all missing IMPORT FGL instructions for each module. See Identifying modules to be imported on page 483.</td>
</tr>
<tr>
<td>--module-size</td>
<td>Shows the amount of limited pcode size for a module.</td>
</tr>
<tr>
<td>--program-size</td>
<td>Shows the amount of limited pcode size for an entire program.</td>
</tr>
<tr>
<td>--merge-cov name.4gl [name.42m.cov]</td>
<td>Merges FGLCOV coverage data files with source files to produce a readable file name.4gl.cov module. If the .42m.cov file is not located beside the source file, you can specify the full path with the second optional file path. See Source code coverage on page 2168.</td>
</tr>
</tbody>
</table>

### Table 412: Linker options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-e extfile[,....]</td>
<td>Specify a C extension module to be loaded. This option can take a comma-separated list of extensions.</td>
</tr>
<tr>
<td>-o progname.42r</td>
<td>libname.42x]</td>
</tr>
<tr>
<td>--java-option=option</td>
<td>Passes Java runtime options when initializing the JNI interface.</td>
</tr>
<tr>
<td></td>
<td>See Java Interface for more details.</td>
</tr>
</tbody>
</table>

### Table 413: Informational options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-V or --version</td>
<td>Displays version information.</td>
</tr>
<tr>
<td>-h or --help</td>
<td>Displays options for the tool.</td>
</tr>
<tr>
<td>-i mbcs</td>
<td>Displays information about the current locale / character set settings. See Application locale on page 512.</td>
</tr>
</tbody>
</table>
Usage

The `fglrun` command line tool executes p-code programs, for example:

```bash
fglrun myprogram.42r -x 123
```

The program file must contain the `MAIN` routine.

The arguments passed to the program can be queried with the `arg_val()` built-in function.

The `.42r` or `.42m` extension is optional:

```bash
fglrun myprogram -x 123
```

**Note:** First `fglrun` tries to find the program file with the name provided in the command line. If the file is not found, the extension is removed (if it is present in the provided file name), and a new search is done by adding the `.42r` extension. If the file is still not found, `fglrun` tries with the `.42m` extension. This means that a program file `myprogram.42m` will be found and loaded, even if you pass `myprogram.42r` to `fglrun`. Specify program files without a `.42r` or `.42m` extension, to avoid mistakes and simplify migration from `.42r` linked programs to `.42m-only` modules (using `IMPORT FGL`).

Related concepts

- **Executing programs** on page 488
- There are different ways to execute compiled programs, depending on the configuration and the development or production context.

fglform

The `fglform` tool compiles form specification files into XML formatted files used by programs.

**Syntax 1: Compiling forms**

```bash
fglform [comp-options] [prepro-options] form-name[,per]
```

1. *comp-options* are described in compilation options.
2. *prepro-options* are described in preprocessor options.
3. *form-name*.per is the form specification file. The .per extension is optional.

**Syntax 2: Extracting localized strings**

```bash
fglform -m [prepro-options] form-name[,per]
```

1. *prepro-options* are described in preprocessor options.
2. *form-name*.per is the form specification file. The .per extension is optional
3. In this form, `fglform` extracts localized strings from the source.

**Syntax 3: Information options**

```bash
fglform info-option
```

1. *info-option* can be any of the informational options.
Options:

Table 414: Compilation options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-M</td>
<td>Write error messages to standard output instead of creating a .err error file.</td>
</tr>
<tr>
<td>-W { all }</td>
<td>Produce warning messages. Only -W all option is supported for now.</td>
</tr>
</tbody>
</table>

Table 415: Preprocessor options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-E</td>
<td>Preprocess only. See Source preprocessor on page 2132 for more details.</td>
</tr>
<tr>
<td>-p option</td>
<td>Preprocessing control, where option can be one of:</td>
</tr>
<tr>
<td></td>
<td>• nopp: Disable preprocessing.</td>
</tr>
<tr>
<td></td>
<td>• noli: No line number information (only with -E option).</td>
</tr>
<tr>
<td></td>
<td>• fglpp: Use # syntax instead of &amp; syntax.</td>
</tr>
<tr>
<td></td>
<td>• auto: Detect # syntax or &amp; syntax automatically.</td>
</tr>
<tr>
<td>-I path</td>
<td>Provides a single path to search for include files. See Source preprocessor on page 2132 for more details.</td>
</tr>
<tr>
<td>-D ident[=value]</td>
<td>Defines the macro 'ident' with an optional value (default is 1). See Source preprocessor on page 2132 for more details.</td>
</tr>
<tr>
<td>-U ident</td>
<td>Undefines the macro 'ident'. See Source preprocessor on page 2132 for more details.</td>
</tr>
</tbody>
</table>

Table 416: Informational options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-V or --version</td>
<td>Displays version information.</td>
</tr>
<tr>
<td>-h or --help</td>
<td>Displays options for the tool.</td>
</tr>
<tr>
<td>-i \ mbcs \</td>
<td>Displays information about the current locale / character set settings. See Application locale on page 512.</td>
</tr>
</tbody>
</table>

Usage

The fglform command line tool compiles a .per form specification file into a .42f compiled version:

fglform custform.per

The .per extension is optional, if not used, fglform will automatically search for files with this extension.

The .42f compiled version is an XML formatted file used by programs when a form definition is loaded with the OPEN FORM or OPEN WINDOW WITH FORM instructions.

Related concepts

Compiling form specification files (.per) on page 2106
The .per form definition files must be compiled to .42f XML files, in order to be loaded by the runtime system.

Form specification files on page 1237
Form specification files are the source files defining the layout and content of application forms.

fglcomp on page 2071
The fglcomp tool compiles .4gl source files into .42m p-code modules, and does various other tasks.

fgl2p
The fgl2p tool compiles source files and assembles p-code modules into a .42r program or a .42x library.

Syntax
To create a library:

```bash
fgl2p [options] -o outfile.42x [ pcmod.42m | source.4gl ] [...]
```

To create a program:

```bash
fgl2p [options] -o outfile.42r [ pcmod.42m | source.4gl | library.42x ] [...]
```

1. `options` are described in Table 417: fgl2p options on page 2070.
2. `outfile.42r` is the name of the program to be created.
3. `outfile.42x` is the name of the library to be created.
4. `pcmod.42m` is a p-code module compiled with fglcomp.
5. `source.4gl` is a program source file.
6. `library.42x` is the name of a library to be linked.

Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-V</td>
<td>Displays version information.</td>
</tr>
<tr>
<td>-h</td>
<td>Displays options for the tool.</td>
</tr>
<tr>
<td>-o outfile.ext</td>
<td>Output file specification, where ext can be 42r for a program or 42x for a library.</td>
</tr>
<tr>
<td>otheroption</td>
<td>Other options are passed to the linker or compiler.</td>
</tr>
</tbody>
</table>

Usage

The fgl2p command line tool can compile .4gl source files and link .42m p-code modules together, to create a .42x library or a .42r program file.

```bash
fgl2p -o myprog.42r module1.4gl module2.42m lib1.42x
```

This tool is provided for convenience, in order to create programs or libraries in one command line. It uses the fglcomp and the fgllink tools to compile and link modules together.

Related concepts

Compiling source files on page 2106
Describes how to build the runtime files from source files.

fglcomp
The fglcomp tool compiles .4gl source files into .42m p-code modules, and does various other tasks.

Syntax 1: Compiling .4gl sources

```
fglcomp [comp-options] [prepro-options] file-list
```

1. *comp-options* are described in compilation options.
2. *prepro-options* are described in preprocessor options.
3. In this form, fglcomp can handle several .4gl sources files as arguments, see *file-list*.

where *file-list* is:

```
<module[.4gl]|pattern [...] @argfile>
```

1. *module*.4gl is a source file to compile. The .4gl extension is optional.
2. *pattern* is a MATCHES-style pattern to find files, like `'[a-z]*.4gl'`.
3. *argfile* defines a file that contains a list of .4gl sources to be compiled. Each line must specify a filename or a pattern.

Syntax 2: Building documentation

```
fglcomp --build-doc [doc-options] [prepro-options] module[.4gl]
```

1. *doc-options* are described in documentation options.
2. *prepro-options* are described in preprocessor options.
3. *module*.4gl is a source file to compile. The .4gl extension is optional.

Syntax 3: Extracting localized strings

```
fglcomp -m [prepro-options] module[.4gl]
```

1. *module*.4gl is a source file to compile. The .4gl extension is optional.
2. *prepro-options* are described in preprocessor options.
3. In this form, fglcomp extracts localized strings from the source.

Syntax 4: Source code formatting

```
fglcomp --format [format-options] module[.4gl]
```

1. *format-options* can be any of the formatting options.
2. *module*.4gl is a source file to compile. The .4gl extension is optional.

Syntax 5: Add imported module prefixes

```
fglcomp --qualify-imports [prepro-options] module[.4gl]
```

1. *prepro-options* are described in preprocessor options.
2. *module*.4gl is a source file to compile. The .4gl extension is optional.

**Syntax 6: Show dependencies**

```
fglcomp --dependencies [prepro-options] file-list
```

1. *prepro-options* are described in preprocessor options.
2. In this form, fglcomp can handle several .4gl sources files as arguments, see file-list.

**Syntax 7: Information options**

```
fglcomp info-option
```

1. *info-option* can be any of the informational options.

**Options**

**Table 418: Compilation options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--M</td>
<td>Write error messages to standard output instead of creating a .err error file.</td>
</tr>
<tr>
<td>--make</td>
<td>Do not recompile .42m modules up-to-date with provided .4gl source. See <strong>Compiling in make mode</strong> on page 2109.</td>
</tr>
<tr>
<td>--simulate</td>
<td>Print the list of .4gl sources that would be compiled (can be used with or without --make). See <strong>Compiling in make mode</strong> on page 2109.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>-W</code> warning-argument</td>
<td>Produce warning messages.</td>
</tr>
<tr>
<td></td>
<td>The warning argument can be used as follows:</td>
</tr>
<tr>
<td></td>
<td>• <code>-W all</code> enables all warning flags.</td>
</tr>
<tr>
<td></td>
<td>• <code>-W error</code> makes the compiler stop, if any warning is raised, as</td>
</tr>
<tr>
<td></td>
<td>if an error occurred.</td>
</tr>
<tr>
<td></td>
<td>• <code>-W to-err-file</code> writes warnings to the <code>.err</code> file when this file is</td>
</tr>
<tr>
<td></td>
<td>produced. By default warnings go to the stderr stream.</td>
</tr>
<tr>
<td></td>
<td>• <code>-W unused</code> displays a message for all unused symbols.</td>
</tr>
<tr>
<td></td>
<td>• <code>-W unused-parameter</code> displays a message for all unused function</td>
</tr>
<tr>
<td></td>
<td>parameters.</td>
</tr>
<tr>
<td></td>
<td>• <code>-W return</code> displays a warning if the same function returns</td>
</tr>
<tr>
<td></td>
<td>different number of values with several <code>RETURN</code> statements.</td>
</tr>
<tr>
<td></td>
<td>• <code>-W stdsql</code> displays a message for all non-portable SQL</td>
</tr>
<tr>
<td></td>
<td>statements or language instructions.</td>
</tr>
<tr>
<td></td>
<td>• <code>-W print</code> displays a message when the <code>PRINT</code> instruction is used</td>
</tr>
<tr>
<td></td>
<td>outside a <code>REPORT</code>.</td>
</tr>
<tr>
<td></td>
<td>• <code>-W implicit</code> warns on references to undeclared functions. A</td>
</tr>
<tr>
<td></td>
<td>function is undeclared if not defined in the current module or in any</td>
</tr>
<tr>
<td></td>
<td>imported module.</td>
</tr>
<tr>
<td></td>
<td>• <code>-W apidoc</code> prints a warning for invalid source documentation</td>
</tr>
<tr>
<td></td>
<td>tags when using the <code>--build-doc</code> option.</td>
</tr>
<tr>
<td><code>-W</code> option</td>
<td>The <code>-W</code> option also supports the negative form of arguments by</td>
</tr>
<tr>
<td></td>
<td>using the <code>no-</code> prefix as in: <code>no-return</code>, <code>no-unused</code>, <code>no-stdsql</code>. You</td>
</tr>
<tr>
<td></td>
<td>might need to use these negative forms in order to</td>
</tr>
<tr>
<td></td>
<td>disable some warning when using the <code>-W all</code> option:</td>
</tr>
<tr>
<td></td>
<td><code>fglcomp -Wall -Wno-stdsql customers.4gl</code></td>
</tr>
<tr>
<td></td>
<td>Switches will be enabled/disabled in the order of appearance in the</td>
</tr>
<tr>
<td><code>-timestamp</code></td>
<td>Add compilation timestamp to build information in 42m header. See</td>
</tr>
<tr>
<td></td>
<td>42m module information on page 2120.</td>
</tr>
<tr>
<td><code>-omit-source-name</code></td>
<td>Omit the source file name in the build information of the 42m</td>
</tr>
<tr>
<td></td>
<td>header. See 42m module information on page 2120.</td>
</tr>
<tr>
<td><code>-tag=string</code></td>
<td>Write a custom string in the build information of the 42m header. See</td>
</tr>
<tr>
<td></td>
<td>42m module information on page 2120.</td>
</tr>
<tr>
<td><code>-build-rdd</code></td>
<td>While compiling, generate the <code>module.rdd</code> Report Data Definition file (of</td>
</tr>
<tr>
<td></td>
<td><code>REPORT</code> routines).</td>
</tr>
<tr>
<td><code>-verbose</code></td>
<td>Print detailed compilation information.</td>
</tr>
<tr>
<td><code>-implicit=type</code></td>
<td>Specify whether or not to compile imported modules, if the .42m does not</td>
</tr>
<tr>
<td></td>
<td>exist, or if the .4gl source is more recent as the .42m.</td>
</tr>
<tr>
<td></td>
<td><code>type</code> can be one of:</td>
</tr>
<tr>
<td></td>
<td>• <code>none</code>: Disable auto-compilation of imported modules.</td>
</tr>
<tr>
<td></td>
<td>• <code>42m</code> (default): Compile imported modules if needed.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-r or --resolve-calls</td>
<td>Throw an error on references to undeclared functions. Each external function must be made known to the compiler by IMPORT FGL. When using this option, the linking phase is no longer needed; a source (.4gl) file compiled with this option must not be linked. See IMPORT FGL module on page 481 for more details.</td>
</tr>
<tr>
<td>--java-option=option</td>
<td>Passes Java runtime options when initializing the JNI interface. See Java Interface for more details.</td>
</tr>
<tr>
<td>-S</td>
<td>With this option, fglcomp extracts all Static SQL commands from the source while compiling the source code.</td>
</tr>
</tbody>
</table>

Table 419: Documentation options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--doc-private</td>
<td>When using the --build-doc option, include PRIVATE symbols to the documentation.</td>
</tr>
</tbody>
</table>

Table 420: Formatting options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--fo-align-consecutive-assignments=0</td>
<td>1</td>
</tr>
<tr>
<td>--fo-align-consecutive-types=0</td>
<td>1</td>
</tr>
<tr>
<td>--fo-align-trailing-comments=0</td>
<td>1</td>
</tr>
<tr>
<td>--fo-inplace</td>
<td>Write formatted output back to the provided file, instead of stdout. Creates a copy of the original file in filename.4gl~</td>
</tr>
<tr>
<td>--fo-fallback-style=filename</td>
<td>Specify the configuration filename to be used if no .fgl-format file is found.</td>
</tr>
<tr>
<td>--fo-continuation-indent-width=integer</td>
<td>Define the source line width limit. Default is 80.</td>
</tr>
<tr>
<td>--fo-column-limit=integer</td>
<td>Number of columns to use for indentation. Default is 4.</td>
</tr>
<tr>
<td>--fo-indent-width=integer</td>
<td>Indent width for line continuations. Default is 4.</td>
</tr>
<tr>
<td>--fo-label-indent=0</td>
<td>1</td>
</tr>
<tr>
<td>--fo-pack=0</td>
<td>1</td>
</tr>
<tr>
<td>--fo-lowercase-keywords=0</td>
<td>1</td>
</tr>
<tr>
<td>--fo-use-tab=0</td>
<td>1</td>
</tr>
</tbody>
</table>
Option | Description
--- | ---
--fo-tab-width=integer | Defines the number of columns used for tab stop, when --fo-use-tab is specified. The default is 8 columns.
--fo-lines=start-line:end-line | Formats only the range of lines specified. This option can be used multiple times to specify several pieces of code to be reformatted.

Table 421: Preprocessor options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
</table>
| -E | Preprocess only. See Source preprocessor on page 2132 for more details. Preprocessing control, where option can be one of:
  - nopp: Disable preprocessing.
  - noli: No line number information (only with -E option).
  - fglpp: Use # syntax instead of & syntax.
  - auto: Detect # syntax or & syntax automatically. |
| -p option | Provides a single path to search for include files. See Source preprocessor on page 2132 for more details. |
| -I path | |
| -D ident=value | Defines the macro 'ident' with an optional value (default is 1). See Source preprocessor on page 2132 for more details. |
| -U ident | Undefines the macro 'ident'. See Source preprocessor on page 2132 for more details. |

Table 422: Informational options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-V or --version</td>
<td>Displays version information.</td>
</tr>
<tr>
<td>-h or --help</td>
<td>Displays options for the tool.</td>
</tr>
<tr>
<td>-i [mbcs]</td>
<td>Displays information about the current locale / character set settings. See Application locale on page 512.</td>
</tr>
</tbody>
</table>

Usage

The fglcomp command line tool compiles a .4gl into a .42m p-code module:

```
fglcomp customers.4gl
```

If a compilation error occurs, the compiler generates an error file with an .err extension. The error file contains the original source code with error messages. Use the option -M to display the error messages to standard error instead of producing the .err file.

Related concepts

Compiling program code files (.4gl) on page 2108
The .4gl source files must be compiled to .42m p-code files, in order to be loaded by the runtime system.

fglform on page 2068
The fglform tool compiles form specification files into XML formatted files used by programs.

**fgllink**

The fgllink tool assembles p-code modules produced with fglcomp into a .42r program or a .42x library.

**Syntax 1: Create a 42x library**

```bash
fgllink [options] -o outfile.42x
  ↓  ↓
  module.42m ↓ pattern
  ↓ @argfile ↓
```

1. **options** are described in Table 423: fgllink options on page 2076.
2. **outfile.42x** is the name of the library to be created.
3. **module.42m** is a p-code module compiled with fglcomp.
4. **pattern** is a MATCHES-style pattern to find files, like '{[a-z]*.42m}'.
5. **argfile** defines a file that contains a list of .42m files. Each line must specify a filename or a pattern.

**Syntax 2: Create a 42r program**

```bash
fgllink [options] -o outfile.42r
  ↓  ↓
  module.42m ↓ library.42x ↓ [,...]
  ↓ pattern ↓ @argfile ↓
```

1. **options** are described in Table 423: fgllink options on page 2076.
2. **outfile.42r** is the name of the program to be created.
3. **module.42m** is a p-code module compiled with fglcomp.
4. **library.42x** is the name of a library to be linked.
5. **pattern** is a MATCHES-style pattern to find files, like '{[a-z]*.42m}'.
6. **argfile** defines a file that contains a list of .42m or .42x files. Each line must specify a filename or a pattern.

**Options**

**Table 423: fgllink options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-V or --version</td>
<td>Displays version information.</td>
</tr>
<tr>
<td>-h or --help</td>
<td>Displays options for the tool.</td>
</tr>
<tr>
<td>-e extfile[,...]</td>
<td>Specify a C extension module to be loaded. This option can take a</td>
</tr>
<tr>
<td></td>
<td>comma-separated list of extensions.</td>
</tr>
<tr>
<td>-o progname.42r</td>
<td>Output file specification, it can be a 42r program or a 42x library.</td>
</tr>
<tr>
<td>otheroption</td>
<td>Other options are passed to fglrun for linking.</td>
</tr>
</tbody>
</table>

**Usage**

The fgllink command line tool links .42m p-code modules together to create a .42x library or a .42r program file.

```bash
fgllink -o myprog.42x module1.42m module2.42m lib1.42x
```

**Note:** fgllink is a wrapper calling fglrun with the -l option.
Related concepts

Linking libraries on page 2113
Describes how to link .42m modules together to build a .42x library file.

Linking programs on page 2115
Describes how to link .42m modules together to build a .42x program file.

fglmkmsg

The fglmkmsg tool compiles .msg message files into a binary version used by programs.

Syntax

```
fglmkmsg [options] srcfile.msg [outfile.iem]
```

1. `options` are described in Table 424: fglmkmsg options on page 2077.
2. `srcfile.msg` is the source message file.
3. `outfile.iem` is the destination file.

Options

Table 424: fglmkmsg options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-V</td>
<td>Displays version information.</td>
</tr>
<tr>
<td>-h</td>
<td>Displays options for the tool.</td>
</tr>
<tr>
<td>-r msgfile</td>
<td>De-compiles a binary message file.</td>
</tr>
</tbody>
</table>

Usage

The fglmkmsg command line tool compiles a .msg message file into a .iem compiled version:

```
fglmkmsg mess01.msg
```

For backward compatibility, you can specify the output file as second argument:

```
fglmkmsg mess01.msg mess01.iem
```

The .iem compiled version can be used by BDL programs, for example, when the HELP clause is used in a MENU or INPUT instruction.

Related concepts

Compiling message files (.msg) on page 1161
The .msg message files must be compiled to .iem binary files, in order to be loaded by the runtime system.

Message files on page 1160
Message files centralize strings and larger texts identified by a number, that can be used in programs.

fglmkext

The fglmkext tool compiles and links a user C Extension.

Syntax

```
fglmkext [options] source.c [...]
```

1. `options` are described in Table 425: fglmkext options on page 2078.
2. *source.c* is a C source file implementing C extension functions.

**Options**

**Table 425: fglmkext options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-V</td>
<td>Displays version information.</td>
</tr>
<tr>
<td>-h</td>
<td>Displays options for the tool.</td>
</tr>
<tr>
<td>-o libname</td>
<td>Output file specification, defines the C Extension library name.</td>
</tr>
</tbody>
</table>

**Usage**

The `fglmkext` command line tool compiles and links a C Extension library.

The command can be used with a single source file, the name of the library will default to the name of the specified source:

```
fglmkext myext.c
```

If a single C source file is provided, it must define the `usrFunctions` C extension interface structure as well as the functions to be used from a BDL program.

In order to specify a library name, use the -o option, several C source files can also be specified. For example, on a UNIX platform:

```
fglmkext -o mycext.so module_a.c module_b.c
```

**Related concepts**

*Creating C-Extensions* on page 2231

Custom C-Extensions must be provided to the runtime system as Shared Objects (.so) on UNIX™, and as Dynamically Loadable Libraries (.DLL) on Windows®.

**fgldb**

The `fgldb` tool is an interface program for remote debugging.

**Syntax 1: Debugging an application running on the computer**

```
fgldb -p process-id
```

1. `process-id` is the process identifier of the `fglrun` process.

**Syntax 2: Debugging an app running on a mobile device**

```
fgldb -m host[:port]
```

1. `host` is the hostname or IP address of the mobile device where the program executes.
2. `port` is the TCP debug port number to connect to, default is 6400.
Options

Table 426: fgldb options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-V</td>
<td>Displays version information.</td>
</tr>
<tr>
<td>-h</td>
<td>Displays options for the tool.</td>
</tr>
<tr>
<td>-p process-id</td>
<td>Attach to an fglrun process running on the same computer, by using its process id.</td>
</tr>
<tr>
<td>-m host:port</td>
<td>Attach to a running mobile app to debug, using the mobile device hostname/IP and debug port.</td>
</tr>
</tbody>
</table>

Usage

The fgldb command line tool is an interface tool to attach to the FGL integrated debugger remotely.

The fgldb tool can be used to:

- Debug an fglrun process currently running on the same computer, by using its process id. For more details, see Attaching to a running program on page 2143.
- Debug an application on a mobile device, by using the mobile device IP adress and its TCP debug port. For more details, see Debugging on a mobile device on page 2145

Related concepts

Integrated debugger on page 2142
Describes the command-line debugger you can use to find bugs in your programs.

fgldbsch

The fgldbsch tool generates the database schema files from an existing database.

Syntax

fgldbsch -db dbname [options]

1. dbname is the name of the database from which the schema is to be extracted.
2. options are described in Table 427: fgldbsch options on page 2079.

Options

Table 427: fgldbsch options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-V</td>
<td>Displays version information.</td>
</tr>
<tr>
<td>-h</td>
<td>Displays options for the tool.</td>
</tr>
<tr>
<td>-H</td>
<td>Displays long help.</td>
</tr>
<tr>
<td>-v</td>
<td>Enable verbose mode (display information messages).</td>
</tr>
<tr>
<td>-ct</td>
<td>Display data type conversion tables.</td>
</tr>
<tr>
<td>-cx dbtype</td>
<td>Display data type conversion table for the given database type.</td>
</tr>
<tr>
<td>-db dbname</td>
<td>Specify the database as dbname. This option is required to generate the schema files.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-dv dbdriver</td>
<td>Specify the database driver to be used.</td>
</tr>
<tr>
<td>-un user</td>
<td>Define the user name for database connection as user.</td>
</tr>
<tr>
<td>-up pswd</td>
<td>Define the user password for database connection as pswd.</td>
</tr>
<tr>
<td>-ow owner</td>
<td>Define the owner of the database tables as owner.</td>
</tr>
<tr>
<td>-cv string</td>
<td>Specify the data type conversion rules by character positions in string.</td>
</tr>
<tr>
<td>-of name</td>
<td>Specify output files prefix, default is database name.</td>
</tr>
<tr>
<td>-tn tabname</td>
<td>Extract the description of a specific table.</td>
</tr>
<tr>
<td>-ie</td>
<td>Ignore tables with columns having data types that cannot be converted.</td>
</tr>
<tr>
<td>-cu</td>
<td>Generate upper case table and column names.</td>
</tr>
<tr>
<td>-cl</td>
<td>Generate lower case table and column names.</td>
</tr>
<tr>
<td>-cc</td>
<td>Generate case-sensitive table and column names.</td>
</tr>
<tr>
<td>-sc</td>
<td>Extract shadow columns.</td>
</tr>
<tr>
<td>-st</td>
<td>Extract system tables.</td>
</tr>
</tbody>
</table>

Run schema extractor in old fglschema mode, allowing the following compatibility options:

- -c is equivalent to -cv BBBBBB..., enabling Informix SQL type to FGL data type conversion.
- -r writes partial tables definitions into the .sch file, for tables having columns with unsupported types.

**Note:** Unlike -r option, -ie skips the whole table definition.

**Usage**

The fgldbsch command line tool extracts the schema description for any database supported by the product. The .sch schema file is mandatory to compiler forms or source modules using the SCHEMA instruction.

**Related concepts**

[Database schema on page 467](#)

Defines database table structures with column type information to be reused in program variable definitions.

**fglmkstr**

The fglmkstr tool compiles .str localized string resource files.

**Syntax**

```
fglmkstr [options] source.str
```

1. options are described in Table 428: fglmkstr options on page 2081.
2. source.str is the string source file. The file extension is optional.
Options

Table 428: fglmkstr options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-V</td>
<td>Displays version information.</td>
</tr>
<tr>
<td>-h</td>
<td>Displays options for the tool.</td>
</tr>
</tbody>
</table>

Usage

The fglmkstr command line tool is used to compile .str localized string files into .42s files.

Related concepts

Compiling message files (.msg) on page 1161
The .msg message files must be compiled to .iem binary files, in order to be loaded by the runtime system.

Localized strings on page 538
Localized strings provide a means of writing applications in which the text of strings can be customized on site.

fglwsdl

The fglwsdl tool produces web services stub files for client or server programs (from WSDL / XSD).

Syntax

```
fglwsdl command [options] [argument]
```

1. `command` indicates what operation must be done by fglwsdl.
2. `options` are described in fglwsdl options.
3. `argument` is a parameter for `command`.

Options

Table 429: fglwsdl commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-V</td>
<td>Displays version information.</td>
</tr>
<tr>
<td>-h</td>
<td>Displays options for the tool.</td>
</tr>
<tr>
<td>-l</td>
<td>List services from a WSDL or variables from a XSD</td>
</tr>
<tr>
<td>-c [options] wsdl-spec</td>
<td>Generate client stub (default) to be used in a GWS client application. <code>wsdl-spec</code> is the name of a WSDL description file or the URL of a WSDL description for a published web service. Typically, <a href="http://host/service?WSDL">http://host/service?WSDL</a>. The <code>options</code> are listed in Table 430: WSDL Options on page 2082 and Table 432: Common options on page 2083.</td>
</tr>
<tr>
<td>-s [options] wsdl-spec</td>
<td>Generate server stub to be used in a GWS server application. <code>wsdl-spec</code> is the name of a WSDL description file or the URL of a WSDL description for a published web service. Typically, <a href="http://host/service?WSDL">http://host/service?WSDL</a>. The <code>options</code> are listed in Table 430: WSDL Options on page 2082 and Table 432: Common options on page 2083.</td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>-x [options] xsd-spec</code></td>
<td>Generate BDL data types from a XML schema (XSD). <code>xsd-spec</code> is the name of an XML schema file or the URL of an XSD schema resource on the web.</td>
</tr>
<tr>
<td></td>
<td>The <code>options</code> are listed in Table 431: XSD Options on page 2083 and Table 432: Common options on page 2083.</td>
</tr>
<tr>
<td><code>-regex regex value</code></td>
<td>Validate the <code>value</code> against the <code>regex</code> regular expression described in XML schema specification.</td>
</tr>
</tbody>
</table>

### Table 430: WSDL Options

<table>
<thead>
<tr>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-o file</code></td>
<td>Specify a base name for the output files.</td>
</tr>
<tr>
<td><code>-n service port</code></td>
<td>Generate only for the given service name and port type.</td>
</tr>
<tr>
<td><code>-b binding</code></td>
<td>Generate only for the given binding.</td>
</tr>
<tr>
<td><code>-prefix name</code></td>
<td>Add <code>name</code> as the prefix of the generated web service functions, variables and types. (<code>name</code> can contain <code>%s</code> for servicename, <code>%p</code> for portname and <code>%f</code> for file name)</td>
</tr>
<tr>
<td><code>-compatibility</code></td>
<td>Generate a Genero 1.xx compatibility client stub.</td>
</tr>
<tr>
<td><code>-fRPC</code></td>
<td>Force RPC convention; use RPC Convention to generate the code, regardless of what the WSDL information contains.</td>
</tr>
<tr>
<td><code>-fRPCNamespace</code></td>
<td>Generate code to support the <code>namespace</code> attribute for RPC parameters.</td>
</tr>
<tr>
<td><code>-disk</code></td>
<td>Save WSDL and all dependencies from an URL on the disk.</td>
</tr>
<tr>
<td><code>-domHandler</code></td>
<td>Generates the use of DOM in the client stub and calls to callback handlers.</td>
</tr>
<tr>
<td><code>-alias</code></td>
<td>Generates FGLPROFILE Logical names in place of URLs for the client stub.</td>
</tr>
<tr>
<td><code>-soap11</code></td>
<td>Generates only client and server stubs supporting SOAP 1.1 protocol.</td>
</tr>
<tr>
<td><code>-soap12</code></td>
<td>Generates only client and server stubs supporting SOAP 1.2 protocol.</td>
</tr>
<tr>
<td><code>-ignoreFaults</code></td>
<td>Do not generate extra code to handle SOAP faults.</td>
</tr>
<tr>
<td><code>-wsa yes no</code></td>
<td>Force support of WS-Addressing 1.0 if <code>yes</code>, disable support of WS-Addressing 1.0 if <code>no</code>, otherwise support WS-Addressing 1.0 according to the WSDL definition.</td>
</tr>
<tr>
<td><code>-mtom yes no</code></td>
<td>Override the WSDL MTOM policy with this option. If <code>yes</code>, generates the stub with MTOM support. If <code>no</code>, generates the stub without MTOM support. This option can be applied to the client side or server side.</td>
</tr>
</tbody>
</table>
Table 431: XSD Options

<table>
<thead>
<tr>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-o file</td>
<td>Name of the output file. If file has no extension, .inc is added.</td>
</tr>
<tr>
<td>-n name [ ns ]</td>
<td>Generate only for the given variable name and namespace (if there is one).</td>
</tr>
<tr>
<td>--prefix name</td>
<td>Add name as the prefix of the generated data types.</td>
</tr>
<tr>
<td>--disk</td>
<td>Save XSD and all dependencies from an URL on the disk.</td>
</tr>
<tr>
<td>--hexb64AsString</td>
<td>Generate all XSD base64 and hexBinary type as BDL STRING.</td>
</tr>
<tr>
<td></td>
<td>Note: No code is generated.</td>
</tr>
<tr>
<td></td>
<td>Generate all XSD base64 and hexBinary type as BDL STRING.</td>
</tr>
</tbody>
</table>

If the WSDL has a Message Transmission Optimization Mechanism (MTOM) policy, the tool generates any xsd:base64Binary and xsd:hexBinary as a STRING data type with the XMLOptimizedContent attribute (instead of a BYTE data type). The STRING represents a file name on disk, that will be handled as a SOAP-attached file.

**Tip:** This option is useful for avoiding loading a file into a BYTE each time you want to send a big file.

If the WSDL does not have an MTOM policy, the tool generates any xsd:base64Binary and xsd:hexBinary as a STRING data type (instead of a BYTE data type). The programmer is responsible for setting a base64 or hexbinary value to the STRING in order to avoid a XML serialization error with the peer, as the latter expects a base64 or an hexbinary value.

Table 432: Common options

<table>
<thead>
<tr>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--comment</td>
<td>Add XML comments to the generation.</td>
</tr>
<tr>
<td>--fArray</td>
<td>Force XML array generation instead of XML list when possible.</td>
</tr>
<tr>
<td></td>
<td>If the WSDL contains an XML definition of a BDL list, generate a BDL array</td>
</tr>
<tr>
<td></td>
<td>matching the same definition.</td>
</tr>
<tr>
<td>--fInheritance</td>
<td>Force generation of XML choice records for all inheritance types found in</td>
</tr>
<tr>
<td></td>
<td>the schemas, otherwise only for abstract types and elements.</td>
</tr>
<tr>
<td>--fInlineTypes</td>
<td>Force generation of TYPE definitions for all global inlined types found in</td>
</tr>
<tr>
<td></td>
<td>the schemas.</td>
</tr>
<tr>
<td>--noFacets</td>
<td>Don't generate facet constraints restricting the value-space of simple</td>
</tr>
<tr>
<td></td>
<td>data type.</td>
</tr>
<tr>
<td>--legacyTypes</td>
<td>Don't generate BIGINT, TINYINT and BOOLEAN data types.</td>
</tr>
<tr>
<td>--ignoreMixed</td>
<td>Ignore attribute mixed=&quot;true&quot; in XML schemas when generating code.</td>
</tr>
<tr>
<td>--ext schema</td>
<td>Add an external schema. See option `--extDir'</td>
</tr>
<tr>
<td>Options</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-extDir directory</td>
<td>Add all external schema files ending with .xsd in the directory. Note: External schemas for dependencies won't be included in the WSDL description or in the XSD schema if their location attributes are missing. Use this option to add a missing external schema for a WSDL or XSD dependency.</td>
</tr>
<tr>
<td>-noValidation</td>
<td>Disable XML schema validation warnings.</td>
</tr>
<tr>
<td>-autoNsPrefix nb</td>
<td>Automatic prefix generation for variables and types using a substring of the namespace by removing the nb first elements (-1 means only the last element). For example: If a variable belongs to the namespace <a href="http://www.mycompany.com/Global/Service">http://www.mycompany.com/Global/Service</a>, a value of -1 will give Service as a prefix, and a value of 1 will give Global_Service as a prefix.</td>
</tr>
<tr>
<td>-nsPrefix ns value</td>
<td>Add value as prefix of the generated variables and types belonging to namespace ns (supersede the -prefix and the -autoNsPrefix option, and can be called several times).</td>
</tr>
</tbody>
</table>

Table 433: Network options (when specifying an URL)

<table>
<thead>
<tr>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-noHTTP</td>
<td>Disable HTTP - search for the WSDL description or the XML schema and its dependencies on the client instead of the internet. Useful, for example, if a company has restricted access to the internet.</td>
</tr>
<tr>
<td>-proxy location</td>
<td>Connect via proxy where location is host[:port] or ip[:port].</td>
</tr>
<tr>
<td>-pAuth login pass</td>
<td>Proxy authentication login and password.</td>
</tr>
<tr>
<td>-hAuth login pass</td>
<td>HTTP authentication login and password.</td>
</tr>
<tr>
<td>-cert cert</td>
<td>File of the X509 PEM-encoded certificate for HTTPS purpose.</td>
</tr>
<tr>
<td>-key key</td>
<td>File of the PEM-encoded private key for HTTPS purpose.</td>
</tr>
<tr>
<td>-wCert cert</td>
<td>Certificate name in the Windows® keystore for HTTPS purposes (Windows® only).</td>
</tr>
<tr>
<td>-CA list</td>
<td>A file name with the list of concatenated X509 PEM-encoded certificate authorities. (On Windows®, if not set, the Certificate Authority list of the key store is used).</td>
</tr>
</tbody>
</table>

Usage

The fglwsdl command line tool produces the WSDL description of a web service that will be accessed by a GWS client application, or to define a WSDL description for creating a corresponding GWS server application. The tool generates the BDL data types from XML schemas (also known as XSD).

To access a remote web service, you must get the WSDL information from the service provider. Sample services can be found through UDDI registries (http://www.uddi.org), or on other sites such as XMethods (http://www.xmethods.net).

Related concepts

WS client stubs and handlers on page 3328
To access a remote Web Service, you first must get the WSDL information from the service provider.

**WS server stubs and handlers** on page 3344
Describes using a server stub from a compatible Web service that you can use in your GWS server application.

**fglrestful**
The `fglrestful` tool produces REST web services stub files for client programs using an OpenApi specification.

**Syntax**

```
fglrestful [ options ] service
```

1. `options` are described in `fglrestful` options.
2. `service` is the OpenApi service definition file or URL.

**Options**

<table>
<thead>
<tr>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-V</code> or <code>--version</code></td>
<td>Displays version information.</td>
</tr>
<tr>
<td><code>-h</code> or <code>--help</code></td>
<td>Displays options for the tool.</td>
</tr>
<tr>
<td><code>-l</code> or <code>--list path</code></td>
<td>List all the paths or resources of a service. See <code>List path or resource</code></td>
</tr>
<tr>
<td><code>-o</code> or <code>--output filename</code></td>
<td>Specify a name for the output stub file.</td>
</tr>
<tr>
<td><code>-n</code> or <code>--name resource</code></td>
<td>In a service providing access to one or more resources, specify the creation of a stub file for a given resource. See <code>Generate stub file for resource</code> on page 2086.</td>
</tr>
<tr>
<td><code>-p</code> or <code>--prefix prefix</code></td>
<td>Set a prefix for constants and variables.</td>
</tr>
<tr>
<td><code>-b</code> or `--binary [byte</td>
<td>Specify that binary types are generated as either <code>BYTE</code> or <code>FILE</code> types in the stub.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-f</code> or <code>--format XML</code></td>
<td>Specify the preferred REST format (XML or JSON).</td>
</tr>
<tr>
<td><code>-a</code> or <code>--oauth</code></td>
<td>Generates OAuth support, default is <code>openapi</code>.</td>
</tr>
<tr>
<td><code>-t</code> or <code>--token token</code></td>
<td>Specify the access token value if the service is protected by an access token.</td>
</tr>
<tr>
<td><code>-k</code> or <code>--tokenfile filename</code></td>
<td>Specify the access token file if the service is protected by an access token.</td>
</tr>
<tr>
<td><code>--comment</code></td>
<td>Add comments to the generated stub file. See <code>Output comments</code> on page 2086.</td>
</tr>
<tr>
<td><code>-W</code> or <code>--Warnings</code></td>
<td>Specify if warnings should be output or not. By default the option value is <code>yes</code>, meaning the warnings are reported. See <code>Output warnings</code> on page 2086.</td>
</tr>
</tbody>
</table>

**Usage**

You use the `fglrestful` tool to generate the GWS client stubs from an OpenApi specification file (JSON format).
For example, to access a remote web service, you must get the REST information from the service provider. If the server supports the OpenAPI specification, the fglrestful command can generate the description of the service for access by a GWS client application.

If your REST service is written with Genero REST high-level API, a specification file is provided by default when you launch the service with the query string ?openapi.json or ?openapi.yaml. If the GWS Server was started on your local machine, for example, the OpenAPI information would be at:

```
http://localhost:8090/<service-name>?openapi.json
```
or
```
http://localhost:8090/<service-name>?openapi.yaml
```

The fglrestful command line tool supports OpenApi version 3.

For usage options, run the command fglrestful without any other parameters or with the help (-h) option.

These are examples of common commands used by REST Web service developers.

**List path or resource**

The --list path resource option provides you with the option to list all the paths or resources of a service. In the example, the command lists the resources of the Web service.

```
fglrestful -l resource http://localhost:8090/MyService?openapi.json
```

**Generate stub file for resource**

Your Genero Web service may provide services for one or more resources. You can specify the creation of a stub file for a given resource with the -n option in the command. For example, this command generates a separate stub file for the "users" resource:

```
fglrestful -n users http://localhost:8090/MyService?openapi.json
```

**Output comments**

When you specify the option --comment in the command, descriptions set on the WSDescription on page 3474 attribute on parameters of a REST function are generated on the client stub.

```
fglrestful -o stub_filename --comment http://localhost:8090/MyService?openapi.json
```

**Note:** WSDescription set on the ATTRIBUTES() clause of the function (as opposed to the attribute set on parameters) are generated even if the --comment option is not specified.

**Output warnings**

If you need to see warning messages when generating the client stub, you can specify the -W option in the command. In the example warnings are written to the standard output.

```
fglrestful -W yes -o stub_filename http://localhost:8090/MyService?openapi.json
```

The warning message gives the path in the openapi.json file where you locate the issue, and gives the reason.

```
Warning in /paths/Property/GetAllProperties/{applicationID}/
{includeWillBeSold}/{update}/get/responses
Reason : Unsupported media='text/plain' on
type='WebAPI_CreditAnalysis_Models_PropertyContainer'
```

**Note:** Errors are reported by default. The error message gives same kind of path, but with Error in instead of Warning in.
Related concepts
RESTful Web services with high-level framework on page 3375
These topics give you the information you need to begin working with RESTful Web services applications using BDL function with support for attributes.
Error codes of com.WebServicesEngine on page 2843
Error codes returned by com.WebServiceEngine methods.

fglpass
The fglpass tool allows you to encrypt passwords.

Syntax

fglpass [options]

1. options are described in fglpass options.

Options

Table 435: fglpass options

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-V</td>
<td>Displays version information of the tool.</td>
</tr>
<tr>
<td>-Vssl</td>
<td>Displays OpenSSL version.</td>
</tr>
<tr>
<td>-h</td>
<td>Displays options for the tool.</td>
</tr>
<tr>
<td>-e</td>
<td>Encrypt the password with a RSA key or certificate and encode it in BASE64 form.</td>
</tr>
<tr>
<td>-d</td>
<td>Decode the BASE64 form of the password and decrypt it with a RSA private key.</td>
</tr>
<tr>
<td>-w cert</td>
<td>Windows® certificate name to encrypt the password (Windows® only)</td>
</tr>
<tr>
<td>-c cert</td>
<td>File of the PEM-encoded certificate to encrypt the password.</td>
</tr>
<tr>
<td>-k key</td>
<td>File of the PEM-encoded private key to encrypt or decrypt the password.</td>
</tr>
<tr>
<td>-enc64 file</td>
<td>File to be BASE64 encoded (result to stdout)</td>
</tr>
<tr>
<td>-dec64 file</td>
<td>BASE64 encoded file to be decoded (result to stdout)</td>
</tr>
<tr>
<td>-agent:port files</td>
<td>Start password agent on specified port to serve the list of private key files.</td>
</tr>
<tr>
<td>-gid</td>
<td>When executing fglpass in agent mode (with -agent option), allows authentication to be performed for all users belonging to the group of current users executing the command.</td>
</tr>
</tbody>
</table>

Note: Requires the FGLPROFILE entry security.global.agent.gid=true for fglrun.

Usage
The fglpass command line tool allows you to:

• Encrypt a password using a RSA key or X.509 certificate and encode it in BASE64 form.
• Run a password agent that returns (in a protected way) the passwords that grant access to the different private keys used in all your applications.
• Encode a file in BASE64 form and decode it back.

For security reasons, it is recommended to avoid storing clear passwords in a file, or leave private keys unprotected without a password. The fglpass command can be used to encrypt passwords.

**Related concepts**

*Encryption, BASE64 and password agent with fglpass tool on page 3290*

Genero Web Services supports password encryption with fglpass as password agent.

**fglWrt**

Use the fglWrt tool to manage product licenses.

**Tip:** Limitations regarding the use of the fglWrt tool are discussed in the *Install and License your Genero Products* manual.

**Syntax**

```bash
fglWrt [options]
```

1. *options* are described in Table 436: fglWrt options on page 2088.

**Options**

**Table 436: fglWrt options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-v or -V</td>
<td>Displays version information.</td>
</tr>
<tr>
<td>-h</td>
<td>Displays options for the tool.</td>
</tr>
<tr>
<td>-l license</td>
<td>Installs a license.</td>
</tr>
<tr>
<td>-m option</td>
<td>Note: To escape when prompted to enter a license number and license key, type &quot;stop&quot; at the prompt.</td>
</tr>
<tr>
<td>-u</td>
<td>Check for active users.</td>
</tr>
<tr>
<td>-k option</td>
<td>Installation key for license validation, possible options are:</td>
</tr>
<tr>
<td>-d</td>
<td>Remove current installed license.</td>
</tr>
<tr>
<td>-i</td>
<td>Clears the list of registered user sessions.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td><strong>Check or view options.</strong></td>
</tr>
<tr>
<td></td>
<td>Here <em>option</em> can be one of the following:</td>
</tr>
<tr>
<td>-a option</td>
<td>• <em>ps</em>: Shows processes on this machine.</td>
</tr>
<tr>
<td></td>
<td>• <em>env</em>: Shows current environment</td>
</tr>
<tr>
<td></td>
<td>• <em>cpu</em>: Shows number of CPU in the computer.</td>
</tr>
<tr>
<td></td>
<td>• <em>hostname</em>: Shows name of this machine.</td>
</tr>
<tr>
<td></td>
<td>• <em>info license</em>: Shows license information.</td>
</tr>
<tr>
<td></td>
<td>• <em>info stat</em>: Shows statistics of license server.</td>
</tr>
<tr>
<td></td>
<td>• <em>info users</em>: Shows all registered active users.</td>
</tr>
<tr>
<td></td>
<td>• <em>info up</em>: Shows if license server is up.</td>
</tr>
<tr>
<td>-x or -kill-session pid</td>
<td><strong>Clears the session referenced by the specified pid.</strong></td>
</tr>
<tr>
<td></td>
<td>Performs the specified action in batch mode.</td>
</tr>
<tr>
<td></td>
<td>Here <em>command</em> can be one of the following:</td>
</tr>
<tr>
<td>--batch action=command</td>
<td>• <em>target-info</em> (Get operating system information)</td>
</tr>
<tr>
<td></td>
<td>• <em>lm-info</em> (Get license manager information)</td>
</tr>
<tr>
<td></td>
<td>• <em>product-info</em> (Get license controller information)</td>
</tr>
<tr>
<td></td>
<td>• <em>license-info</em> (Get license information)</td>
</tr>
<tr>
<td></td>
<td>• install-license<em>lnum=license_num</em>lkey=license_key</td>
</tr>
<tr>
<td></td>
<td>• uninstall-license</td>
</tr>
<tr>
<td></td>
<td>• install-mkey*mkey=maintenance_key</td>
</tr>
<tr>
<td></td>
<td>• install-mkey-<em>http</em>login=USERNAME*proxy=<a href="http://myproxy*proxy-port=port-number*proxy-user=proxy-user*proxy-passwd=proxy-password">http://myproxy*proxy-port=port-number*proxy-user=proxy-user*proxy-passwd=proxy-password</a> \</td>
</tr>
<tr>
<td></td>
<td>• install-ikey*ikey=installation_key</td>
</tr>
<tr>
<td></td>
<td>• install-ikey-<em>http</em>user=USERNAME*proxy=<a href="http://myproxy*proxy-user=proxy-user*proxy-passwd=proxy-password">http://myproxy*proxy-user=proxy-user*proxy-passwd=proxy-password</a> \</td>
</tr>
<tr>
<td></td>
<td>The caret character (^) is used for parsing. In Windows® the caret is also used to escape special characters. To escape it, you may need to double (^) it up.</td>
</tr>
<tr>
<td></td>
<td>fglWrt --batch action=install-mkey^^mkey=maintenance_key</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>--batch-console action=command</td>
<td>Opens fglWrt in console mode for the batch commands.</td>
</tr>
<tr>
<td>docker-assign docker-instance-name docker-instance-id</td>
<td>Assigns the license to a running docker instance.</td>
</tr>
<tr>
<td>docker-release</td>
<td>Release a docker-assigned license.</td>
</tr>
</tbody>
</table>

**Usage**

The fglWrt command line tool is used to install, upgrade or delete licenses.

If no license is installed, it is not possible to use Genero Business Development Language.

**fglgar**

The fglgar is a tool for packaging applications for deployment as standalone on a server or on any Web server with Genero Application Server (GAS).

**Syntax**

\[fglgar\ \texttt{command} \ [\texttt{options}] \ [\texttt{argument} \ [\ldots] \]

1. \texttt{command} is an fglgar command, it can be one of the following:
   - \texttt{gar}: To create a Genero Archive file (.gar).
   - \texttt{war}: To create a Java Web Archive file (.war).
   - \texttt{run}: To execute an application packaged in a .war (including a .gar).
2. \texttt{options} are specific to each command as described below.
3. \texttt{argument} is a parameter for \texttt{command}.

The fglgar tool supports the following commands:

- \texttt{fglgar gar} [\texttt{options}]
  
The \texttt{gar} command creates a .gar Genero Archive file. Options are described in Table 438: fglgar gar options on page 2091.
- \texttt{fglgar war} [\texttt{options}]

The output is JSON formatted. You can enable / disable indentation on the output. The default is indented. To quit the console type \texttt{exit}. Type \texttt{help} for instructions on its use.
The **war** command creates a `.war` Java Web Archive file embedding a Genero Archive and GAS in a jar file (JGAS). It also provides the option to add a customized Genero Browser Client (GBC). Options are described in Table 439: `fglgar war options` on page 2092.

```
fglgar run [options]
```

The **run** command executes a Genero Archive application previously packaged as a `.war` file. Options are described in Table 440: `fglgar run options` on page 2092.

### Options

**Table 437: fglgar options without command**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-V or --version</td>
<td>Displays version information.</td>
</tr>
<tr>
<td>-h or --help</td>
<td>Displays options for the tool.</td>
</tr>
</tbody>
</table>

**Table 438: fglgar gar options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h or --help</td>
<td>Displays help for the <code>gar</code> command.</td>
</tr>
<tr>
<td>-v or --verbose</td>
<td>Displays the verbose output of additional information.</td>
</tr>
<tr>
<td>-q or --quiet</td>
<td>Operates in silent mode</td>
</tr>
<tr>
<td>-o or --output <strong>output_file_path</strong></td>
<td>The relative or absolute path to the archive file to create. If not specified, the archive defaults to the name of current directory where the command is run.</td>
</tr>
<tr>
<td>-s or --input-source <strong>directory</strong></td>
<td>Directory to archive.</td>
</tr>
<tr>
<td>--resource <strong>directory</strong></td>
<td>Specifies the Genero Archive resource directory where the application's public images are found. (These are common or can be shared by all your applications.)</td>
</tr>
<tr>
<td>--trigger-component <strong>component_name</strong></td>
<td>Specifies Genero Archive application trigger execution component. (Optional).</td>
</tr>
<tr>
<td>--deploy-trigger <strong>command</strong></td>
<td>(Optional) Specifies the Genero Archive trigger command to deploy the application. Generates a MANIFEST file, if none exists. Raises errors if a MANIFEST already exists.</td>
</tr>
<tr>
<td>--undeploy-trigger <strong>command</strong></td>
<td>(Optional) Specifies the Genero Archive trigger command to undeploy the application. Generates a MANIFEST file, if none exists. Raises errors if a MANIFEST already exists.</td>
</tr>
<tr>
<td>--application <strong>application_file</strong></td>
<td>Specifies the application configuration or executable file. If you specify executable (42r or 42m) files instead of xcf file, xcf files are created automatically based on default configuration defined in the GAS as.xcf file. Multiple applications may be specified. Generates a MANIFEST file, if none exists. Raises errors if a MANIFEST already exists.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>--service application_file</td>
<td>Specifies the service configuration file or executable. If you specify executable (42r or 42m) files instead of xcf file, xcf files are created automatically based on default configuration defined in the GAS as.xcf file. Multiple applications may be specified. Generates a MANIFEST file, if none exists. Raises errors if a MANIFEST already exists.</td>
</tr>
</tbody>
</table>

### Table 439: fglgar war options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h or --help</td>
<td>Displays help for the war command.</td>
</tr>
<tr>
<td>-v or --verbose</td>
<td>Displays the verbose output of additional information.</td>
</tr>
<tr>
<td>-q or --quiet</td>
<td>Operates in silent mode.</td>
</tr>
<tr>
<td>-o or --output</td>
<td>Specifies the relative or absolute path to the war file to create. If not specified, the war defaults to the name of the current directory where the command is run.</td>
</tr>
<tr>
<td>-g or --input-gar</td>
<td>Specifies the Genero Archive (gar) file you want to use to create the war. This option is mandatory.</td>
</tr>
<tr>
<td>-w or --web-content</td>
<td>Specifies Java Web content directory. This is optional but it allows you specify a different Java Servlet content directory if needed to package some additional Java applications or files. See the Java documentation.</td>
</tr>
<tr>
<td>-c or --gbc</td>
<td>Specifies your customized Genero Browser Client (GBC). By default, the gbc installed with the FGLGWS package is embedded in the war file.</td>
</tr>
</tbody>
</table>

### Table 440: fglgar run options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h or --help</td>
<td>Displays help for the run command.</td>
</tr>
<tr>
<td>-w or --war</td>
<td>Specifies the war file for input.</td>
</tr>
<tr>
<td>-p or --http-port</td>
<td>Specifies the port where the GAS is accessible. If not set, the default is 8080.</td>
</tr>
<tr>
<td>-P or --https-port</td>
<td>Specifies the port for secure connections (HTTPS) on the server where the GAS is accessible. If not set, the default is 443.</td>
</tr>
<tr>
<td>-E or --resource-overwrite</td>
<td>Specifies the GAS resources you want to overwrite, such as the directory of the logs. For example,</td>
</tr>
<tr>
<td></td>
<td>-E log.file.path=<code>pwd</code></td>
</tr>
</tbody>
</table>

### Usage

The fglgar command line tool is used to create Genero archive (gar), or Java Web archive (war) files. You can use it for the following methods of application deployment:

- You can deploy Web applications or services in a gar file on any Web server where the GAS is installed.
• You can deploy Genero applications and services on a Java EE server like Apache Tomcat®, or Glassfish (via war) and then with the browser you can access your applications and services.

• You can also run Web applications or services in a Java servlet (war) as standalone on a machine (without any Java EE Web server) and access your applications and services via a browser.

In other words, the fglgar tool allows you to develop, package, and test your Web application or service on a browser without GAS being installed on the server.

**Important:**

• FGLGWS needs to be installed on the server to interpret the Genero 4GL applications and services.

• You need a Java Runtime Environment (JRE) version that is at least version 1.8 or greater.

For usage options, run the command fglgar without any other parameters or with the help (-h) option.

**Tip:** For help with using a specific command, for example gar, running fglgar gar, is the same as running fglgar gar --help.

**Related concepts**

Packaging web applications on page 2183
Describes methods of packaging the runtime files and resources of your web applications and services using the fglgar tool.

### fglgitformat

The fglgitformat tool reformat the source lines changed in a GIT history.

**Syntax**

```plaintext
fglgitformat [options] [commit] [file [...]]
```

1. **commit** is a GIT commit specification (like HEAD~2). Uses HEAD by default.
2. **file** is a filename to process. When no file is specified, reformat changes for all modified files.
3. **options** are described in Table 441: fglgitformat options on page 2093.

**Options**

#### Table 441: fglgitformat options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h or --help</td>
<td>Displays options for the tool.</td>
</tr>
<tr>
<td>-i or --inplace</td>
<td>Reformats the source files instead of writing the diff with reformatted code to stdout.</td>
</tr>
<tr>
<td>-v or --verbose</td>
<td>Output the new diff to stdout. Ineffective without -i/--inplace.</td>
</tr>
</tbody>
</table>

**Usage**

The fglgitformat command line tool reformats changes registered in a GIT project history, by using the formatter (fglcomp --format) to beautify the modified lines.

By default, the diff text is written to the (stdout) standard output stream. Use the --inplace option to modify directly the source files referenced by the GIT diff. Optionally, combine --inplace with --verbose to print the changes to stdout:

```plaintext
fglgitformat --inplace --verbose
```
When specifying a commit such as HEAD~2, the tool reformat all changes relative to the commit, plus the changes staged for the next commit:

```
fglgitformat --inplace HEAD~2
```

When no commit is provided, `fglgitformat` only reformat the changes staged for the next commit (relative to the HEAD).

By default, all modified .4gl source files are processed. To reformat the changes in specific files, these can be provided as arguments:

```
fglgitformat --inplace common/sql_utils.4gl common/gui_utils.4gl
```

**Tip:** If the code modifications to be reformatted are part of the last commit, after reformatting with `fglgitformat`, consider using `git commit --amend`, to merge the reformatting with your last commit. Or, commit the reformatting as a separate commit, and reorganize the history with interactive rebase (`git rebase -i`).

**Related concepts**
- Source code beautifier on page 2174
- `fglformatdiff` on page 2094

---

### fglformatdiff

The `fglformatdiff` tool reformat source lines of a unified diff text.

**Syntax**

```
fglformatdiff [options]
```

1. The `fglformatdiff` command reads input from stdin stream.
2. `options` are described in Table 442: `fglformatdiff` options on page 2094.

**Options**

**Table 442: fglformatdiff options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-h</code> or <code>--help</code></td>
<td>Displays options for the tool.</td>
</tr>
<tr>
<td><code>-i</code> or <code>--inplace</code></td>
<td>Reformat the source files referenced in the diff text instead of writing the diff with reformatted code to stdout.</td>
</tr>
<tr>
<td><code>-p=num</code> or <code>--strip=num</code></td>
<td>Strip the smallest prefix containing num leading slashes from the file name found in the diff data.</td>
</tr>
<tr>
<td><code>-v</code> or <code>--verbose</code></td>
<td>Output the new diff to stdout. Ineffective without <code>-i</code> or <code>--inplace</code>.</td>
</tr>
<tr>
<td><code>-e</code> or <code>--assume-name=name</code></td>
<td>editor support: assume this name when reading from a temporary file.</td>
</tr>
<tr>
<td><code>-c</code> or <code>--cursor=pos</code></td>
<td>editor support: define the the cursor position.</td>
</tr>
</tbody>
</table>

**Usage**

The `fglformatdiff` command line tool reads a unified diff text from the (stdin) standard input stream, calculates what lines have changed and calls the formatter (`fglcomp --format`) to reformat the modified lines.
**Tip:** If GIT is your version control system, use `fglgitformat` instead of `fglformatdiff`: `fglgitformat` is designed for GIT. `fglformatdiff` is provided for other version control systems such as CVS and SVN.

The `fglformatdiff` tool is to be used after modifying sources, and before committing the changes into the repository of the version control system.

**Important:** The diff data must be in unified format and contain only modified lines (with no context lines). To get no context lines, use the `diff -U/ --unified` option with the parameter value 0 (zero).

By default, `fglformatdiff` writes the new diff result to the `stdout` standard output stream, after reformatting the lines provided in the input diff text. Use the `--inplace` option to modify directly the source files referenced by the diff text. Additionally, you can use the `--verbose` option to print the changes to `stdout`:

```
svn diff --diff-cmd=diff -x-U0 | fglformatdiff --inplace --verbose
```

Use the `--strip=NUM` option to remove leading path elements of the file names of the diff text. For example, when in the root directory or a GIT project, a `git diff` will produce filenames with a leading `a/` and `b/` prefixes:

```
$ git diff -U0
diff --git a/database/tools/ext.4gl b/database/tools/ext.4gl
    --- a/database/tools/ext.4gl
    +++ b/database/tools/ext.4gl
    ...                      ...
```

To process such diff text with the `fglformatdiff`, you must use the `--strip=1` option:

```
git diff -U0 | fglformatdiff --inplace --strip=1
```

**Related concepts**

- **Source code beautifier** on page 2174
  Reformat the source code for better readability.

- **fglgitformat** on page 2093
  The `fglgitformat` tool reformats the source lines changed in a GIT history.

**fpi**

The `fpi` tool displays product version information.

**Syntax:**

```
fpi [options]
```

1. `options` are described in **Table 443: fpi options** on page 2095.

**Options**

**Table 443: fpi options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-V</td>
<td>Displays version information for the Genero BDL package.</td>
</tr>
<tr>
<td>-h</td>
<td>Displays options for the tool.</td>
</tr>
<tr>
<td>-l or --list</td>
<td>List version information for all Genero BDL command line tools.</td>
</tr>
</tbody>
</table>
Usage

The fpi command line tool can be used to list version information for all command line tools available in the Genero BDL package.

gmabuildtool

The gmabuildtool is a utility to create and test applications for an Android™ device.

Syntax

```
gmabuildtool command [option [...]]
```

1. `command` can be one of the following:
   - `updatesdk`: updates the Android™ SDK, to download packages required by GMA.
   - `scaffold`: manages scaffold archives.
   - `build`: builds an APK package.
   - `test`: deploys and launches an app on the device or emulator.
2. `option` can be a general or command-specific option, as described in Options on page 2096.

Options

Table 444: General gmabuildtool options

<table>
<thead>
<tr>
<th>Option</th>
<th>Short option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--android-sdk path</code></td>
<td>-as</td>
<td>The path to the Android™ SDK installation directory.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If not specified, it defaults to the ANDROID_SDK_ROOT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>environment variable. If ANDROID_SDK_ROOT is not defined,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>defaults to ANDROID_HOME.</td>
</tr>
<tr>
<td><code>--help</code></td>
<td>-h</td>
<td>Display options for the tool.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Path to the file containing gmabuildtool options.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Define all options in a file and pass the file to the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>gmabuildtool command with the --input-options argument.</td>
</tr>
<tr>
<td><code>--input-options path</code></td>
<td>-i</td>
<td>The options file must use the following format:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>command</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>option-name option-value</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td><code>--java-home path</code></td>
<td>-jh</td>
<td>Java home path.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default is JAVA_HOME.</td>
</tr>
<tr>
<td><code>--proxy-host host</code></td>
<td>-ph</td>
<td>Defines the proxy host.</td>
</tr>
<tr>
<td><code>--proxy-host port</code></td>
<td>-pp</td>
<td>Defines the proxy port.</td>
</tr>
<tr>
<td><code>--verbose-fine</code></td>
<td>-v</td>
<td>Verbose mode (level 1)</td>
</tr>
<tr>
<td><code>--verbose-finer</code></td>
<td>-vv</td>
<td>Verbose mode (level 2)</td>
</tr>
<tr>
<td><code>--verbose-finest</code></td>
<td>-vvv</td>
<td>Verbose mode (level 3) - shows all possible logs.</td>
</tr>
<tr>
<td>Option</td>
<td>Short option</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>--------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>--version</td>
<td>-V</td>
<td>Displays version information.</td>
</tr>
</tbody>
</table>

Table 445: `gmabuildtool updatesdk` options

<table>
<thead>
<tr>
<th>Option</th>
<th>Short option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--accept-licenses</td>
<td>-al</td>
<td>Silently accept Android™ SDK licenses when the Android™ SDK is updated.</td>
</tr>
<tr>
<td>--no-install-extras</td>
<td>-uN</td>
<td>Avoid installation of extra SDK modules.</td>
</tr>
</tbody>
</table>

Table 446: `gmabuildtool scaffold` options

<table>
<thead>
<tr>
<th>Option</th>
<th>Short option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--install-plugins</td>
<td>-ip</td>
<td>Install the specified plugins in the scaffold archive.</td>
</tr>
<tr>
<td>plugin-list</td>
<td></td>
<td>The <code>plugin-list</code> must be a comma-separated list of plugins.</td>
</tr>
<tr>
<td>--list-plugins</td>
<td>-lp</td>
<td>List the plugins available in the scaffold archive.</td>
</tr>
</tbody>
</table>

Table 447: `gmabuildtool build` options

<table>
<thead>
<tr>
<th>Option</th>
<th>Short option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--clean</td>
<td>-c</td>
<td>Cleans the intermediate build files before a rebuild.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Use the <code>--clean</code> option if the previous build was interrupted or has failed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> The <code>--clean</code> option does not remove and replace the scaffold, as done by the <code>--build-force-scaffold-update</code> option.</td>
</tr>
<tr>
<td>Option</td>
<td>Short option</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>--------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>--build-app-colors color-list</td>
<td>-bc</td>
<td>Define the Android™ color theme for the app (Android™ 5.0+ / SDK 21+)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The value must be a comma-separated list of four hexadecimal RGB colors: #F44336, #B71C1C, #EF9A9A,...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The position of the RGB value in the color list defines its purpose:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1. <strong>colorPrimary</strong>: The main color used in the app.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. <strong>primaryDark</strong>: The color used for the status bar and the navigation bar.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3. <strong>accent</strong>: The accent color used for widgets and table lines.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4. <strong>actionBarText</strong>: The foreground color for the texts in the action bar.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5. <strong>primaryText</strong>: The text color for items in the whole application.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6. <strong>windowBackground</strong>: The window background color.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7. <strong>navigationBarBackground</strong>: The background color of the bottom bar.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>By default, the color theme is the Genero purple color.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Relative path to the main module of the application (can be .42m or .42r).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Defaults to main.42m</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Defines the path to the application program files (.42m, .42f, etc)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The contents of this directory will be zipped and bundled inside APKs. This option can handle an already zipped Genero program archive.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If not specified, defaults to the current working directory.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note</strong>: The path defined by this option is used as base directory for other options such as --build-project and application icon resources options.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Defines the path to application icon in hdpi.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default is top-dir/gma/ic_app_hdpi.png, where top-dir is defined by the --build-app-genero-program option.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Defines the path to application icon in mdpi.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default is top-dir/gma/ic_app_mdpi.png, where top-dir is defined by the --build-app-genero-program option.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Defines the path to application icon in xhdpi.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default is top-dir/gma/ic_app_xhdpi.png, where top-dir is defined by the --build-app-genero-program option.</td>
</tr>
<tr>
<td>Option</td>
<td>Short option</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>--------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>--build-app-icon-xxhdpi path</td>
<td>-bixxh</td>
<td>Defines the path to application icon in xxhdpi. Default is top-dir/gma/ic_app_xxhdpi.png, where top-dir is defined by the --build-app-genero-program option.</td>
</tr>
<tr>
<td>--build-app-name app-name</td>
<td>-bn</td>
<td>Application name. If not specified, the application name defaults to the current working directory.</td>
</tr>
<tr>
<td>--build-app-package-name name</td>
<td>-bpn</td>
<td>APK package name. It is recommended to format the package name as &quot;com.organization-name.app-name&quot;. If not specified, the application package name defaults to com.example.current-working-directory.</td>
</tr>
<tr>
<td>--build-app-permissions permissions</td>
<td>-ba</td>
<td>Android™ application permissions. The list of permissions is provided as a comma separated list of android.permission.* identifiers. For more details, see Android permissions on page 3595.</td>
</tr>
<tr>
<td>--build-app-version-code version-code</td>
<td>-bvc</td>
<td>Application version code. For example: 100915 The value of this option must be an integer (do not use decimal numbers).</td>
</tr>
<tr>
<td>--build-app-version-name version-name</td>
<td>-bvn</td>
<td>Application version name. For example: 10.09.15 This will be the actual app version visible on devices.</td>
</tr>
<tr>
<td>--build-apk-outputs path</td>
<td>-bo</td>
<td>Defines the destination folder where the APK packages must be created.</td>
</tr>
<tr>
<td>--build-cordova cordova-plugin-names</td>
<td>-bco</td>
<td>Defines Cordova plugins to be embedded in the app package. When specifying multiple cordova plugins, use the comma (,) as separator. The name of the plugin must match the Git repository name. It is case-sensitive. Note: To get the list of available Cordova plugins, use the gmabuildtool scaffold --list-plugins command. For further information, see Cordova plugins on page 3644.</td>
</tr>
<tr>
<td>--build-force-scaffold-update</td>
<td>-bfsu</td>
<td>Forces to re-create the app project directory with the original GMA scaffold directory (defined by --build-project)</td>
</tr>
<tr>
<td>Option</td>
<td>Short option</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>--------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>--build-gbc-runtime gbc-archive</code></td>
<td>-bgr</td>
<td>Defines the GBC to be used for Universal Rendering. <code>gbc-archive</code> is the GBC ZIP archive. For more details, see the Create a runtime zip topic in the Genero Browser Client User Guide.</td>
</tr>
<tr>
<td><code>--build-jarsigner-alias alias</code></td>
<td>-bja</td>
<td>Jarsigner alias. This is the alias provided to the keystore utility to build the keystore file to sign the app. Used when APK artifacts are signed.</td>
</tr>
<tr>
<td><code>--build-jarsigner-keypass keypass</code></td>
<td>-bjk</td>
<td>Jarsigner keypass. Specifies the password used to protect the private key of the keystore entry addressed by the alias specified in the --build-jarsigner-alias option. The password is required when using jarsigner to sign a JAR file. Used when APK artifacts are signed.</td>
</tr>
<tr>
<td><code>--build-jarsigner-keystore path</code></td>
<td>-bjks</td>
<td>Jarsigner keystore path. This is the path to the keystore file generated by the keystore utility to sign the app. Used when APK artifacts are signed.</td>
</tr>
<tr>
<td><code>--build-jarsigner-storepass storepass</code></td>
<td>-bjs</td>
<td>Jarsigner storepass. Specifies the password that is required to access the keystore. Used when APK artifacts are signed.</td>
</tr>
<tr>
<td>`--build-mode [release</td>
<td>debug]`</td>
<td>-bm</td>
</tr>
<tr>
<td><code>--build-output-apk-name name</code></td>
<td>-ban</td>
<td>The file name of the APK package is formed from: 1. The APK file name prefix defined by the --build-output-apk-name option (by default, &quot;app&quot;), 2. When building a debug version, the --debug suffix, 3. The .apk file extension. For example, if the APK file name prefix is MyApp and is a debug package, the resulting APK file name will be: MyApp-debug.apk. Defines the path to the directory containing the original GMA binary archive files (i.e. scaffolding), or the directory containing the Android™ Studio project, when building a customized GMA. Default is <code>top-dir/gma/project</code>, where <code>top-dir</code> is defined by the --build-app-genero-program option.</td>
</tr>
<tr>
<td><code>--build-project path</code></td>
<td>-bp</td>
<td></td>
</tr>
<tr>
<td>Option</td>
<td>Short option</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------</td>
<td>--------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>--build-quietly</td>
<td>-bq</td>
<td>Forces a silent build, by answering yes to all questions asked during the build process. By default, the user must answer to the build questions by yes/no.</td>
</tr>
<tr>
<td>--build-status-icon-hdpi</td>
<td>-bsh</td>
<td>Status icon path for hdpi (high dots per inch) size. The default path is <code>top-dir/gma/ic_status_hdpi.png</code>, where <code>top-dir</code> is defined by the --build-app-genero-program option.</td>
</tr>
<tr>
<td>path</td>
<td></td>
<td>If this option is not specified, yet you provide default files under the gma directory named like those defined for the default path, your package will use these files. If you don't provide any status icon files, the default files are used.</td>
</tr>
<tr>
<td>--build-status-icon-mdpi</td>
<td>-bsm</td>
<td>Status icon path for mdpi (medium dots per inch) size. The default path is <code>top-dir/gma/ic_status_mdpi.png</code>, where <code>top-dir</code> is defined by the --build-app-genero-program option.</td>
</tr>
<tr>
<td>path</td>
<td></td>
<td>If this option is not specified, yet you provide default files under the gma directory named like those defined for the default path, your package will use these files. If you don't provide any status icon files, the default files are used.</td>
</tr>
<tr>
<td>--build-status-icon-xhdpi</td>
<td>-bsxh</td>
<td>Status icon path for xhdpi (extra-high dots per inch) size. The default path is <code>top-dir/gma/ic_status_xhdpi.png</code>, where <code>top-dir</code> is defined by the --build-app-genero-program option.</td>
</tr>
<tr>
<td>path</td>
<td></td>
<td>If this option is not specified, yet you provide default files under the gma directory named like those defined for the default path, your package will use these files. If you don't provide any status icon files, the default files are used.</td>
</tr>
</tbody>
</table>

Table 448: gmaBuildTool test options

<table>
<thead>
<tr>
<th>Option</th>
<th>Short option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--test-apk path</td>
<td>-ta</td>
<td>Path to the APK file to deploy and launch for testing.</td>
</tr>
</tbody>
</table>

Related concepts

Building Android apps with Genero on page 3591
Genero provides a command-line tool to create applications for Android™ devices.

**gmibuildtool**

The gmibuildtool is a utility to create and test applications for an iOS devices.

**Syntax**

```
gmibuildtool [options]
```

1. **options** are described in Options on page 2102.

**Options**

**Table 449: gmibuildtool options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--app-name application-name</td>
<td>Display name of the mobile app.</td>
</tr>
<tr>
<td></td>
<td>This option can be specified to define the display name of the app, it sets the CFBundleDisplayName property in the Info.plist file.</td>
</tr>
<tr>
<td></td>
<td>If not specified, the name defaults to &quot;Noname&quot;.</td>
</tr>
<tr>
<td>--app-version application-version</td>
<td>Defines app version visible to the users on the App Store.</td>
</tr>
<tr>
<td></td>
<td>This option is mandatory and sets CFBundleVersion properties in the Info.plist file.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> If the --build-number option is not used, --app-version will also set both the CFBundleShortVersionString property.</td>
</tr>
<tr>
<td></td>
<td>In iTunes® Connect, you define the version of your app, that must match the CFBundleVersion property in the Info.plist file of the app. If these versions do not match, the app cannot be published. Once the app is visible on App Store, the version specified in iTunes® Connect shows up in the &quot;Version&quot; section of the application page.</td>
</tr>
<tr>
<td></td>
<td>The recommendation for the app version number is that it is a string comprised of three period-separated integers. For example: &quot;1.4.2&quot;</td>
</tr>
<tr>
<td>--bundle-id bundle-identifier</td>
<td>Defines the Bundle Identifier (a.k.a. App Id) for the app.</td>
</tr>
<tr>
<td></td>
<td>This option is mandatory and sets the CFBundleIdentifier property in the Info.plist file.</td>
</tr>
<tr>
<td></td>
<td>A bundle identifier is the unique identifier of your app, to let iOS recognize new app versions. When developing for the simulator, you can choose your own identifier. When creating an application for the App Store, the bundle identifier must be registered with Apple.</td>
</tr>
<tr>
<td></td>
<td>If not specified, the name defaults to &quot;noname&quot; (for prototyping).</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>--build-cordova cordova-plugin-names</strong></td>
<td>Defines Cordova plugins to be embedded in the app package. When specifying multiple cordova plugins, use the comma (,) as separator. The name of the plugin must match the Git repository name. It is case-sensitive. <strong>Note:</strong> To get the list of available Cordova plugins, use the <code>gmibuildtool --list-plugins</code> command.</td>
</tr>
<tr>
<td><strong>--build-number build-number</strong></td>
<td>Defines the build number used to upload a new binary of the same app version. This option must be used to distinguish different builds for the same app version. It sets the <code>CFBundleShortVersionString</code> property in the <code>Info.plist</code> file. The build number needs to be incremented in order to upload a new binary version of the same app version in iTunes® Connect. If this option is not used, the build number defaults to the version specified with the <code>--app-version</code> option. The build number is a string comprised of three period-separated integers. For example: “1.4.2”</td>
</tr>
<tr>
<td><strong>--certificate identity</strong></td>
<td>Name of a certificate to sign the app. This option is mandatory to build apps for a physical device or for the app store. The certificate can be found in the Keychain® access program, in the “Common Name” field of the certificate panel. The command <code>security find-identity -v</code> can be used to list all available certificates.</td>
</tr>
<tr>
<td>**--crypto {yes</td>
<td>no}**</td>
</tr>
<tr>
<td><strong>--device device-name</strong></td>
<td>Defines the name of a device or simulator.</td>
</tr>
<tr>
<td>• By default, when not specifying the <code>--device</code> option, a GMI.app directory is created for the simulator.</td>
<td></td>
</tr>
<tr>
<td>• When specifying the <code>--device booted</code> option, the GMI.app directory is created and the app is installed on the booted simulator.</td>
<td></td>
</tr>
<tr>
<td>• When specifying the <code>--device phone</code> option, the GMI.app directory and .ipa file are created.</td>
<td></td>
</tr>
<tr>
<td>• When specifying the <code>--device physical-device-name</code> option (with a real physical device name plugged on your Mac), the GMI.app directory and .ipa file are created and the app is installed on the device. <strong>Note:</strong> Use the <code>instruments -s Xcode®</code> command to find the list of available devices (simulators or connected devices).</td>
<td></td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>--extension-libs</td>
<td>Specify the libraries to use when compiling and linking the app. This option is used when you want to provide your own C extension or custom front calls.</td>
</tr>
<tr>
<td>--help</td>
<td>Displays options for the tool.</td>
</tr>
<tr>
<td>--gbc gbc-archive</td>
<td>Defines the GBC to be used for Universal Rendering.</td>
</tr>
<tr>
<td>--icons icons-dir</td>
<td>Provides the directory where the application icons are located. By default, the application icons directory is <code>current-working-dir/gmi</code>. The name of the app icon files must be: <code>icon_57x57.png, icon_72x72.png, icon_29x29.png, icon_40x40.png, icon_120x120.png, icon_152x152.png, icon_58x58.png, icon_76x76.png, icon_80x80.png</code></td>
</tr>
<tr>
<td>--install [yes</td>
<td>no]</td>
</tr>
<tr>
<td>--install-plugins github-url</td>
<td>This option installs additional plugins in the GMI installation directory.</td>
</tr>
<tr>
<td>--launch-images launch-images-dir</td>
<td>The directory where launch images are located. By default, the launch images directory is <code>current-working-dir/gmi</code>. Note: This option is ignored if the --storyboard option is provided. The name of the image files must be: <code>Default.png, Default@2x.png Default-568h@2x.png, Default-Portrait-667h@2x.png Default-Landscape-667h@2x.png, Default-Portrait-736h@3x.png, Default-Landscape-736h@3x.png, Default-Landscape.png Default-Portrait@2x.png, Default-Landscape@2x.png</code>. Each file name corresponds to a device type (you may not need to provide all files if you target only recent iOS devices), see Apple® Developer documentation for more details about launch images.</td>
</tr>
<tr>
<td>--list-plugins</td>
<td>This option lists the shipped plugins and additional plugins installed in the GMI installation directory.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| `--mode [debug|release]` | Controls the debug or release mode for the app. By default, the mode is debug. Note that the provisioning profile must correspond:
- `--mode debug`: Development provisioning profile.
- `--mode release`: Distribution provisioning profile. |
| `--output ipa-file-name` | Path to output IPA and APP files to be generated. By default, a "build" directory is created, with subdirectories containing the .ipa and .app files. An IPA file is created when building an application for a physical device and the App Store. The IPA file is not needed and will not be created when building for the simulator. |
| `--program-files program-dir` | Path to Genero BDL program files (.42m, .42f, etc). By default, the program files directory is the current work directory. Following files are automatically excluded: *.4gl, *.per, *.msg, *.str, *.sch, [Mm]akefile, *.42d, [Mm]akefile, *.chdmo, *.xib, build/ (the build directory), gmi/ (this folder is the default location of LaunchScreens and AppIcons). If the file gmlignore exists, then this file contains additional files to be ignored. |
| `--provisioning provisioning-file` | Path to the provisioning profile (.mobileprovision). The provisioning profile is mandatory to build apps for a physical device or for the app store. Provisioning profiles can be found in $HOME/Library/MobileDevice/ProvisioningProfiles/ |
| `--Storyboard storyboard-file` | Path to the storyboard file, to get a splash screen to be displayed when the app starts. This file is an alternative for Launch Screens (`--launch-images` option). This option is mandatory if you do not provide launch images with the `--launch-images` option. The default storyboard is showing an empty navigation bar and an empty toolbar. If the storyboard references images, gmibuildtool searches for the images in the same directory the storyboard is in, and bundles the images with the application. |
| `--verbose [yes|no]` | Enable the verbose mode. |
| `--version` | Displays version information. |

**Related concepts**

Building iOS apps with Genero on page 3606
Genero provides a command-line tool to build applications for iOS devices.

## Compiling source files

Describes how to build the runtime files from source files.

### Compiling form specification files (.per)

The `.per` form definition files must be compiled to `.42f` XML files, in order to be loaded by the runtime system.

#### Understanding .per source compilation

Form specification files (with `.per` file extension) must be compiled to runtime form files (with `.42f` file extension) by using the `fglform` tool.

Compiled form files are XML independent from the platform and processor architecture.

The following lines show a compilation in a UNIX™ shell session:

```bash
$ cat form.per
LAYOUT
GRID
{
  [f01   ]
}
END
END
ATTRIBUTES
f01 = FORMONLY.field1;
END

$ fglform form.per

$ ls -s form.42f
  4 form.42f
```

### Automatic compilation of imported modules

When compiling a `.per` module that includes other forms with the `FORM` instruction, `fglform` will automatically compile the included forms, if the `.per` source is more recent as the `.42f` file. The included forms can be located in a different directory as the main form.

For more details, see **FORM clause** on page 1303.

### Handling fglform compiler errors

If an error occurs, the compiler writes an error file with the `.err` extension.

```bash
$ cat form.per
LAYOUT
GRID
{
}

$ fglform form.per
The compilation was not successful. Errors found: 1.
The file 'form.err' has been written.

$ cat form.err
LAYOUT
GRID
```
With the `-M` option, you can force the compiler to display an error message instead of generating an `.err` error file (line break added for documentation readability):

```bash
$ fglform -M form.per
form.per:4:1:4:1: error: (-6803)
   A grammatical error has been found at '}', expecting SCR_TEXT.
```

**Produce compiler warnings with `-W`**

By default, the compiler does not raise any warnings. You can turn on warnings with the `-W` option:

```bash
$ cat form.per
LAYOUT
GRID
{ [f01 ] }
END
ATTRIBUTES
f01 = FORMONLY.field1, WIDGET="COMBO";
END

$ fglform -Wall form.per
form.per:9: warning (-8005) Deprecated feature: The WIDGET attribute is obsolete
```

**Compiling message files (.msg)**

The `.msg` message files must be compiled to `.iem` binary files, in order to be loaded by the runtime system.

In order to use message files in a program, the message source files (with `.msg` extension) must be compiled with the `fglmkmsg` utility to produce compiled message files (with `.iem` extension).

The following command line compiles the message source file `mess01.msg`:

```bash
fglmkmsg mess01.msg
```

This creates the compiled message file `mess01.iem`.

For backward compatibility, you can specify the output file as second argument:

```bash
fglmkmsg mess01.msg mess01.iem
```

The `.iem` compiled version of the message file must be distributed on the machine where the programs are executed.

**Compiling string resource files (.str)**

The `.str` source string files must be compiled to `.42s` binary files, in order to be loaded by the runtime system.

To compile a source string file, use the `fglmkstr` compiler:

```bash
$ fglmkstr filename.str
```

The `fglmkstr` tool generates a `.42s` file with the `filename` prefix.
Important: When compiling a `.str` source string file, you must set the locale (character set) corresponding to the encoding used in the `.str` file.

Related concepts

Extracting strings from sources on page 542
Localized strings can be easily extracted from `.4gl` and `.per` source files.

Compiling program code files (.4gl)
The `.4gl` source files must be compiled to `.42m` p-code files, in order to be loaded by the runtime system.

Understanding .4gl source compilation

Genero BDL source code modules (with `.4gl` file extension) must be compiled to p-code modules (with `.42m` file extension) by using the `fglcomp` tool.

Compiled p-code modules are independent of the platform and processor architecture. They are interpreted by the Genero runtime system (`fglrun`).

The following lines show the compilation of the `prog.4gl` source, in a UNIX™ shell session:

```
$ cat prog.4gl
MAIN
  DISPLAY "hello"
END MAIN
$ fglcomp prog.4gl
$ ls -s prog.42m
  4 prog.42m
```

Verbose compilation

Consider using the `--verbose` option of the compiler to get detailed information about the source compilation:

```
$ fglcomp --verbose main.4gl
[loading fglhelp]
[loading main.4gl]
[loading mod1.4gl]
[loading mod2.4gl]
[loading mod2.42m]
[loading mod1]
[loading mod1.42m]
[building main]
[building main.42m]
[total modules: 4 variables: 6 functions: 274 types: 9 fields: 10]
```

Compiling several `.4gl` sources in a single command

Several `.4gl` source files can be provided to `fglcomp`: The compiler builds a dependency tree of imported modules, and compiles the files in the calculated order.

For example, if the `main.4gl` module imports `module1.4gl`, but does not import `module2.4gl`, when passing `main.4gl module1.4gl module2.4gl` as arguments to `fglcomp`, the compiler will first compile the `module1.4gl` because it is imported by `main.4gl`, then `main.4gl` and finally `module2.4gl` (assuming no `.42m` file exists before executing this command):

```
$ rm *.42m
$ fglcomp --verbose main.4gl module1.4gl module2.4gl
[loading fglhelp]
```
The `fglcomp` compiler supports MATCHES-style pattern on the command line, to achieve pathname expansion:

```text
fglcomp [a-z]*.4gl
```

This feature exists on any supported platform, including Microsoft® Windows®.

**Note:** On UNIX™ platforms, pathname expansion is part of the shell interpreter. File names are resolved before Genero commands get the specified argument(s). In order to use the Genero build-in pathname expansion on a Unix platform, specify the search pattern in single quotes:

```text
fglcomp '^[a-z]*.4gl'
```

For more details about supported patterns and wildcards, see the `os.Path.glob` on page 2801 utility method.

See also **Providing the source files in an arguments file** on page 2111

### Automatic compilation of imported modules

When compiling a `.4gl` module that imports other modules with the `IMPORT FGL` instruction, `fglcomp` will automatically compile the imported modules, if they are located in the same directory of the current module, and if the `.4gl` source is more recent as the `.42m` file.

For more details, see **IMPORT FGL module** on page 481.

### Stricter compilation options

By default, `fglcomp` allows you to compile legacy application sources in a relaxed manner. For example, it is possible to compile a module calling a function which is not defined in that module, and resolved during the link process.

In recent developments based on features such as `IMPORT FGL`, consider using `fglcomp` options that make the compilation stricter:

- `--resolve-calls`: To check that all functions used by a module are defined locally, or by imported modules.
- `--Wall`: To print all type of warnings.
- `--Werror`: To stop compilation if a warning occurs.
- `--M`: To write error messages to standard output instead of producing a `.err` file.

For more details about `fglcomp` options, see the `fglcomp command reference topic`.

Additionally, the `linker` provides options such as `--print-imports` and `--print-missing-imports`, that produce useful information to use new language features.

### Compiling in make mode

The `fglcomp` compiler supports the `--make` option, to compile all provided sources, in the way the `make` utility does: If the `.42m` file is older than the `.4gl` file, it will be recompiled. If the `.42m` is up-to-date with the `.4gl`, it is not recompiled.
As with a regular compilation (not using the --make option), for a provided .4gl module, imported modules will be automatically compiled when needed.

To test the compilation process and dependencies without actually compiling the modules, add the --simulate option, to see what modules would be compiled. The --simulate option can also be used without the --make option: fglcomp will then print the modules that would be compiled, even if the .42m file is up to date.

**Tip:** fglcomp produces the .42m files in the current working directory. Consider creating a bin directory, to hold all .42m files, and run the compiler in this directory by providing relative paths of the .4gl sources.

For example:

```
$ tree .
.
  -- bin
  -- dir1
    -- module1.4gl
    -- module2.4gl
    `-- module3.4gl
  -- progl.4gl
  `-- prog2.4gl
2 directories, 5 files
$ head *.4gl */*.4gl
===> prog1.4gl <==
IMPORT FGL module1
MAIN
  CALL module1.function1()
END MAIN

===> prog2.4gl <==
IMPORT FGL module2
MAIN
  CALL module2.function2()
END MAIN

===> dir1/module1.4gl <==
FUNCTION function1()
END FUNCTION

===> dir1/module2.4gl <==
IMPORT FGL module3
FUNCTION function2()
  CALL module3.function3()
END FUNCTION

===> dir1/module3.4gl <==
FUNCTION function3()
END FUNCTION
```

Go to the bin directory:

```
$ cd bin
```

Assuming there are no .42m in this directory tree, first check what would be compiled if we pass all .4gl sources to fglcomp:

```
$ fglcomp --make --simulate ../*.4gl ../*.4gl
[fglcomp ../dir1/module1.4gl usedByCount:1]
[fglcomp ../progl.4gl usedByCount:0]
[fglcomp ../dir1/module3.4gl usedByCount:1]
[fglcomp ../dir1/module2.4gl usedByCount:1]
```
Now compile (same output as simulated previously): fglcomp produces the .42m files:

```
$ fglcomp --make *.4gl dir1/*.4gl
[fglcomp ../dir1/module1.4gl usedByCount:1]
[fglcomp ../dir1/module3.4gl usedByCount:1]
[fglcomp ../dir1/module2.4gl usedByCount:1]
[fglcomp ../prog2.4gl usedByCount:0]

$ ls
module1.42m module2.42m module3.42m prog1.42m prog2.42m
```

Touch some modules and see which one are recompiled, based on the IMPORT FGL dependence tree:

```
$ sleep 1 && touch ../dir1/module3.4gl ../prog1.4gl
$ fglcomp --make ../*.4gl ../*/*.4gl
[fglcomp ../prog1.4gl usedByCount:0]
[fglcomp ../dir1/module3.4gl usedByCount:1]
[fglcomp ../dir1/module2.4gl usedByCount:1]
[fglcomp ../prog2.4gl usedByCount:0]
```

### Producing make-style dependency rules

The `fglcomp --dependencies` option can be used to produce makefile-style dependency rules for imported modules. In the next example, the `main.4gl` source imports `module1` and thus the `main.42m` compilation depends from `module1.42m`:

```
$ head *.4gl
==> main.4gl ===
IMPORT FGL module1

MAIN
    CALL module1.function1()
END MAIN

==> module1.4gl ===
FUNCTION function1()
END FUNCTION
```

Circular imports are handled with proper dependency rules that can even be used for parallel builds (with `make -j`).

**Tip:** Try to perform the command `fglcomp --dependencies *.4gl` on all your sources and see the output.

### Providing the source files in an arguments file

The `fglcomp` compiler supports the `@argfile` argument, to provide a file that contains the list of source files to be compiled. This can be used when it is not possible to pass all source files in the command line.

Only source files must be specified in the arguments file. Compiler options must be provided in the command line.

The argument file must contain one source file per line:

```
$ cat mysources.txt
module1.4gl
```
Note that the arguments file can contain expansion patterns as in the `fglcomp` command line:

```
$ cat mysources.txt
module[123]*.4gl
$ fglcomp -M -Wall @mysources.txt
```

### Handling fglcomp compiler errors

If an error occurs, the compiler writes an error file by default with the `.err` extension.

```
$ cat prog.4gl
MAIN
 LET x = "hello"
END MAIN

$ fglcomp prog.4gl
Compilation was not successful. Errors found: 1.
The file prog.4gl has been written.

$ cat prog.err
MAIN
 LET x = "hello"
 | The symbol 'x' does not represent a defined variable.
 | See error number -4369.
END MAIN
```

With the `-M` option, you can force the compiler to display an error message instead of generating an `.err` error file:

```
$ fglcomp prog.4gl
xx.4gl:2:8 error:(-4369) The symbol 'x' does not represent a defined variable.
```

### Produce compiler warnings with `-W`

To improve code quality, enable compiler warnings with the `-W` option:

```
$ cat prog.4gl
MAIN
 DATABASE test1
 SELECT COUNT(*) FROM x, OUTER(y) WHERE x.k = y.k
END MAIN

$ fglcomp -W stdsql prog.4gl
xx.4gl:3: warning: SQL statement or language instruction with specific SQL syntax.
```

When a warning is raised, you can use the `-W error` option to force the compiler to stop as if an error was found.

Some warnings are generated by default (without using the `-W` option), when the source code uses a feature that is considered as "fragile" yet to be supported for backward compatibility.

By default warnings go to the `stderr` stream. When creating a `.err` file, warnings can be redirected to the `.err` file with the `-W to-err-file` option.

For the complete list of warning options, see `-W option` in `fglcomp` command reference.
Related concepts
Importing modules on page 480
Use the IMPORT ... instruction to import BDL, C or Java external modules in the current module.

Importing modules
Describes how to define module interdependence with IMPORT FGL.

With the IMPORT FGL instruction, module symbols such as variables, types and constants can be referenced in the importing module.

The following source example imports the myutils and account modules, and uses the init() and set_account() functions of the imported modules.

The first function call is qualified with the module name - this is optional but required to resolve ambiguities when the same function name is used by different modules:

```
IMPORT FGL myutils
IMPORT FGL account
MAIN
   CALL myutils.init()
   CALL set_account("CFX4559")
   ...
END MAIN
```

For more details, see IMPORT FGL module on page 481.

Linking libraries
Describes how to link .42m modules together to build a .42x library file.

Grouping .42m modules in .42x libraries
Compiled .42m modules can be grouped in libraries using the fgllink on page 2076 linker. The library file gets the .42x extension.

The linker can be used to create .42x libraries or .42r program files. If none of the modules provided to the linker defines the MAIN block, the linker creates a library file; if a MAIN block is present, the linker creates a program file. Make sure to use the correct file extension.

Note: Linking is supported for backward compatibility, it is recommended that you use IMPORT FGL instead.

Library linking is done with the fglrun tool by using the -l option. The fgllink tool can be used for convenience, it is a simple script calling fglrun -l.

The following lines show a link procedure to create a library in a UNIX™ shell session:

```
$ fglcomp fileutils.4gl
$ fglcomp userutils.4gl
$ fgllink -o libutils.42x fileutils.42m userutils.42m
```

When you create a library, all functions of the .42m modules used in the link command are registered in the .42x file.

Important: The .42x library file does not contain the .42m p-code. When deploying your application, you must provide all compiled .42m modules.

When creating a .42x library, all functions must be uniquely defined; otherwise, error -6203 will be returned by the linker.
Providing the files to link in an arguments file

The fgllink linker supports the @argfile argument, to provide a file that contains the list of .42m modules and .42x libraries to be used for the link. This can be used when it is not possible to pass all files in the command line.

Only link files must be specified in the arguments file. Linker options must be provided in the command line.

The argument file must contain one file per line:

```
$ cat myfiles.txt
module1.42m
module2.42m
$ fgllink -o lib1.42x @myfiles.txt
```

Using libraries when linking programs

The 42x libraries are typically used to link the final 42r programs:

```
$ fglcomp mymain.4gl
$ fgllink -o myprog.42r mymain.42m libutils.42x
```

The 42r programs must be re-linked, if the content of the 42x libraries changes.

In this example, if a function of the userutils.4gl source file was removed, you must recompile userutils.4gl, re-link the libutils.42x library and re-link the myprog.42r program.

Linking libraries with other libraries

It is possible to create a library by referencing other 42x library files in the link command, as long as 42m modules can be found:

```
$ fglcomp module_1.4gl
$ fglcomp module_2.4gl
$ fgllink -o lib_A.42x module_1.42m
$ fgllink -o lib_B.42x module_2.42m lib_A.42x
$ fgllink -o myprog.42r lib_B.42x
-- will hold functions of module_1 and module_2.
```

P-Code module find path FGLLDPATH

If you do not specify an absolute path for a file, the linker searches by default for 42m modules and 42x libraries in the current directory.

If the 42m modules are not in the current directory, you can specify the 42m module search path with the FGLLDPATH environment variable.

Linking libraries when using C Extensions

If you are using C-Extensions, you may need to use the -e option to specify the list of extension modules, if the IMPORT keyword is not used:

```
$ fgllink -e extlib,extlib2,extlib3 -o libutils.42x fileutils.42m userutils.42m
```

Related concepts

Compiling program code files (.4gl) on page 2108
The .4gl source files must be compiled to .42m p-code files, in order to be loaded by the runtime system.

Linking programs

Describes how to link .42m modules together to build a .42r program file.

Purpose of .42r program files

Traditional Genero programming allows you to call a function without seeing the definition of that function. The goal of linking programs is to resolve function symbols and build a .42r program file.

Note: When writing new applications, consider using IMPORT FGL instead of traditional linking. When using IMPORT FGL, the link stage is no longer required, and the 42m module containing the MAIN block can be directly executed.

Genero .42r program files are created by linking several .42m modules and/or .42x libraries together, where one of the modules defines a MAIN block.

By convention, the resulting program file gets the .42r extension.

Program linking is done with the fglrun tool by using the -l option. The fgllink tool can be used for convenience; it is a simple script calling fglrun -l.

The following lines show a link procedure to create a program in a UNIX™ shell session:

```
$ fglcomp main.4gl
$ fglcomp store.4gl
$ fgllink -o stores.42r main.42m store.42m
```

Important: If you omit the -o option in the fgllink command, the default output file will get the .42x extension (used for libraries), with the name of the module containing the MAIN block. The .42r file extension is used by convention, to distinguish a program dictionary file from a library dictionary file.

Providing the files to link in an arguments file

The fglrun/fgllink linker supports the @argfile argument, to provide a file that contains the list of .42m modules and .42x libraries to be used for the link. This can be used when it is not possible to pass all files in the command line.

Only link files must be specified in the arguments file. Linker options must be provided in the command line.

The argument file must contain one file per line:

```
$ cat myfiles.txt
main.42m
module1.42m
module2.42m
lib1.42x
$ fgllink -o prog.42x @myfiles.txt
```

Symbol resolution with the linker

The purpose of the linking phase is to check for missing function symbols, and reference all the symbols in the resulting .42r program file.

Note: The IMPORT FGL method is the preferred method for linking. However, you can mix both methods. The job of the linker is to resolve symbols that are not already resolved by the compiler from IMPORT FGL usage.

Any function used in the .42m modules specified in the link line must be provided. Missing symbols will result in a -1338 linker error:

```
$ cat main.4gl
MAIN
```
CALL myfunc()
END MAIN

$ fglcomp main.4gl
$ fgllink -o prog.42r main.42m
ERROR(-1338):The function 'myfunc' has not been defined in any module in the program.

**Note:** Symbol resolution in only be done when linking programs. When linking a 42x library, there can be references to undefined functions.

When linking a 42r program, global symbols must be unique; otherwise, error -6203 will be returned by the linker. The same error will be returned when linking a 42x library by using modules defining the same functions.

The link process searches recursively for the functions used by the program. For example, if the MAIN block calls function FA in module MA, and FA calls FB in module MB, all functions from module MA and MB will be included in the 42r program definition.

**The linking process steps in detail**

When linking a .42r program (with modules possibly using IMPORT FGL), the linker works as follows:

1. Loads all modules specified on the command-line. Builds a symbol-table of all public functions. Raises error -6203, if a public function is defined more than once.
2. Loads all imported modules (if not specified on the command line). Adds public functions to the symbol-table.
   a. If an imported function is already defined by a linked module, the imported function will be ignored.
   b. If an imported function is not defined by a linked module, and is defined more than once, then this function is marked as “ambiguous” (this is legal, no warning or error is produced).
3. All unresolved symbols on any module (linked modules and imported modules) will be resolved.
   • If a function is not defined, error -1338 is thrown.
   • If a function is an ambiguous function (2.b), error -8401 is thrown.

**Note:**
Understand the difference in case of error -6203 and -8401:

• Error -6203: A function in a linked module is defined more then once. That's illegal.
• Error -8401: A function in an imported module is define more then once, and used by the linker to resolve a symbol. It's legal to define an imported function more than once, but this function can not be used by the linker.

**Content of a .42r program file**

The generated 42r program files do not contain the 42m p-code, it is basically a dictionary of global symbols used by the program.

When deploying an application, both 42m modules and 42r program files must be provided.

**Note:** Since 42x library files are only used to build programs, you do not have to deploy 42x library files.

When linking a 42r program by using 42x libraries, the modules defined in a library are included only if one of the symbols in the module is used by the program. However, all symbols of 42m modules specified in the command line will always be referenced in the resulting 42r program file.

**Note:** During the link, when the same function symbols are defined in distinct libraries, the linker will select the function of the first library that was specified in the command line.

All symbols referenced in a module must exist in the final 42r program dictionary file. If a symbol is not found, the runtime system stops with error -1338. This error is fatal and cannot be trapped with an exception handler.
P-Code module find path FGLLDPATH

If you do not specify an absolute path for a file, the linker searches by default for 42m modules and 42x libraries in the current directory.

You can specify the 42m search path with the FGLLDPATH environment variable:

```
$ FGLLDPATH=/usr/dev/lib/maths:/usr/dev/lib/utils
$ export FGLLDPATH
$ ls /usr/dev/lib/maths
mathlib1.42x
mathlib2.42x
mathmodule11.42m
mathmodule12.42m
mathmodule22.42m
$ ls /usr/dev/lib/utils
fileutils.42m
userutils.42m
dbutils.42m
$ fgllink -o myprog.42r mymodule.42m mathlib1.42x fileutils.42m
```

In this example the linker will find the specified files in the /usr/dev/lib/maths and /usr/dev/lib/utils directories defined in FGLLDPATH.

**Library versus module function precedence**

When creating a .42r program by linking .42m modules with .42x libraries, if the same function is defined in a 42m and in a module of a 42x library, the function of the specified 42m module will be selected by the linker, and the function of the library will be ignored.

However, the linker will raise error -6203, if two 42m modules specified in the link command define the same function.

**Exclusion of unused library module**

When linking a .42r program by using a .42x library, if none of the functions of a module in the .42x library are used in the program, the complete module is excluded by the linker.

**Note:** Unused module exclusion may cause undefined function errors at runtime, such as when a function is only used in a dynamic call (an initialization function, for example.)

The following case illustrates this behavior:

```
$ cat module1.4gl
FUNCTION func11()
END FUNCTION

$ cat module2.4gl
FUNCTION func21()
END FUNCTION

$ cat main.4gl
MAIN
  CALL func11()
END MAIN

$ fgllink module1.4gl
$ fgllink module2.4gl
$ fgllink main.4gl
$ fgllink -o lib.42x module1.42m module2.42m
```
Here, module x1.42m (with functions fx11 and fx12) will be referenced in the .42r program file, but functions of module x2.42m will not. At runtime, any dynamic call to functions fx21() or fx22() will fail with an untrappable error -1338.

Symbol conflicts with IMPORT FGL and linking

The job of the linker is to resolve symbols that are not already solved by the compiler from IMPORT FGL usage. If the same function is defined by a linked module and an imported module, and the function is called without the module prefix, the imported function takes precedence over the linked module function.

The following example illustrates this case:

```
$ cat module1.4gl
FUNCTION func11()
  DISPLAY "module1.func11()"
END FUNCTION

$ cat module2.4gl
IMPORT FGL module1
FUNCTION func21()
  DISPLAY "module2.func21()"
  CALL func11() -- from module1, because it is imported
END FUNCTION

$ cat module3.4gl
FUNCTION func11() -- Same name as in module1.4gl
  DISPLAY "module3.func11()"
END FUNCTION

$ cat main.4gl
MAIN
  CALL func11() -- from module3, because it is linked
  CALL func21()
END MAIN

$ fglcomp module1.4gl
$ fglcomp module2.4gl
$ fglcomp module3.4gl
$ fglcomp main.4gl
$ fgllink -o prog.42r main.42m module2.42m module3.42m
$ fgllrun prog.42r
module3.func11()
module2.func21()
module1.func11()
```

Note: To make your code more readable, consider prefixing imported symbols with the module name and thus avoid any ambiguity.

Imported module using linked functions

When linking a program with modules using the IMPORT FGL instruction, the imported modules do not have to be specified in the link line.

However, if the imported module uses functions that come from other modules, which are not imported by this module, these non-imported modules must be specified in the link command line.
For example, if the main module imports module module1 to call the func11(), which in turn calls func21() from module2, but module1 does not import module2, then module2 must be linked to the program:

```
$ cat module1.4gl
FUNCTION func11()
   DISPLAY "module1.func11()"
   CALL func21()
END FUNCTION

$ cat module2.4gl
FUNCTION func21()
   DISPLAY "module2.func21()"
END FUNCTION

$ cat main.4gl
IMPORT FGL module1
MAIN
   CALL module1.func11()
END MAIN

$ fglcomp module1.42m
$ fglcomp module2.42m
$ fglcomp main.4gl
$ fglrun main.42m
module1.func11()
Program stopped at 'module1.4gl', line number 3.
FORMS statement error number -1338.
The function 'func21' has not been defined in any module in the program.

$ fgllink -o prog.42r main.42m module2.42m

$ fglrun prog.42r
module1.func11()
module2.func21()
```

### Linking programs with C Extensions

If you are using C-Extensions, you may need to use the -e option to specify the list of extension modules if the IMPORT keyword is not used:

```
$ fgllink -e extlib,extlib2,extlib3 -o stores.42r main.42m store.42m
```

### Related concepts

- [Linking programs using C-Extensions](#)

When creating a 42r program or 42x library, the linker needs to resolve all function names, including C-Extension functions.

### Using makefiles

Describes how to define program construction rules in makefiles.

Most UNIX™ platforms provide the make utility program to compile projects. The make program is an interpreter of makefiles. These files contain directives to compile and link programs and/or generate other kind of files.

When developing on Microsoft™ Windows® platforms, you may use the NMAKE utility provided with Visual C++. However, this tool does not have the same behavior as the UNIX™ make program. To have a compatible make on Windows®, you can install a GNU make or third party UNIX™ tools such as Cygwin.

For more details about the make utility, see the platform-specific documentation.
The following example shows a typical makefile for Genero applications:

```bash
# Generic makefile rules to be included in Makefiles
.SUFFIXES: .42s .42f .42m .42r .str .per .4gl .msg .hlp
FGLFORM=fglform -M
FGLCOMP=fglcomp -M
FGLLINK=fglrun -l
FGLMKMSG=fglmkmsg
FGLMKSTR=fglmkstr
FGLLIB=$$FGLDIR/lib/libfgl4js.42x
all::
  .msg.hlp:
    $(FGLMKMSG) $*.msg $*.hlp
  .str.42s:
    $(FGLMKSTR) $*.str $*.42s
  .per.42f:
    $(FGLFORM) $*.per
  .4gl.42m:
    $(FGLCOMP) $*.4gl
clean::
  rm -f *.hlp *.42? *.out
#---------------------
# Makefile example
include Makeincl
FORMS=\customers.42f\orderlist.42f\itemlist.42f
MODULES=\customerInput.42m\zoomOrders.42m\zoomItems.42m
customer.42x: $(MODULES)
  $(FGLLINK) -o customer.42x $(MODULES)
all:: customer.42x $(FORMS)
```

### 42m module information
Describes how to handle module information in .42m p-code files.

**Compiler and runtime compatibility**

The runtime system (fglrun) used to execute programs must be compatible with the fglcomp compiler version used to build the .42m modules.

To control this compatibility, the compiler writes Genero and p-code version information in the generated 42m files, in the p-code header.

The compiler also writes additional information in the p-code header, such as the source file name used at compilation, a timestamp, or a user-defined string/tag.

The p-code header information of a 42m file can be read with the fglrun -b module.42m command. This is useful on production sites, to check the version of the compiler that was used to build the 42m modules.

It is also possible to get additional information about the p-code module such as the size used in memory with fglrun --module-size.

**Identifying the version of the compiler and runtime system**

To check if the version of the compiler or runtime system, use the -V option:

```
$ fglcomp -V
```
### Extracting p-code header information

To extract build information from a `.42m` file, run `fglrun` with the `-b` option:

```
$ fglrun -b mymodule.42m
3.10.12 /home/devel/stores/mymodule.4gl 24
```

The p-code header contains the following fields:

1. An optional timestamp, if the `--timestamp` option was used with `fglcomp`.
2. The Genero product version.
3. The full path of the source file, or the module name, if the `--omit-source-name` option was used with `fglcomp`.
4. The internal identifier of the p-code version.
5. An optional user-defined tag, if the `--tag=string` option was used with `fglcomp`.

### Avoiding the full source file path in the p-code header

By default, `fglcomp` writes the source file name (full path) in the resulting `.42m` module:

```
$ fglcomp mymodule.4gl
$ fglrun -b mymodule.42m
3.10.12 /home/devel/stores/mymodule.4gl 24
```

To avoid the source name in the `.42m` module, use the `--omit-source-name` option of `fglcomp`, to write only the module name in the p-code header:

```
$ fglcomp --omit-source-name mymodule.4gl
$ fglrun -b mymodule.42m
3.10.12 mymodule 24
```

### Writing a custom string to the p-code header

Use the `--tag="custom-string"` option of `fglcomp`, to add a user-defined string to the p-code header. When reading p-code header information with `fglrun -b`, the user-defined string is printed as `tag="custom-string"`.

The tag field can for example be used to stamp the 42m file with the custom product version:

```
$ fglcomp --tag="OXOGEN 5.23" mymodule.4gl
$ fglrun -b mymodule.42m
3.10.12 /home/devel/stores/mymodule.4gl 24 tag="OXOGEN 5.23"
```
Writing a compilation timestamp to the p-code header

To write timestamp information in the p-code header, use the `--timestamp` option of `fglcomp`:

```
$ fglcomp --timestamp mymodule.4gl
$ fglrun -b mymodule.42m
2008-12-24 11:22:33 3.10.12 /home/devel/stores/mymodule.4gl 24
```

**Important:** When using the `--timestamp` compiler option to write build timestamp information in p-code modules, you will not be able to easily compare 42m files (based on a checksum, for example). Without the timestamp, `fglcomp` generates exactly the same p-code module as if the source file was not modified.

Reading p-code header of older versions

`fglrun` can read the header of p-code modules compiled with older versions of `fglcomp` and display version information for such old modules.

If `fglrun` cannot recognize a p-code module, it returns an execution status that is different from zero.

Reading p-code header of 42x and 42r files

When reading build information of a 42x or 42r file, `fglrun` scans all modules used to build the library or program. You will see different versions in the first column if the modules were compiled with different versions of `fglcomp`. However, it's not recommended that you mix versions on a production site:

```
$ fglrun -b myprogram.42r
3.10.11 /home/devel/stores/mymodule1.4gl 24
3.10.02 /home/devel/stores/mymodule2.4gl 24
3.10.12 /home/devel/stores/mymodule3.4gl 24
```

Computing the p-code size of a module or program

The `fglrun` command provides the `--module-size` option to compute the p-code size of a module:

```
$ fglrun --module-size mymodule.42m
12.34K mymodule
```

This size is defined by the amount of p-code instructions of the module.

**Note:** P-code modules are shared by all `fglrun` instances executing on the same computer. For more details, see [Elements shared by multiple programs](#) on page 586.

The `--program-size` option reports the total p-code size of all modules of a program. The argument must be a `.42m` module containing the MAIN (with main module importing other modules by `IMPORT FGL`):

```
$ fglrun --module-size myprogram.42m
0.34K module1
2.59K module2
...
12.76K total
```

The `--program-size` option can also use a `.42r` program file as argument, for linked programs:

```
$ fglrun --module-size myprogram.42r
0.34K module1
2.59K module2
...
12.76K total
```
Source code edition

Simple helper to better render sources in configurable text editors.

These topics concern source code editing. You are free to use your preferred source code editor to write your programs.

Choosing the correct locale

Before starting to edit source files, you must identify and configure the editor with the locale (character set) you want to use in your sources.

The language supports single-byte and multibyte character sets. When developing multilingual applications, we recommend that you write .per and .4gl source files in ASCII, and externalize language-dependent messages in string resource files.

Related concepts

Localization on page 512
Localization support allows you to implement programs that follow specific language and cultural rules.

Avoid Tab characters in screen layouts

When editing .per form files, avoid using Tab characters in sources, especially in the LAYOUT or SCREEN sections of forms. Different kinds of text or source code editors can expand Tab characters differently, depending on the configuration settings. As a result, if two programmers are using different Tab expansion settings, the form layout will display in different ways. If used in a grid area, a Tab character will be interpreted as 8 blanks by fglform.

It is legal to use Tab characters in the rest of the .per file or .4gl sources (for example, to indent the code).

Configure VIM for Genero BDL

The VIM editor

VIM is a well-known source code editor for programmers.

Automatic code completion, syntax highlighting and code formatting/indentation is supported by fglcomp and fglform compilers, when using VIM.

Important: In order to use Genero code completion with VIM, you need at least VIM version 7 with the Omni Completion feature.

Configuring VIM for Genero BDL

Perform the following steps to enable code-completion for Genero in VIM:

1. Locate the VIM resource file for your operating system user:
   - On UNIX® platforms, the VIM resource file is ~/.vimrc.
   - On Windows® platforms, the VIM resource file is %USERPROFILE%\_vimrc.

2. Add the following lines to make VIM find Genero VIM files in $FGLDIR/vimfiles:

   ```vim
   let generofiles=expand($FGLDIR . "\vimfiles")
   if isdirectory(generofiles)
     let &rtp=generofiles.'\',.'&rtp
   endif
   ...
   ```

3. Add the next line to enable syntax highlighting in VIM:

   ```vim
   syntax on
   ```
4. Add the following lines to associate source file extensions to the corresponding syntax definition file:

   autocmd BufNewFile,BufRead *.per setlocal filetype=per

   **Note:** In fact, the .4gl file type "fgl" is usually detected in the default configuration files of VIM (like /usr/share/vim/vim80/filetype.vim). If this file type is not detected by default by your VIM installation, add the next line in your .vimrc file:

   autocmd BufNewFile,BufRead *.4gl setlocal filetype=fgl

5. Disable case sensitivity for keywords by adding the following line:

   let fgl_ignore_case=1

   **Note:** Genero BDL is not case sensitive and allows to use language keywords for identifiers (DEFINE name STRING). Common style guidelines and the default VIM syntax highlighting encourage the use of uppercase keywords. This avoids highlighting of keywords used as identifiers. Case sensitive highlighting can be disabled by setting fgl_ignore_case to 1 in the VIM resource file. When using this option, symbols like variable names matching language keywords (define name string) will be highlighted.

6. To enable lowercase keywords in code completion proposals, add the following line:

   let fgl_lowercase_keywords=1

   **Note:** Setting fgl_lowercase_keywords=1 implies implicitly fgl_ignore_case=1.

7. Add the following lines to define function keys and commands that reformat the code using fglcomp --format:

   - F2: formats the whole source file
   - F3: formats lines changes since last git commit (requires a git repository)
   - F4: in command mode: formats a range, in edit mode: formats the current line.
   - The command Fo can also be used in this form:

     :'a,.Fo

     Format all lines from mark a to the current line

   autocmd BufNewFile,BufRead *.4gl map <F2> :call FglFormatAll()<CR>
   autocmd BufNewFile,BufRead *.4gl map <F3> :call FglGitFormat()<CR>
   autocmd BufNewFile,BufRead *.4gl map <F4> :call FglFormatRange()<CR>
   autocmd BufNewFile,BufRead *.4gl imap <F4> <c-o>:call FglFormatRange()<CR>
   autocmd BufNewFile,BufRead *.4gl command! -range Fo <line1>,<line2>call FglFormatRange()

   function! FglFormatAll()
     let l = line('.')
     silent! :%!fglcomp --format stdin:%
     if v:shell_error
       undo
     endif
     call cursor(l,1)
   endfun

   function! FglFormatRange() range
     let l = line('.')
     silent! :exe '%!fglcomp --format --fo-lines " . a:firstline.":%a:lastline. " stdin:%'
     if v:shell_error
       undo
     endif
     call cursor(l,1)
8. If you want to reformat the source code with `fglcomp --format` each time you save the file, add the following lines:

```vim
endfun

function! FglGitFormat()
    let l = line('.')
    silent! :w
    let s = system("fglgitformat -i ".expand('%'))
    silent! :e
    call cursor(l,1)
endfun
```

**Note:** The auto-formatting may not be done, if the source code contains errors. In this case the source file is left untouched, but it will be saved.

### Using VIM on Microsoft Windows platforms

On Windows® platforms, you typically install the GVim (Graphical VIM) software.

**Note:** Some versions of VIM for Windows® may use different configuration files and locations (`_vimrc` or `.vimrc`?). Refer to the VIM documentation, to make sure that you use the proper files.

When using the command line version VIM, you may want to add:

```vim
color shine
```

### Using code completion with Genero and VIM

First make sure the Genero environment is set (FGLDIR, PATH).

Open a `.4gl` or `.per` file, start to edit the file with VIM.

**Note:** On Windows® platforms, if you start GVim from the icon, the Genero environment may not be set. As result `fglcomp/fglform` cannot be called from VIM. You need to set the Genero BDL environment before starting VIM.

When in insert mode, press `CTRL-X + CTRL-O`, to get a list of language elements to complete the instruction syntax or expression.

For convenience, `TAB` can also be used to get the completion list as with the `CTRL-X + CTRL-O` key combinations. However, `TAB` will only show the completion list, if the edit cursor is after a keyword. At the beginning of the line, `TAB` adds indentation characters.

Use the `F2` function key to indent/beautify the current `.4gl` source code, with the `fglcomp --format` beautifier tool.

For more details about VIM, see [http://www.vim.org](http://www.vim.org).
Source documentation

Explains how to automatically generate documentation from your sources.

Understanding source code documentation

This is an introduction to source code documentation.

Documenting sources is an important task in software development, to share the code among applications and achieve better re-usability.

Source documentation must be concise, clear, and complete. However, documenting sources can be boring and subject to mistakes if large repetitive documentation sections have to be written by hand.

Source documentation can be produced automatically with the fglcomp compiler. The compiler can generate source documentation from the .4gl files of your project with minimum effort. The resulting source documentation is generated in simple HTML format and can be published on a web server.

Source documentation is generated with the --build-doc option of fglcomp. To extract documentation from a .4gl source:

```shell
fglcomp --build-doc filename.4gl
```

You can generate default documentation from the existing sources. For a better description of the code, add special #+ comments in your sources to describe code elements such as functions, function parameters, and return values.

By default, only PUBLIC symbols are documented. If you want to include PRIVATE symbols, use the --doc-private option:

```shell
fglcomp --build-doc --doc-private filename.4gl
```

Prerequisites to produce documentation

This topic lists the requirements to generate source documentation.

To generate the HTML pages, fglcomp first generates .xa files which must be converted to .html files. The conversion from .xa to .html is done with an XSLT processor using the .xsl style sheet files provided in $FGLDIR/lib/fgldoc/

You must have an XSLT processor installed on the machine where the documentation is generated.

- **On UNIX™**, fglcomp runs the $FGLDIR/lib/fgldoc/Transform.sh script to convert .xa files to .html files. Therefore you need the xsltproc command line XSLT processor (from the libxml package).
- **On Windows®**, fglcomp runs the $FGLDIR\lib\fgldoc\Transform.js script to convert .xa files to .html files. To run the Transform.js script, you must have cscript.exe installed with the Microsoft™.XMLDOM class (this is the case on recent Windows® versions).

**Note:** If the default result of the transformation does not fit your needs, the style sheets provided in $FGLDIR/lib/fgldoc can be adapted to generate different HTML files.

Source documentation structure

The source documentation structure is based on the well-known Java-doc technique.

The generated documentation reflects the structure of your sources; in order to have nicely structured source documentation, you must have a nicely structured source tree.

The source documentation elements are structured as follows (elements in italic must be created by hand, others are generated files):

- **Top/root directory (the root of your project)**
  - `overview.4gl` (description of the project)
First create a file named `overview.4gl` in the top directory of the project. This file contains the overall description of the project. In that directory, the documentation generator creates the files `overview-summary.html`, `overview-frame.html`, `allclasses-frame.html`, `index-all.html`, `index.html` and `fgldoc.css`.

The documentation generator can scan sub-directories to build the documentation for a whole project; each source directory defines a package. For each directory (package), the generator creates a `package-summary.html` and a `package-frame.html` file. If a file with the name `package-info.4gl` exists, it will be scanned to complete the `package-summary.html` file with the package description.

The documentation generator creates a `filename.html` file for each `.4gl` source module, seen as a class in the documentation.

### Adding comments to sources

Adding relevant information to your source documentation.

#### Commenting a module constant

To document a module constant definition, add `#+` lines just before the `CONSTANT` declaration.
The comment body is composed of paragraphs separated by blank lines. The first paragraph of the comment is a short description of the constant. This description will be placed in the constant summary table. The next paragraph is long text describing the constant in detail. Other paragraphs must start with a tag to identify the type of paragraph; a tag starts with @ (the “at” sign).

**Table 450: Supported @ tags**

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@code</td>
<td>Indicates that the next lines show a code example using the constant.</td>
</tr>
</tbody>
</table>

**Example**

```plaintext
#+ This is the constant Pi
#+ To be used in trigo computing
#+ @code
#+ DISPLAY util.Math.cos( Pi / 2 )
#+
PUBLIC CONSTANT Pi = 3.14159
```

**Commenting a module variable**

To document a module variable declaration, add #: lines just before the DEFINE declaration.

The comment body is composed of paragraphs separated by blank lines. The first paragraph of the comment is a short description of the variable. This description will be placed in the variable summary table. The next paragraph is long text describing the variable in detail. Other paragraphs must start with a tag to identify the type of paragraph; a tag starts with the @ “at” sign.

**Table 451: Supported @ tags**

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@code</td>
<td>Indicates that the next lines show a code example using the variable.</td>
</tr>
</tbody>
</table>

**Example**

```plaintext
#+ Customer array
#+ Fill this array with customer records and use it
#+ in DISPLAY ARRAY to control a TABLE
#+
PUBLIC DEFINE cust_list DYNAMIC ARRAY OF RECORD
cust_num INTEGER,
cust_name VARCHAR(50),
cust_addr VARCHAR(100)
END RECORD
```

**Commenting a module type**

To document a module type definition, add #: lines just before the TYPE declaration.

The comment body is composed of paragraphs separated by blank lines. The first paragraph of the comment is a short description of the type. This description will be placed in the type summary table. The next paragraph is long text describing the type in detail. Other paragraphs must start with a tag to identify the type of paragraph; a tag starts with the @ “at” sign.
Table 452: Supported @ tags

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@code</td>
<td>Indicates that the next lines show a code example using the type.</td>
</tr>
</tbody>
</table>

Example

```lisp
#+ This is the customer type
#+
#+ Define variables with this type to hold customer records.
#+
#+ @code
#+ DEFINE myvar t_cust
#+
PUBLIC TYPE t_cust RECORD
  cust_num INTEGER,
  cust_name VARCHAR(50),
  cust_addr VARCHAR(100)
END RECORD
```

Commenting a function

To document a function, add some lines starting with `#+`, before the `FUNCTION` declaration.

The comment body is composed of paragraphs separated by blank lines. The first paragraph of the comment is a short description of the function. This description will be placed in the function summary table. The next paragraph is long text describing the function in detail. Other paragraphs must start with a tag to identify the type of paragraph; a tag starts with the `@` "at" sign.

Table 453: Supported @ tags

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@code</td>
<td>Indicates that the next lines show a code example using the function.</td>
</tr>
</tbody>
</table>
| @param name description   | Defines a function parameter identified by `name`, explained by a `description`.
|                           | `name` must match the parameter name in the function declaration.           |
| @returnType data-type     | Defines the data type of the value returned by the function.                |
|                           | If the function returns several values, write a comma-separated list of types.|
|                           | **Note:** The `@returnType` tag does not need to be used when the `FUNCTION` is defined with a `RETURNS` clause. |
| @return description       | Describes the values returned by the function.                              |
|                           | Several `@return` comment lines can be written.                             |

Example using a FUNCTION definition with signature

```lisp
#+ Compute the amount of the orders for a given customer
#+
```
This function calculates the total amount of all orders for the customer identified by the cust_id number passed as parameter.

@code
DEFINE total DECIMAL(10,2)
LET total = ordersTotal(r_customer.cust_id)

@param cid Customer identifier
@return The total amount as DECIMAL(10,2)

FUNCTION ordersTotal(cid INTEGER) RETURNS DECIMAL(10,2)
DEFINE ordtot DECIMAL(10,2)
SELECT SUM(ord_amount) INTO ordtot
   FROM orders WHERE orders.cust_id = cid
RETURN ordtot
END FUNCTION

Example using a FUNCTION definition without signature (needs @returnType)

@code
DEFINE total DECIMAL(10,2)
LET total = ordersTotal(r_customer.cust_id)

@param cid Customer identifier
@returnType DECIMAL(10,2)
@return The total amount as DECIMAL(10,2)

FUNCTION ordersTotal(cid)
DEFINE cid INTEGER
DEFINE ordtot DECIMAL(10,2)
SELECT SUM(ord_amount) INTO ordtot
   FROM orders WHERE orders.cust_id = cid
RETURN ordtot
END FUNCTION

Commenting a report

To document a report, add some lines starting with `#+`, before the `REPORT` declaration.

The comment body is composed of paragraphs separated by blank lines. The first paragraph of the comment is a short description of the report. This description will be placed in the function summary table. The next paragraph is long text describing the report in detail. Other paragraphs must start with a tag to identify the type of paragraph; a tag starts with the `@` "at" sign.

Table 454: Supported @ tags

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@code</td>
<td>Indicates that the next lines show a code example using the report.</td>
</tr>
</tbody>
</table>
### Tag: @param name description

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@param name description</td>
<td>Defines a report parameter identified by name, explained by a description. name must match the parameter name in the report declaration.</td>
</tr>
</tbody>
</table>

### Commenting a module

To add a summary to a module, add `#+` lines at the beginning of the source, before module element declarations such as variables, types, constants and functions.

The `#+` lines of the module source comment must appear at the top of the source.

**Example**

```plaintext
#+ This module implements customer information handling
#+
#+ This code uses the 'customer' and 'custdetail' database tables.
#+ Customer input, query and list handling functions are defined here.
#+
DEFINE r_cust RECORD
  cust_id INTEGER,
  cust_name VARCHAR(50),
  cust_address VARCHAR(200)
END RECORD
```

### Commenting a package

To describe a complete directory (package), you must create a `package-info.4gl` file in the directory and add a `#+` comment in the file. The comment will be added to the `package-summary.html` file.

**Commenting a project**

In the top directory of your sources, you must create a `overview.4gl` file with a `#+` comment describing the project. This file is mandatory in order to generate the tree of HTML pages for an entire project, as it is used as the starting point by `fglcomp`.

### Commenting globals

Source comments can be added to symbols defined in a `GLOBALS / END GLOBALS` block.

**Note:** The global symbols included from an external file with `GLOBALS "filename"` will not be part of the module documentation.

**Example**

```plaintext
GLOBALS
  #+ a global variable
  DEFINE globalVar1 INT
  #+ a global type
  TYPE GlobalType1 RECORD
    f1, f2 INT
  END RECORD
  #+ a global constant
  CONSTANT globalConst1 = 9999
END GLOBALS
```
Run the documentation generator

Follow this procedure to produce the source documentation.

Follow this procedure to produce the source documentation.

1. Go to the top directory of your sources.
2. Create a file named overview.4gl, with a #+ comment describing your project.
3. Go to the subdirectories and create files named package-info.4gl with a #+ comment describing the package.
4. Edit the .4gl modules to add #+ comments to functions that must be documented.
5. Go back to the top directory of your sources.
6. Run fglcomp --build-doc overview.4gl
   Use the --W apidoc compiler option to get warnings for invalid comment tags. For example, when a @param tag is missing for a function parameter.
7. To test the result, load the generated index.html file in your preferred browser.

Source preprocessor

A typical preprocessor like in the C language.

Understanding the preprocessor

This is an introduction to the code preprocessor.

The preprocessor is used to transform your sources before compilation. It allows you to include other files and to define macros that will be expanded when used in the source. It behaves similar to the C preprocessor, with some differences.

Important: It is recommended to avoid using the preprocessor if there is an alternative in the native language. For example, instead of defining program constants with an &define macro, use the CONSTANT instruction. Other language features such as IMPORT FGL increase code readability and modular programming, without the need of a preprocessor.

The preprocessor transforms files as follows:

• The source file is read and split into lines.
• Continued lines are merged into one long line if it is part of a preprocessor definition.
• Comments are not removed unless they appear in a macro definition.
• Each line is split into a list of lexical tokens.

The preprocessor options can be passed as compilers command options.

The preprocessor implements the following features:

1. File inclusion
2. Conditional compilation
3. Macro definition and expansion. There are different kind of macros:
   • Simple macros
   • Function macros
   • Predefined macros
4. Macros can be defined with operators for:
   • Stringification
   • Concatenation
5. You can undefine macros.
If a preprocessing directive is invalid, the compilers will generate an .err file with the preprocessing error included in the source file at the line position where the problem exists. When using the -M option, preprocessor errors will be printed to stderr, like regular compiler errors.

**Compilers command line options**

Preprocessor options can be used with fglcomp and fglform compilers.

**File inclusion path**

The -I option defines a single path used to search files included by the &include directives:

```
-I path
```

**Note:** To specify multiple include directories, repeat the -I path option:

```
fglcomp -I /usr/appl/src/common -I /usr/appl/src/stock ...
```

**Macro definition**

The -D option defines a macro with a default value of 1, so that it can be used in conditional directives like &ifdef:

```
-D identifier
```

The -D option can also define a macro with a value:

```
-D identifier=value
```

**Note:** If you want to define a macro with a string value containing spaces, you need to provide the identifier and value as a single command line parameter. For example on UNIX®:

```
-D "VSTR="Version 1.32""
```

The -U option undefines a macro. The macro will not be defined, even if it is defined with the -D option later in the command line, or when it is defined in the code with a &define directive:

```
-U identifier
```

However, predefined macros such as __LINE__ cannot be undefined with the -U option.

**Preprocessing only**

```
-E
```

By using the -E option, only the preprocessing phase is done by the compilers. Result is dumped in standard output.

**Preprocessing style option**

```
-p | nopp | noln | fglpp | auto |
```

When using option -p nopp, it disables the preprocessor phase.

By using option -p noln with the -E preprocessing-only option, you can remove line number information and unnecessary empty lines.

By default, the preprocessor expects an ampersand '&' as preprocessor symbol for macros. The option -p fglpp enables the old syntax, using the hash '#' as preprocessor symbol. The hash '#' syntax is not compatible with single-line comments.
When using \texttt{-p auto}, the compiler detects automatically the '&' ampersand or '#' hash usage for preprocessor commands in the source code. The first preprocessor instruction found in the source will define the preprocessing style.

**Examples**

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{fglcomp -D DEBUG -I /usr/sources/headers program.4gl}</td>
</tr>
<tr>
<td>\texttt{fglcomp -p fglpp program.4gl}</td>
</tr>
<tr>
<td>\texttt{fglcomp -p auto module.4gl}</td>
</tr>
</tbody>
</table>

**Related concepts**

\texttt{fglcomp} on page 2071

The \texttt{fglcomp} tool compiles .4gl source files into .42m p-code modules, and does various other tasks.

\texttt{fglform} on page 2068

The \texttt{fglform} tool compiles form specification files into XML formatted files used by programs.

**File inclusion**

The \&include directive instructs the preprocessor to include a file.

**Syntax**

\texttt{&include "filename"}

1. \textit{filename} is the file to be included during preprocessing. \textit{filename} can be a relative or an absolute path. To specify a path, the slash (/) directory separator can be used for UNIX™ and Windows® platforms.

**Usage**

The file to be included is searched first in the directory containing the current file, then in the directory(s) provided with \texttt{-I} option.

The \textit{filename} argument can be followed by spaces and comments.

The included file will be scanned and processed before continuing with the rest of the current file.

Source: File A

<table>
<thead>
<tr>
<th>First line</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;include &quot;B&quot;</td>
</tr>
<tr>
<td>Third line</td>
</tr>
</tbody>
</table>

Source: File B

<table>
<thead>
<tr>
<th>Second line</th>
</tr>
</thead>
</table>

Result (fglcomp \texttt{-E} output):

\texttt{& 1 "A"}
\texttt{First line}
\texttt{& 1 "B"}
\texttt{Second line}
\texttt{& 3 "A"}
\texttt{Third line}

These preprocessor directives inform the compiler of its current location with special preprocessor comments, so the compiler can provide the right error message when a syntax error occurs.
The preprocessor-generated comments use the following format:

```c
& number "filename"
```

where:
- `number` is the current line in the preprocessed file
- `filename` is the current file name

### Recursive inclusions

Recursive inclusions are not allowed. Doing so will fail and output an error message.

The following example is incorrect:

Source: File A

```c
&include "B"
```

Source: File B

```c
HELLO
&include "A"
```

```c
fglcomp -M A.4gl output
```

```c
B.4gl:2:1:2:1:error:(-8029) Multiple inclusion of the source file 'A'.
```

Including the same file several times is allowed:

Source: File A

```c
&include "B"
&include "B"  -- correct
```

Source: File B

```c
HELLO
```

Result (`fglcomp -E output`):

```c
& 1 "A"
& 1 "B"
HELLO
& 2 "A"
& 1 "B"
HELLO
```

### File path

In the `&include "filename"` macro, `filename` can be a relative or an absolute path.

To specify a path, the slash (/) directory separator can be used for UNIX™ and Windows® platforms:

```c
&include "../include/common.inc"
```
Conditional compilation
Integrate code lines conditionally.

Syntax 1
```
&ifdef identifier
...  
&else
...\}
&endif
```

1. `identifier` is a preprocessor constant.

Syntax 2
```
&ifndef identifier
...  
&else
...\}
&endif
```

1. `identifier` is a preprocessor constant.

Usage
The `&ifdef` and `&ifndef` preprocessor macros can be used to integrate code lines conditionally depending on the existence of a preprocessor constant.

The constant is defined with a `&define` or with the `-D` option in the command line.

Even if the condition is evaluated to false, the content of the `&ifdef` block is still scanned and tokenized. Therefore, it must be lexically correct.

Sometimes it is useful to use some code if a macro is not defined. You can use `&ifndef`, that evaluates to true if the macro is not defined.

Source: File A
```
&define IS_DEFINED
&ifdef IS_DEFINED
DISPLAY "The macro is defined"
&endif /* IS_DEFINED */
```

Result (fglcomp -E output):
```
 & 1 "A"

DISPLAY "The macro is defined"
```

Simple macro definition
A simple macro is identified by its name and body.

Syntax
```
&define identifier body
```

1. `identifier` is the name of the macro. Any valid identifier can be used.
2. *body* is any sequence of tokens until the end of the line. After substitution, the macro definition is replaced with blank lines.

**Note:** A preprocessor directive must start at the beginning of the line. The & ampersand character can be preceded by blanks.

**Usage**

As the preprocessor scans the text, it substitutes the macro body for the name identifier.

The following example show macro substitution with 2 simple macros:

**Source:** File A

```c
#define MAX_TEST 12
#define HW "Hello world"

MAIN
    DEFINE i INTEGER
    FOR i=1 TO MAX_TEST
        DISPLAY HW
    END FOR
END MAIN
```

**Result (fglcomp -E output):**

```c
& 1 "A"

MAIN
    DEFINE i INTEGER
    FOR i=1 TO 12
        DISPLAY "Hello world"
    END FOR
END MAIN
```

The macro definition can be continued on multiple lines, but when the macro is expanded, it is joined to a single line as follows:

**Source:** File A

```c
#define TABLE_VALUES 1, \ 2, \ 3
DISPLAY TABLE_VALUES
```

**Result (fglcomp -E output):**

```c
& 1 "A"

DISPLAY 1, 2, 3
```

The source file is processed sequentially, so a macro takes effect at the place it has been written:

**Source:** File A

```c
DISPLAY X
#define X "Hello"
```
DISPLAY X

Result (fglcomp -E output):

& 1 "A"
DISPLAY X
DISPLAY "Hello"

The macro body is expanded only when the macro is applied:

Source: File A

&define AA BB
&define BB 12
DISPLAY AA

Result (fglcomp -E output):

& 1 "A"

DISPLAY 12

- AA is first expanded to BB.
- The text is re-scanned and BB is expanded to 12.
- When the macro AA is defined, BB is not known yet; but it is known when the macro AA is used.

In order to prevent infinite recursion, a macro cannot be expanded recursively.

Source: File A

&define A B
&define B A
&define C C
A C

Result (fglcomp -E output):

& 1 "A"

A C

- A is first expanded to B.
- B is expanded to A.
- A is not expanded again as it appears in its own expansion.
- C expands to C and can not be expanded further.

**Note:** It is also possible to define a macro with the `-D` command line option of compilers.

**Function macro definition**

Function macros are preprocessor macros which can take arguments.

**Syntax**

&define identifier( arglist ) body
1. **identifier** is the name of the macro. Any valid identifier can be used.
2. **body** is any sequence of tokens until the end of the line.
3. **arglist** is a list of identifiers separated with commas and optionally white space.
4. There must be no space or comment between the macro name and the opening parenthesis (). Otherwise the macro is not a function macro, but a simple macro.

**Note:** A preprocessor directive must start at the beginning of the line. The & ampersand character can be preceded by blanks.

**Usage**

Function macros take arguments that are replaced in the body by the preprocessor.

Source: File A

```plaintext
&define function_macro(a,b) a + b
&define simple_macro (a,b) a + b
function_macro( 4 , 5 )
simple_macro (1,2)
```

Result (fglcomp -E output):

```
& 1 "A"
4 + 5
(a,b) a + b (1,2)
```

A function macro can have an empty argument list. In this case, parentheses () are required for the macro to be expanded. As we can see in the following example, the third line is not expanded because there is no parentheses after foo. The function macro cannot be applied even if it has no arguments.

Source: File A

```plaintext
&define foo() yes
foo()
foo
```

Result (fglcomp -E output):

```
& 1 "A"
yes
foo
```

The comma separates arguments. Macro parameters containing a comma can be used with parentheses. In this example, the second line has been substituted, but the third line produced an error, because the number of parameters is incorrect.

Source: File A

```plaintext
&define one_parameter(a) a
one_parameter((a,b))
one_parameter(a,b)
```

```
source.4gl:3:1:3:1:error:(-8039) Invalid number of parameters for macro one_parameter.
```
Macro arguments are completely expanded and substituted before the function macro expansion. A macro argument can be left empty.

Source: File A

```c
#define two_args(a,b) a b
two_args(,b)
two_args(,)
two_args()
two_args(,,)
```

```bash
fglcomp -M output
```

```
source.4gl:4:1:4:1:error:(-8039) Invalid number of parameters for macro two_args.
source.4gl:5:1:5:1:error:(-8039) Invalid number of parameters for macro two_args.
```

Macro arguments appearing inside strings are not expanded.

Source: File A

```c
#define foo(x) "x"
foo(toto)
```

Result (fglcomp -E output):

```
& 1 "A"
"x"
```

Related concepts

**Simple macro definition** on page 2136

A simple macro is identified by its name and body.

Predefined macros

A set of predefined preprocessor macros are available.

The preprocessor predefines 2 macros:

1. `__LINE__` expands to the current line number. Its definition changes with each new line of the code.
2. `__FILE__` expands to the name of the current file as a string constant. For example: "subdir/file.inc"

These macros are often used to generate error messages.

An `#include` directive changes the values of `__FILE__` and `__LINE__` to correspond to the included file.

Stringification operator

Transforms a preprocessor macro element to a string.

**Syntax**

```
#param
```

1. `param` is a parameter of the macro
Usage
The stringification operator `#` converts a preprocessor macro parameter to a string.
When a macro parameter is used with a preceding `#`, it is replaced by a string containing the literal text of the argument.
The argument is not macro expanded before the substitution.
Source: File A

```
&define disp(x) DISPLAY #x
disp(abcdef)
```

Result (`fglcomp -E` output):
```
& 1 "A"
DISPLAY "abcdef"
```

Concatenation operator
Concatenates two parameters of a preprocessor macro.

Syntax
```
token1 ## token2
```
1. `token1` is a parameter of the macro or a simple token.
2. `token2` is a parameter of the macro or a simple token.

Usage
The double-hash operator `##` can be used to merge two tokens while expanding a macro and create a single token.
All tokens can not be merged. Usually these tokens are identifiers, or numbers.
The concatenation result produces an identifier.
Source: File A

```
&define COMMAND(NAME) #NAME, NAME ## _command
COMMAND(quit)
```

Result (`fglcomp -E` output):
```
& 1 "A"
"quit", quit_command
```

Undefining a macro
Undefines a preprocessor macro.

Syntax
```
&undef identifier
```
1. `identifier` is a preprocessor constant.
Usage

If a macro is redefined without having been undefined previously, the preprocessor issues a warning and replaces the existing definition with the new one. First undefine a macro with the `&undef` directive.

Source: File A

```c
#define HELLO "hello"
DISPLAY HELLO
&undef HELLO
DISPLAY HELLO
```

Result (`fglcomp -E` output):

```c
&A
DISPLAY "hello"
DISPLAY HELLO
```

Note: It is also possible to undefine a macro with the `-U` command line option of compilers. However, predefined macros cannot be undefined with this option.

Integrated debugger

Describes the command-line debugger you can use to find bugs in your programs.

Understanding the debugger

This is an introduction to the integrated debugger.

The debugger is a feature built in to the runtime system (`fglrun`) that allows you to control the execution of a program step by step, so that you can find logical and runtime errors.

There are three debug modes possible with the Genero runtime system:

1. Start the `fglrun` program from the command line with the `-d` option. For more details, see Starting `fglrun` in debug mode on page 2143.
2. Attaching with the `fgldb` tool, to an `fglrun` process running on the same machine, by using the process id. Note that with the graphical debugger of Genero Studio, a debug session with `fgldb` can be started on a remote server through ssh. For more details, see Attaching to a running program on page 2143.
3. Connect directly with the `fgldb` tool, to the debug TCP port of a runtime system running on a mobile device in standalone mode. For more details, see Debugging on a mobile device on page 2145.

The debugger supports a subset of the standard GNU C/C++ debugger called `gdb`.

In command line mode, the debugger shows the following prompt

```
(fgldb)
```

A command is a single line of input. It starts with a command name, which may be followed by arguments whose meaning depends on the command name. For example, the command `step` accepts as an argument the number of times to step:

```
(fgldb) step 5
```

You can use command abbreviations. For example, the 'step' command abbreviation is 's':

```
(fgldb) s 5
```

Possible command abbreviations are shown in the command's syntax.
A blank line as input to the debugger (pressing just the RETURN or ENTER keys) usually causes the previous command to repeat. However, commands whose unintentional repetition might cause problems will not repeat in this way.

**Prerequisites to run the debugger**

Some requirements are needed before running the integrated debugger.

**FGLPROFILE configuration option to disable the debugger**

Make sure that the following FGLPROFILE entry is not defined, or defined as false:

```
fgr.jgnoreDebuggerEvent = false
```

**Note:** In production environments, the `fgr.ignoreDebuggerEvent` FGLPROFILE entry should be set for true.

**Source file search path (FGLSOURCEPATH)**

Before starting the debugger, make sure you have properly set the FGLLDPATH and FGLSOURCEPATH environment variable, to let the debugger find the source files.

In order to find source files, the debugger will search in the directories defined by FGLLDPATH. If the source file is not found, the search continues in the directories defined by FGLSOURCEPATH.

The FGLSOURCEPATH environment variable is provided to distinguish execution directories (containing .42m files), from source directories (containing .4gl files), when the sources are not located in the same directory as the pcode files.

**UNIX™ example:**

```
$ FGLSOURCEPATH="/usr/app/source:/home/scott/sources"
$ export FGLSOURCEPATH
```

**Windows® example:**

```
C:\> set FGLSOURCEPATH=C:\app\sources;C:\scott\sources
```

**Starting fgrun in debug mode**

The runtime system can be started in debug mode with the –d option.

To start fgrun in debug mode, use the –d option of fgrun, for example:

```
fgrun -d myprog
```

This mode is typically used in development environments when using the command line tools.

The syntax of the commands is similar to the gdb debugger.

**Attaching to a running program**

It is possible to start the debugger for a program running on the same computer.

**Basics**

Use the fglpdb command with the –p pid option, to switch an fgrun process running on the same computer into debug mode.

The fglpdb command must be executed on the machine where the fgrun process executes.
Tip: The graphical debugger of Genero Studio is able to start the fgldb tool on another computer through ssh, so that it is possible to debug a program running on a remote machine. See Genero Studio documentation for more details.

The fgldb command line tool takes the fglr proc process id as value for the –p argument.

Note: Before starting a debug session, make sure that you fulfill the prerequisites for debugging.

Debug a program running on a UNIX server

First, identify the process id of the fglr process running on your server.

For example, on a UNIX® platform, use the ps command:

```
$ ps a | grep fglrun
10646 pts/0    S+     0:00 /opt/myapp/fgl/lib/fglrun stockinfo.42m
```

Note: Inspect the GAS log files to find the id of a fglrun process running behind a GAS application server. Enable full log reports in the GAS to get detailed information about process execution.

You may want to debug processes that use a lot of machine resources (processor, memory or open files). Use a system utility to find a process id by resources used (for example, the top command on Linux®).

Execute the fgldb tool with the process id of the program you want to attach to:

```
$ fgldb -p 10646
108 DISPLAY ARRAY contlist TO sr.*

(fgldb)
```

The (fgldb) prompt indicates that you are now connected to the fglrun process, and the program flow is suspended. To continue with the program flow, enter the "continue" debugger command:

```
(fgldb) continue
Continuing.
```

The application will then resume. To suspend the program again and enter debugger commands, press CTRL-C in the debug console. fgldb will display the interrupt message and return control to the debugger:

```
... Continuing.
^CINTERRUPT
108 DISPLAY ARRAY contlist TO sr.*

(fgldb)
```

At this point, you can enter debugger commands. For example, set a break point and continue until the break point is reached:

```
(fgldb) b 427
Breakpoint 2 at 0x00000000: file contacts.4gl, line 427.
(fgldb) continue
Continuing.
Breakpoint 2, edit_contact() at contacts.4gl:427
427 IF new THEN

(fgldb)
```

To finish the debug session, close the connection with the "detach" debugger command:

```
(fgldb) detach
Connection closed by foreign host.
```
**Related concepts**

*fgldb* on page 2078

The *fgldb* tool is an interface program for remote debugging.

**Debugging on a mobile device**

It is possible to remotely start the debugger for an app running on a mobile device.

**Basics**

When an app was created with debug mode and is running on a device, it is possible to switch the runtime system in debug mode, by using the *fgldb* command tool with the `-m` option.

**Important:** The app must have been created in debug mode. Apps created in release mode cannot be debugged with the *fgldb* tool. For more details, check out how to build mobile apps with debug mode in the Deploying mobile apps on page 3588 section.

**Important:** On iOS devices, after installing the app, you need to enable the debug port in the app settings, otherwise the app will not listen to the debug port.

The *fgldb* command line tool takes two arguments: The host (or IP address) of the mobile device, and an optional TCP port number to connect to. For mobile devices, the debug TCP port is 6400. Note that this is the same port the mobile front-end is listening to for GUI connection, when working in GUI client/server mode.

**Note:** Before starting a debug session, make sure that you fulfill the prerequisites for debugging.

**Debugging an app running on a physical device**

Considering the mobile device IP address is "192.168.1.23", and the application is running locally on a physical mobile device, you can open a debug session from the development machine as follows:

```
$ fgldb -m 192.168.1.23:6400
```

The *(fgldb)* prompt indicates that you are now connected to the *fglrun* process on mobile device, and the program flow is suspended. To continue with the program flow, enter the "continue" debugger command:

```
(fgldb) continue
Continuing.
```

The application will then resume on the mobile device. To suspend the program again and enter debugger commands, press CTRL-C in the debug console; *fgldb* will show the interrupt message and give you the control back:

```
Continuing.
^CINTERRUPT
```

At this point, you can for example set a break point and continue until the break point is reached:

```
(fgldb) b 427
Breakpoint 2 at 0x00000000: file contacts.4gl, line 427.
```

```
(fgldb) continue
Continuing.
Breakpoint 2, edit_contact() at contacts.4gl:427
427     IF new THEN
```

```
(fgldb)```
To finish the debug session, close the connection with the "detach" debugger command:

```
(fgldb) detach
Connection closed by foreign host.
```

### Debugging an app running on an Android device emulator

When the mobile application is executing on an Android™ device emulator in the same machine as the development environment, you must first forward the 6400 TCP port.

To forward the port 6400, use the following Android Debug Bridge tool command:

```
$ adb forward tcp:6400 tcp:6400
```

To be able to show GMA service debug information from a browser with the http://localhost:6480 URL, you need also to forward the port 6480:

```
$ adb forward tcp:6480 tcp:6480
```

#### Related concepts

**fgldb** on page 2078

The **fgldb** tool is an interface program for remote debugging.

### Stack frames in the debugger

The *stack frame* contains information about function call stack.

- Each time your program performs a function call, information about the call is saved in a block of data called a *stack frame*. Each frame contains the data associated with one call to one function.

- The stack frames are allocated in a region of memory called the *call stack*. When your program is started, the stack has only one frame, that of the function **main**. This is the initial frame, also known as the *outermost frame*. As the debugger executes your program, a new frame is made each time a function is called. When the function returns, the frame for that function call is eliminated.

- The debugger assigns numbers to all existing stack frames, starting with zero for the innermost frame, one for the frame that called it, and so on upward. These numbers do not really exist in your program; they are assigned by the debugger to allow you to designate stack frames in commands.

- Each time your program stops, the debugger automatically selects the currently executing frame and describes it briefly. You can use the **frame** command to select a different frame from the current call stack.

#### Related concepts

**Runtime stack** on page 493

The runtime stack is used to pass/return values to/from functions.

### Setting a breakpoint programmatically

Set a breakpoint in the program source code with the **BREAKPOINT** instruction.

If the program flow encounters this instruction, the program stops as if the break point was set by the **break** command:

```
MAIN
  DEFINE i INTEGER
  LET i=123
  BREAKPOINT
  DISPLAY i
END MAIN
```

The **BREAKPOINT** instruction is simply ignored when running in normal mode.
Expressions in debugger commands

A limited expression syntax can be used in debugger commands.

Some debugger commands such as `display` take an expression as argument. The Genero debugger supports a reduced syntax for command expressions described in this section. For a detailed description of comparison operators, constant values and operands, see Expressions.

Syntax

```
variable
| char-const
| int-const
| dec-const
| NULL
| TRUE
| FALSE
| expression IS [NOT] NULL
| expression = expression
| expression == expression
| expression <= expression
| expression => expression
| expression < expression
| expression > expression
| expression + expression
| expression - expression
| expression * expression
| expression / expression
| expression OR expression
| expression AND expression
| NOT expression
| - expression
| ( expression )
```

Note:

1. `variable` is a program variable name.
2. `char-const` is a character string literal delimited by single or double quotes.
3. `int-const` is an integer literal.
4. `dec-const` is a decimal number literal.
5. `expression` is a combination of one or more listed syntax elements.

Example

```
(fgldb) display a + 1000
1: a = 1140.50
```

Debugger commands

This topic lists all debugger commands.

**backtrace / where**

The `backtrace` commands prints a summary of how your program reached the current state.
Usage
The backtrace command prints a summary of your program's entire stack, one line per frame. Each line in the output shows the frame number and function name.

bt and where are aliases for the backtrace command.

Example

```
(fgldb) backtrace
#1 addcount() at mymodule.4gl:6
#2 main() at mymodule.4gl:2
(fgldb)
```

break
The break command defines a break point to stop the program execution at a given line or function.

Syntax

```
break [ [module. | function | module::line ] [if condition ] ]
```

where location is:

```
<table>
<thead>
<tr>
<th>module.</th>
<th>function</th>
<th>module::line</th>
</tr>
</thead>
</table>
```

1. module is the name of a source file, without extension.
2. function is a function name.
3. line is a source code line.
4. condition is an expression evaluated dynamically.

Usage
The break command sets a break point at a given position in the program.

When the program is running, the debugger stops automatically at breakpoints defined by this command.

If a condition is specified, the program stops at the breakpoint only if the condition evaluates to TRUE.

If you do not specify any location, the breakpoint is created for the current line. For example, if you write "break if var = 1", the debugger adds a conditional breakpoint for the current line, and the program will only stop if the variable is equal to 1 when reaching the current line again.

Example

```
(fgldb) break mymodule:5
Breakpoint 2 at 0x00000000: file mymodule.4gl, line 5.
```

Related concepts
BREAKPOINT on page 493
The BREAKPOINT instruction sets a program breakpoint when running in debug mode.

Setting a breakpoint programmatically on page 2146
Set a breakpoint in the program source code with the BREAKPOINT instruction.

**call**
The **call** command calls a function in the program.

**Syntax**

```
call function-name ( [ expression [, ...] ] )
```

1. *function-name* is the name of the function to call.
2. *expression* is a combination of variables, constants and operators.

**Usage**
The **call** command invokes a function of the program and returns the control to the debugger.
The return values of the function are printed as a comma-separated list delimited by curly brackets.

**Example**

```
MAIN
  DEFINE i INTEGER
  LET i = 1
  DISPLAY i
END MAIN

FUNCTION hello ()
  RETURN "hello", "world"
END FUNCTION
```

```
(fgldb) br main
Breakpoint 1 at 0x00000000: file t.4gl, line 4.
(fgldb) run
Breakpoint 1, main() at t.4gl:4
  4       LET i = 1
(fgldb) call hello()
$1 = { "hello", "world" }
(fgldb)
```

**clear**
The **clear** command clears the breakpoint at a specified line or function.

**Syntax**

```
clear [ location ]
```

where *location* is:

```
  [ module. ] function
  [ module: ] line
```

1. *module* is the name of a source file, without extension.
2. *function* is a function name.
3. *line* is a source code line.
**Usage**

With the `clear` command, you can delete breakpoints wherever they are in your program.

Use the `clear` command with no arguments to delete any breakpoints at the next instruction to be executed in the selected stack frame.

Use the `delete` command to delete individual breakpoints by specifying their breakpoint numbers.

**Example**

```
(fgldb) clear mymodule:5
Deleted breakpoint 2
(fgldb)
```

**continue**

The `continue` command continues the execution of the program after a breakpoint.

**Syntax**

```
continue [ ignore-count ]
```

1. `ignore-count` defines the number of times to ignore a breakpoint at this location.

**Usage**

The `continue` command continues the execution of the program until the program completes normally, another breakpoint is reached, or a signal is received.

`c` is an alias for the `continue` command.

**Example**

```
(fgldb) continue
...
(program output)
...
Program exited normally.
```

**delete**

The `delete` command allows you to remove breakpoints that you have specified in your debugger session.

**Syntax**

```
delete breakpoint
```

1. `breakpoint` is the number assigned to the breakpoint by the debugger.

**Usage**

The `delete` command allows you to remove breakpoints when they are no longer needed in your debugger session.

If you prefer you may disable the breakpoint instead, see the `disable` command.

`d` is an alias for the `delete` command.

**Example**

```
(fgldb) delete 1
(fgldb) run
Program exited normally.
```
**detach**
The `detach` command closes the TCP connection of a remote debug session.

**Syntax**
```
detach
```

**Usage**
The `detach` command must be used to terminate a remote debug session, by closing the debug TCP connection.

**Example**
```
(fglldb) detach
```

**Related concepts**
- **Attaching to a running program** on page 2143
  It is possible to start the debugger for a program running on the same computer.
- **Debugging on a mobile device** on page 2145
  It is possible to remotely start the debugger for an app running on a mobile device.

**disable**
The `disable` command disables the specified breakpoint.

**Syntax**
```
disable breakpoint
```

1. `breakpoint` is the number assigned to the breakpoint by the debugger.

**Usage**
The `disable` command instructs the debugger to ignore the specified breakpoint when running the program.

Use the `enable` command to reactivate the breakpoint for the current debugger session.

**Example**
```
(fglldb) disable 1
(fglldb) run
Program exited normally.
(fglldb)
```

**display**
The `display` command displays the specified expression's value each time program execution stops.

**Syntax**
```
display expression
```

1. `expression` is a combination of variables, constants and operators.
Usage
The display command allows you to add an expression to an automatic display list. The values of the expressions in the list are printed each time program execution stops. Each expression in the list is assigned a number to identify it.

This command is useful in tracking how the values of expressions change during the program's execution.

Example

```
(fgldb) display a
1: a = 6
(fgldb) display i
2: i = 1
(fgldb) step
2: i = 1
1: a = 6
16 for i = 1 to 10
(fgldb) step
2: i = 2
1: a = 6
17 let a = a+1
(fgldb)
```

down
The down command moves down in the call stack.

Syntax

down

Usage
The down command moves the focus of the debugger down from the frame currently being examined, to the frame of its callee.

The command selects and prints the function called by the current function.

See stack frames for a brief description of frames.

```
(fgldb) down
#0 query_cust() at custquery.4gl:22
22 CALL cleanup()
(fgldb)
```

echo
The echo command prints the specified text as prompt.

Syntax

echo text

1. text is the specific text to be output.

Usage
The echo command allows you to generate exactly the output that you want.
Special characters can be included in text using C escape sequences, such as '\n' to print a newline. No newline is printed unless you specify one. In addition to the standard C escape sequences, a backslash followed by a space stands for a space. A backslash at the end of text can be used to continue the command onto subsequent lines.

**Example**

```
(fgldb) echo hello\n
hello
(fgldb)
```

**enable**

The `enable` command enables breakpoints that have previously been disabled.

**Syntax**

```
enable breakpoint
```

1. `breakpoint` is the number assigned to the breakpoint by the debugger.

**Usage**

The `enable` command allows you to reactivate a breakpoint in the current debugger session.

The breakpoint must have been disabled using the `disable` command.

**Example**

```
(fgldb) disable 1
(fgldb) run
Program exited normally.
(fgldb) enable 1
(fgldb) run
Breakpoint 1, at mymodule.4gl:5
```

**finish**

The `finish` command continues the execution of a program until the current function returns normally.

**Syntax**

```
finish
```

**Usage**

The `finish` command instructs the program to continue running until just after the function in the selected stack frame returns, and then stop.

The returned value, if any, is printed.

**Example**

```
(fgldb) finish
Run till exit myfunc() at module.4gl:10
Value returned is $1 = 123
(fgldb)
```
**frame**
The `frame` command selects and prints a stack frame.

**Syntax**

```
frame [ number ]
```

1. `number` is the stack frame number of the frame that you wish to select.

**Usage**
The `frame` command allows you to move from one stack frame to another, and to print the stack frame that you select. Each stack frame is associated with one call to one function within the currently executing program. Without an argument, the current stack frame is printed.

See [stack frames](#) for a brief discussion of frames.

**Example**

```
(fgldb) frame
#0 func1() at stack.4gl:6
6 FUNCTION func1(name,id)
(fgldb)
```

**Related concepts**

- [Runtime stack](#) on page 493
  
The runtime stack is used to pass/return values to/from functions.

**help**
The `help` command provides information about debugger commands.

**Syntax**

```
help [ command ]
```

1. `command` is the name of the debugger command for which you wish information.

**Usage**
The `help` command displays a short explanation of a specified command.

Enter the `help` command with no arguments to display a list of debugger commands.

**Example**

```
(fgldb) help delete
Delete some breakpoints or auto-display expressions
```

**ignore**
The `ignore` command defines the number of times a breakpoint must be ignored.

**Syntax**

```
ignore breakpoint count
```

1. `breakpoint` is the breakpoint number.
2. `count` is the number of times the breakpoint will be ignored.
**Usage**

The `ignore` command defines the number of times a breakpoint is ignored when the program flow reaches that breakpoint.

The next `count` times the breakpoint is reached, the program execution will continue, and no breakpoint condition is checked.

You can specify a `count` of zero to make the breakpoint stop the next time it is reached.

When using the `continue` command to resume the execution of the program from a breakpoint, you can specify an ignore count directly as an argument.

**Example**

```mbox
(fgldb) br main
Breakpoint 1 at 0x00000000: file t.4gl, line 4.
(fgldb) ignore 1 2
Will ignore next 2 crossings of breakpoint 1.
(fgldb) run 1
Program exited normally.
(fgldb) run 1
Program exited normally.
(fgldb) run
Breakpoint 1, main() at t.4gl:4
4 LET i = 1
(fgldb)
```

**info**

The `info` command describes the current state of your program.

**Syntax**

```
info {breakpoints | sources | program | variables | locals | files | line location}
```

where `location` is:

```
[module.]function
[module:]line
```

1. `module` is the name of a source file, without extension.
2. `function` is a function name.
3. `line` is a source code line.

**Usage**

The `info` command describes the state of your program.

- `info breakpoints` lists the breakpoints that you have set.
- `info sources` prints the names of all the source files in your program.
- `info program` displays the status of your program.
- `info variables` displays global variables.
- `info locals` displays the local variables of the current function.
• info files lists the files from which symbols were loaded.
• info line module.function prints the program addresses for the first line of the function named function.
• info line module:line prints the starting and ending addresses of the compiled code for the source line specified. See the list command for all the ways that you can specify the source code line.

Example

(fgldb) info sources
Source files for which symbols have been read in:
mymodule.4gl, fglwinexec.4gl, fglutil.4gl, fgldialog.4gl, fgdummy4js.4gl
(fgldb)

list
The list command prints source code lines of the program being executed.

Syntax

list [location]

where location is:

[ module, ] function
[ module: ] line

1. module is the name of a specific source file, without extension.
2. function is a function name.
3. line is a source code line.

Usage
The list command prints source code lines of your program, by default it begins with the current line.

Example

(fgldb) run
Breakpoint 1, at mymodule.4gl:5
5 CALL addlist() (fgldb) list
5 CALL add_customer(cust_rec.*)
6 MESSAGE "Customer record was added"
...
14 END FUNCTION (fgldb)

next
The next command continues running the program by executing the next source line in the current stack frame, and then stops.

Syntax

next [ count ]

1. count defines the number of lines to execute before stopping.
**Usage**

The `next` command allows you to execute your program one line of source code at a time. The `next` command is similar to `step`, but function calls that appear within the line of code are executed without stepping into the function code.

When the next line of code at the original stack level that was executing when you gave the `next` command is reached, execution stops.

Using a `count` parameter will repeat the `step` command `count` times.

After reaching a breakpoint, the `next` command can be used to examine a troublesome section of code more closely. `n` is an alias for the `next` command.

**Example**

```
(fgldb) next
5   CALL add_customer(cust_rec.*)
(fgldb) next
6   MESSAGE "Customer record was added"
(fgldb) next 2
8   RETURN TRUE
```

**output**

The `output` command prints only the value of the specified expression, suppressing any other output.

**Syntax**

```
output expression
```

1. `expression` is a combination of variables, constants and operators.

**Usage**

The `output` command prints the current value of the expression and nothing else, no newline character, no "expr=", etc.

The usual output from the debugger is suppressed, allowing you to print only the value.

**Example**

```
(fgldb) output cust_rec.cust_id
87324(fgldb)
```

**print**

The `print` command displays the current value of the specified expression.

**Syntax**

```
print expression
```

1. `expression` is a combination of variables, constants and operators.

**Usage**

The `print` command allows you to examine the data in your program.

It evaluates and prints the value of the specified expression from your program, in a format appropriate to its data type.

`p` is an alias for the `print` command.
Example

```
(fgldb) print cust_rec.cust_id
$1 = 87324
(fgldb)
```

ptype
The `ptype` command prints the data type or structure of a variable.

**Syntax**

```
ptype variable-name
```

1. `variable-name` is the name of the variable.

**Example**

```
(fgldb) ptype cust_rec
type = RECORD
cust_num INTEGER,
cust_name VARCHAR(10),
cust_address VARCHAR(200)
END RECORD
```

quit
The `quit` command terminates the debugger session.

**Syntax**

```
quit
```

**Usage**

The `quit` command allows you to exit the debugger.

`q` is an alias for the `quit` command.

**Example**

```
(fgldb) quit
```

run
The `run` command starts the program.

**Syntax**

```
run [argument [...] ]
```

1. `argument` is an argument to be passed to the program.

**Usage**

The `run` command causes your program to execute until a breakpoint is reached or the program terminates normally.

**Example**

```
(fgldb) run a b c
Breakpoint 1, at mymodule.4gl:3
```
set
The set command allows you to configure your debugger session and change program variable values.

Syntax

set {
  annotate {1|0}
  environment envname=value
  prompt ptext
  set print elements elemcount
  variable varname=expression
  verbose {on|off}
}

1. ptext is the string to which the prompt should be set.
2. varname is the program variable to be set to expression.
3. expression is a combination of variables, constants and operators.
4. envname is the environment variable to be set to value.
5. elemcount is the number of elements to define.

Usage
The set command allows to change program variables and/or debug environment settings.

set variable sets a program variable, to be taken into account when continuing program execution. The right operand can be an expression.

set prompt changes the prompt text. The text can be set to add string. A space is not automatically added after the prompt string, allowing you to determine whether to add a space at the end of the prompt string.

set environment sets an environment variable, where value may be any string. If the value parameter is omitted, the variable is set to a null value. The variable is set for your program, not for the debugger itself.

set verbose on forces the debugger to display additional messages about its operations, allowing you to observe that it is still working during lengthy internal operations.

set annotate 1 switches the output format of the debugger to be more machine readable.

set print elements elemcount defines the maximum number of array elements to be printed by the debugger when displaying a program array.

Example

(fgldb) set prompt ($)
($)  

On UNIX™ systems, if your SHELL variable names a shell that runs an initialization file, any variables you set in that file affect your program. You may wish to move the setting of environment variables to files that are only run when you sign on, such as .login or .profile.

source
The source command executes a file of debugger commands.

Syntax

source cmdfile

1. cmdfile is the name of the file containing the debugger commands.
**Usage**

The `source` command allows you to execute a command file of lines that are debugger commands. The lines in the file are executed sequentially. The commands are not printed as they are executed, and any messages are not displayed. Commands are executed without asking for confirmation. An error in any command terminates execution of the command file.

**Example**

Using the text file `cmdfile.txt`, which contains the single line with a `break` command:

```
$ cat cmdfile.txt
break 10

$ fgldr -d myprog
(fgldb) source cmdfile.txt
Breakpoint 2 @ 0x00000000: file mymod.4gl, line 10.
(fgldb)
```

**signal**

The `signal` command sends an interruption signal to the program.

**Syntax**

```
signal signal
```

**Usage**

The `signal` command resumes execution where your program stopped, but immediately gives it the signal `signal`. `signal` can be the name or the number of a signal.

For example, on many systems `signal 2` and `signal SIGINT` are both ways of sending an interrupt signal. The `signal SIGINT` command resumes execution of your program where it has stopped, but immediately sends an interrupt signal. The source line that was current when the signal was received is displayed.

**Note:** The current version only allows the `SIGINT` signal.

**Example**

```
(fgldb) signal SIGINT
Program exited normally.
16  for i = 1 to 10
(fgldb)
```

**step**

The `step` command continues running the program by executing the next line of source code, and then stops.

**Syntax**

```
step \ count \n```

1. `count` defines the number of lines to execute before stopping.

**Usage**

The `step` command allows you to "step" through your program, executing one line of source code at a time.
When a function call appears within the line of code, that function is also stepped through. A common technique is to set a breakpoint prior to the section or function that is causing problems, run the program till it reaches the breakpoint, and then step through it line by line.

Using a `count` parameter will repeat the `step` command `count` times.

`s` is an alias for the `step` command.

**Example**

```
(fgldb) step
4      CALL add_customer(cust_rec.*)
(fgldb) step 2
6      MESSAGE "Customer record was added"
```

**tbreak**

The `tbreak` command sets a temporary breakpoint.

**Syntax**

```
tbreak [ location ] [ if condition ]
```

where `location` is:

```
| [module.]function
| [module:]line
|
```

1. `module` is the name of a source file, without extension.
2. `function` is a function name.
3. `line` is a source code line.
4. `condition` is an expression evaluated dynamically.

**Usage**

The `tbreak` command sets a breakpoint for one stop only.

The breakpoint is set in the same way as with the `break` command, but the breakpoint is automatically deleted after the first time your program stops there.

If a `condition` is specified, the program stops at the breakpoint only if the `condition` evaluates to true.

If you do not specify any location (function or line number), the breakpoint is created for the current line. For example, if you write "`tbreak if var = 1`", the debugger adds a conditional breakpoint for the current line, and the program will only stop if the variable is equal to 1 when reaching the current line again.

**Example**

```
(fgldb) tbreak 12
Breakpoint 2 at 0x00000000: file custmain.4gl, line 12.
(fgldb)
```

**tty**

The `tty` command resets the default program input and output for future run commands.

**Syntax**

```
tty filename
```
1. **filename** is the file which is to be the default for program input and output.

**Usage**

The `tty` command instructs the debugger to redirect program input and output to the specified file for future run commands.

The redirection is for your program only; your terminal is still used for debugger input and output.

**Example**

```
(fgldb) tty /dev/ttyS0
(fgldb)
```

**undisplay**

The `undisplay` command cancels expressions to be displayed when the program execution stops.

**Syntax**

```
undisplay itemnum
```

1. **itemnum** is the number of the expressions for which the display is canceled.

**Usage:**

When the `display` command is used, each expression displayed is assigned an item number.

The `undisplay` command allows you to remove expressions from the list to be displayed, using the item number to specify the expression to be removed.

**Example**

```
(fgldb) step
2: i = 2
1: a = 20
9     FOR i = 1 TO 10
(fgldb) undisplay 2
(fgldb) step
1: a = 20
10   LET cont = TRUE
(fgldb)
```

**until**

The `until` command continues running the program until the specified location is reached.

**Syntax**

```
until [ location ]
```

where **location** is:

```
   [module.]function
   [module:]line
```

1. **module** is the name of a source file, without extension.
2. **function** is a function name.
3. **line** is a source code line.
**Usage**

The `until` command continues running your program until either the specified location is reached, or the current stack frame returns.

This command can be used to avoid stepping through a loop more than once.

**Example**

```
(fgldb) until add_customer()
```

**up**

The `up` command selects and prints the function that called this one, or the function specified by the frame number in the call stack.

**Syntax**

```
up \[ frames \]
```

1. `frames` specifies how many frames up to go in the stack. The default is 1.

**Usage**

The `up` command moves towards the outermost frame, to frames that have existed longer. To print the function that called the current function, use the `up` command without an argument.

See [stack frames](#) for a brief description of frames.

**Example**

```
(fgldb) up #1 main() at customain.4gl:14
14    CALL query_cust()
(fgldb)
```

**watch**

The `watch` command sets a watchpoint for an expression.

**Syntax**

```
watch expression \[if boolean-expression]\n```

1. `expression` is a combination of variables, constants and operators.
2. `boolean-expression` is an optional boolean expression.

**Usage**

The `watch` command stops the program execution when the value of the expression changes.

If `boolean-expression` is provided, the `watch` command stops the execution of the program if the expression value has changed and the `boolean-expression` evaluates to true.

The watchpoint cannot be set if the program is not in the context where `expression` can be evaluated. Before using a watchpoint, you typically set a breakpoint in the function where the `expression` makes sense, then you run the program, and then you set the watchpoint. This example illustrates this procedure.

**Example**

```
MAIN
    DEFINE i INTEGER
```
LET i = 1
DISPLAY i
LET i = 2
DISPLAY i
LET i = 3
DISPLAY i
END MAIN

(fgldb) break main
breakpoint 1 at 0x00000000: file test.4gl, line 4
(fgldb) run
Breakpoint 1, main() at test.4gl:4
4      LET i = 1
(fgldb) watch i if i >= 3
Watchpoint 1:  i
(fgldb) continue
1
2
Watchpoint 1:  i

Old value = 2
New value = 3
main() at t.4gl:9
9         DISPLAY i
(fgldb)

whatis
The *whatis* command prints the data type of a variable.

**Syntax**

```plaintext
whatis variable-name
```

1. *variable-name* is the name of the variable.

**Usage**

The *whatis* command can be used to show the data type of a program variable.

The program variable must exist in the current scope.

**Example**

```plaintext
(fgldb) run
Breakpoint 1, main() at t.4gl:4
4      LET i = 1
(fgldb) whatis i
type = INTEGER
(fgldb)
```
Program profiler

Find out what function is causing the bottleneck in your program.

Understanding the profiler

The profiler is a tool built into the runtime system that allows you to know where your program spends processing time, and which function calls which function.

The profiler can help you to identify parts of your program that are executing slower than expected.

**Important:** The profiler measures CPU times. If the program is waiting (for a user-interaction, or when the program sleeps), then this has no impact on the results. In case of interactive programs, the time spent waiting for user action is ignored.

In order to enable the profiler during the execution of a program, you must start fglrun with the `-p` option, for example:

```
fglrun -p myprog
```

When the program ends, the profiler dumps profiling information to the standard error stream.

**Note:** The times reported by the profiler can change from one execution to the other, depending on the available system resources. Consider executing your program several times, to get an average time.

Syntax of the program profiler

The program profiler is enabled by using the `-p` option of fglrun.

To activate the program profiler, start the `fglrun` command with the `-p` option:

```
fglrun -p program [argument ...]
```

1. *program* is the name of the BDL program.
2. *argument* is a command line argument passed to the program.

Profiling statistics are collected during program execution, and printed when the program ends.

Profiler output: Flat profile

The flat profile shows a summary of the functions called during the program execution.

The flat profile contains a list of the functions called while the programs was running.

**Tip:** Runtime system internal function names start with the `rts_` prefix. For example, the `rts_display()` function implements the `DISPLAY` instruction.

<table>
<thead>
<tr>
<th>Table 455: Flat profile columns</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Column Name</strong></td>
</tr>
<tr>
<td>count</td>
</tr>
<tr>
<td><code>%total</code></td>
</tr>
<tr>
<td><code>%child</code></td>
</tr>
<tr>
<td><code>%self</code></td>
</tr>
<tr>
<td>name</td>
</tr>
</tbody>
</table>
Note: 100% represents the total processing time. The time spent waiting for user interaction is ignored.

Profiler output: Call graph

The profiler call graph provides detailed function call information.

The section "Call graph" provides the following details for each function:

1. The functions that called it, the number of calls, and an estimation of the percentage of time spent in these functions.
2. The functions called, the number of calls, and an estimation of the time that was spent in the subroutines called from this function.

Tip: Runtime system internal function names start with the rts_ prefix. For example, the rts_display() function implements the DISPLAY instruction.

Table 456: Call graph columns

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>Each function has an index which appears at the beginning of its primary line.</td>
</tr>
<tr>
<td>%total</td>
<td>Percentage of time spent in this function. Includes time spent in subroutines called from this function.</td>
</tr>
<tr>
<td>%self</td>
<td>Percentage of time spent in this function excluding the time spent in subroutines called from this function.</td>
</tr>
<tr>
<td>%child</td>
<td>Percentage of time spent in the functions called from this function.</td>
</tr>
<tr>
<td>calls/of</td>
<td>Number of calls / Total number of calls</td>
</tr>
<tr>
<td>name</td>
<td>Function name</td>
</tr>
</tbody>
</table>

Output example:

<table>
<thead>
<tr>
<th>index</th>
<th>%total</th>
<th>%self</th>
<th>%child</th>
<th>calls/of</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15.63</td>
<td>1.09</td>
<td>14.54</td>
<td>1/2</td>
<td></td>
<td>&lt;-- myprog.main</td>
</tr>
<tr>
<td>38.84</td>
<td>2.07</td>
<td>36.77</td>
<td>1/2</td>
<td></td>
<td>&lt;-- myprog.fB</td>
</tr>
<tr>
<td>[5]</td>
<td>54.47</td>
<td>3.16</td>
<td>51.31</td>
<td>2</td>
<td>*** myprog.fC</td>
</tr>
<tr>
<td>51.31</td>
<td>24.42</td>
<td>26.89</td>
<td>7/8</td>
<td></td>
<td>--&gt; mymod.fA</td>
</tr>
</tbody>
</table>

Description:

- Note that functions are prefixed with module names.
- The three stars *** indicate the function that is analyzed: myprog.fC.
- myprog.fC consumed 54.47% of the CPU time: 3.16% in the actual function code, plus 51.31% in the called functions.
- myprog.fC has been called two times (one time by myprog.main and a second time by myprog.fB).
- myprog.fC has called the mymod.fA function 7 times.
- mymod.fA has been called 8 times in the program.
Example

Program profiler example.

Source code

File: mymod.4gl

PUBLIC FUNCTION fA(s,n_a)
  DEFINE s STRING
  DEFINE n_a,i INTEGER
  FOR i=1 TO n_a
    DISPLAY "fA "||s||" n:"||i
  END FOR
END FUNCTION

File: myprog.4gl

IMPORT FGL mymod

MAIN
  DISPLAY "Profiler sample"
  CALL fB()
  CALL fC(2)
END MAIN

PRIVATE FUNCTION fB()
  CALL mymod.fA("fB",10)
  CALL fC(5)
END FUNCTION

PRIVATE FUNCTION fC(n_c)
  DEFINE n_c INTEGER
  WHILE n_c > 0
    CALL mymod.fA("fC",2)
    LET n_c=n_c-1
  END WHILE
END FUNCTION

Profiler output (stderr)

Flat profile (order by self)

<table>
<thead>
<tr>
<th>count</th>
<th>%total</th>
<th>%child</th>
<th>%self</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>90.5</td>
<td>42.9</td>
<td>47.6</td>
<td>mymod.fA</td>
</tr>
<tr>
<td>72</td>
<td>30.5</td>
<td>0.0</td>
<td>30.5</td>
<td>&lt;builtin&gt;.rts_Concat</td>
</tr>
<tr>
<td>25</td>
<td>10.1</td>
<td>0.0</td>
<td>10.1</td>
<td>&lt;builtin&gt;.rts_display</td>
</tr>
<tr>
<td>2</td>
<td>58.9</td>
<td>54.6</td>
<td>4.4</td>
<td>myprog.fC</td>
</tr>
<tr>
<td>8</td>
<td>3.1</td>
<td>0.0</td>
<td>3.1</td>
<td>&lt;builtin&gt;.rts_forInit</td>
</tr>
<tr>
<td>1</td>
<td>99.3</td>
<td>97.2</td>
<td>2.0</td>
<td>myprog.main</td>
</tr>
<tr>
<td>1</td>
<td>77.7</td>
<td>76.2</td>
<td>1.5</td>
<td>myprog.fB</td>
</tr>
</tbody>
</table>

Call graph (order by self)

<table>
<thead>
<tr>
<th>index</th>
<th>%total</th>
<th>%self</th>
<th>%child</th>
<th>calls/of</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>36.40</td>
<td>16.03</td>
<td>20.37</td>
<td>1/8</td>
<td>myprog.fB</td>
<td></td>
</tr>
<tr>
<td>51.31</td>
<td>24.42</td>
<td>26.89</td>
<td>7/8</td>
<td>myprog.fC</td>
<td></td>
</tr>
<tr>
<td>88.71</td>
<td>40.45</td>
<td>47.26</td>
<td>8</td>
<td>mymod.fA</td>
<td></td>
</tr>
<tr>
<td>28.07</td>
<td>28.07</td>
<td>0.00</td>
<td>72/72</td>
<td>&lt;builtin&gt;.rts_Concat</td>
<td></td>
</tr>
<tr>
<td>16.22</td>
<td>16.22</td>
<td>0.00</td>
<td>24/25</td>
<td>&lt;builtin&gt;.rts_display</td>
<td></td>
</tr>
<tr>
<td>2.97</td>
<td>2.97</td>
<td>0.00</td>
<td>8/8</td>
<td>&lt;builtin&gt;.rts_forInit</td>
<td></td>
</tr>
</tbody>
</table>

--------------
Source code coverage

Collect information about used source lines

Understanding the code coverage tool

This is an introduction to the code coverage tool.

The code coverage tool built in the fglrun runtime system allows you to collect information about source code execution.

With the coverage tool, you can:

- Identify dead code that is never executed, or make sure that important code is really executed.
- Find what lines of code are executed many times, to optimize your programs.

For each line of code, the tool reports the number of times it has been executed, if it has never been executed, or if it is not reachable at all.

**Important**: The output format of code coverage tool is for debug purpose only and can change in future product releases.

Syntax of the code coverage tool

The code coverage tool is enabled by setting the FGLCOV environment variable.

Enabling coverage data generation

To collect coverage data while executing programs, you must set the FGLCOV environment variable:

```
$ export FGLCOV=1
```
Merging coverage data files and source files

After producing coverage data files, these must be merged with source files by using the `fglrun --merge-cov` command:

```
fglrun --merge-cov name.4gl [name.42m.cov]
```

**Note:** Specify the path to the `.42m.cov` file, if this file and the corresponding `.42m` file are not located in the same directory as the `.4gl` source file.

Interpreting coverage indicators and execution counts

In the `.4gl.cov` result file, each source line gets a counting indicator, that must be interpreted as follows:

- `num` (a number): The line was executed `num` times.
- `-` (a dash): The line is not reachable.
- `=====`: The line is reachable, but has never been executed.

Code coverage tool usage

This topic describes how to use the code coverage feature.

Steps to get coverage information

Follow the next steps to collect and interpret coverage data:

1. Delete all existing coverage data files (*.42m.cov)
2. Enable coverage data file generation by setting the FGLCOV environment variable.
3. Execute your program several times (with `fglrun`): This will produce `module.42m.cov` files.
4. Use the `fglrun --merge-cov` command, to merge `module.42m.cov` coverage data files and `module.4gl` source files, to produce readable coverage information files (`module.4gl.cov`)
5. Inspect the `module.4gl.cov` to find how many times source code lines are executed.

Cleanup coverage data files

Before starting a new coverage data collection session, remove existing *.42m.cov data files.

```
$ rm -f *.42m.cov
```

Enable coverage data generation

Before starting a program, set the FGLCOV environment variable, to instruct `fglrun` to produce coverage data files:

```
$ export FGLCOV=1
```

In this topic, we will use the following program sample, to show how coverage data is produced:

**Note:** Pay attention to the line containing the `IF FALSE THEN` instruction: This will illustrate how unreachable lines are detected.

```
MAIN
  DEFINE i INTEGER
  FOR i=1 TO 10
    IF i MOD 2 == 0 THEN
      CALL func1()
    END IF
  END FOR
END MAIN

FUNCTION func1()
```
DISPLAY "Hello"
IF FALSE THEN
    DISPLAY "... world!"
END IF
END FUNCTION

A coverage data file (module.42m.cov) will be created for each module (module.42m), when running a program:

$ fglcomp prog.4gl
$ fglrun prog
Hello
Hello
Hello
Hello
Hello
Hello

$ ls prog.*
prog.42m  prog.42m.cov  prog.4gl

The coverage data file is created in the directory where the module is found.

**Important:** Coverage information is collected for several program executions: If a module.42m.cov file exist for a module, the execution count of source code lines will be summed up.

**Merge coverage data with source files**

The .42m.cov coverage data files contain only numbers. In order to interpret coverage data properly, you need to merge the coverage data files with the source file, by using the fglrun --merge-cov command:

$ fglrun --merge-cov prog.4gl
fglcov: prog.4gl.cov created

**Note:** If the .4gl source file and the .42m.cov coverage data file are not located in the same directory as the .42m module, specify the full path to the coverage data file as second parameter for the --merge-cov option:

fglrun --merge-cov mymodule.4gl ../dist/lib/mymodule.42m.cov

After a first time program execution, the resulting prog.4gl.cov file contains:

```
-:   1:MAIN
-:   2:   DEFINE i INTEGER
11:  3:       FOR i=1 TO 10
10:  4:           IF i MOD 2 == 0 THEN
 5:  5:               CALL func1()
-:  6:       END IF
-:  7:   END FOR
 1:  8:END MAIN
-:  9:
-: 10:FUNCTION func1()
 5: 11:       DISPLAY "Hello"
 5: 12:       IF FALSE THEN
======: 13:           DISPLAY "... world!"
-: 14:       END IF
 5: 15:END FUNCTION
```

In the above output, a couple of code coverage indicators can be interpreted as follows:

- Line #2 shows a – (dash) indicator: The DEFINE statement is not reachable, because it is a declaration statement.
• Line #3 shows the number 11: The FOR (head) instruction has been executed 11 times (10 + 1 = 11, to jump out of the loop)
• Line #4 shows the number 10: The IF statement inside the FOR loop has been executed 10 times.
• Line #13 shows =====: This line is reachable, but is never executed, since the parent IF condition always evaluates to FALSE.

By executing the sample program a second time, the existing coverage data files are completed with additional information:

$ fglrun prog

Merge again the data files with source files:

$ fglrun --merge-cov prog.4gl
fglcov: prog.4gl.cov created

And now check the new result file:

```
-:    1:MAIN
-:    2:   DEFINE i INTEGER
22:    3:   FOR i=1 TO 10
20:    4:       IF i MOD 2 == 0 THEN
10:    5:          CALL func1()
-:    6:       END IF
-:    7:   END FOR
2:    8:END MAIN
-:    9:
-:   10:FUNCTION func1()
10:   11:   DISPLAY "Hello"
10:   12:   IF FALSE THEN
========:   13:      DISPLAY "... world!"
-:   14:   END IF
10:   15:END FUNCTION
```

You can see that the execution counts of source code lines are summed up across program execution.

**Execution trace**

Print a function call stack of your program.

**Important:** Sensitive and personal data may be written to the output. Make sure that the log output is written to files that can only be read by application administrators.

**Understanding the program execution trace**

This is an introduction to the program execution trace.

The program execution trace prints all function calls of your program, with the values of parameters passed to and values returned from functions.

**Important:** Sensitive and personal data may be written to the output. Make sure that the log output produced by the runtime trace is written to files that can only be read by application administrators.

This tool can help to understand the code structure of complex programs, and identify bugs in your code.

**Important:** If you plan to use the program execution trace on a production site, pay attention to the amount of data produced by the log: Using the trace can fill up the disk and potentially crash the system in a very short time.

To enable the program execution trace, start fglrun with the `--trace` option, for example:

```
fglrune --trace myprog
```
The function call stack trace will be printed to the stderr stream:

```
Enter prog.main()
Enter prog.func1(12499, "abcdef") from prog.main at line 8
Enter prog.func2(12499, 1) from prog.func1 at line 16
Return 12500 from prog.func2 at line 23
...```

Note: The output format of FGL trace is for debug purpose only and can change in future product releases.

Additional control is available with FGLTRACE* environment variables, as described in the usage topic.

**Syntax of the program execution trace**

The execution trace is enable by using the `--trace` option of fglrun.

In order to enable program execution trace, start the `fglrun` command with the `--trace` option:

```
fglrun --trace program[.42r] [argument [...]]
```

1. `program` is the name of the BDL program.
2. `argument` is a command line argument passed to the program.

Function calls are printed to stderr during program execution.

By default, the trace starts in `MAIN`.

To define a list of functions to be traced, use the `FGLTRACE_FUNCTIONS` environment variable.

To exclude a set of functions from the trace, use the `FGLTRACE_EXCLUDE` environment variable.

**Using the program execution trace**

The program execution trace is typically used in a development environment, to understand the program flow.

**Data produced by the execution trace can be huge**

Important: The amount of data produced can fill up the disk and potentially crash the system in a very short time. Consider using this feature carefully, especially in production environments.

**Starting a program with trace**

Take the following code example:

```
MAIN
   DEFINE arr DYNAMIC ARRAY OF INTEGER,
   c INTEGER
   LET arr[1] = 123
   LET arr[2] = 456
   LET c = incr_elems(arr, 5)
END MAIN

FUNCTION incr_elems(arr, val)
   DEFINE arr DYNAMIC ARRAY OF INTEGER,
   val INTEGER
   DEFINE i INTEGER
   FOR i=1 TO arr.getLength()
      LET arr[i] = arr[i]+val
   END FOR
   RETURN arr.getLength()
END FUNCTION
```
In order to get a function call stack trace, start your program with the \texttt{--trace} option of fglrun:

\begin{verbatim}
fglrun --trace prog.42m
\end{verbatim}

The resulting trace printed to the stderr stream will look as follows:

\begin{verbatim}
Enter prog.main()
   Enter prog.incr_elems\{123,456\},5 from prog.main at line 6
   Enter base.Array.getLength\{123,456\} from prog.incr_elems at line 13
   Return 2 from base.Array.getLength
   Enter base.Array.getLength\{128,456\} from prog.incr_elems at line 13
   Return 2 from base.Array.getLength
   Enter base.Array.getLength\{128,461\} from prog.incr_elems at line 13
   Return 2 from base.Array.getLength
   Enter base.Array.getLength\{128,461\} from prog.incr_elems at line 16
   Return 2 from base.Array.getLength
   Return 2 from prog.incr_elems at line 16
Return from prog.main at line 7
\end{verbatim}

\textbf{Defining the functions to trace}

By default, the trace starts with the \texttt{MAIN} function and prints all subsequent functions calls of the program.

To define the functions where the trace should start, define the \texttt{FGLTRACE_FUNCTIONS} environment variable.

This variable takes a space-separated list of function names.

\textbf{Note:} In \texttt{FGLTRACE_FUNCTIONS}, the functions can be specified with or without their module prefix.

For example, with the following definition:

\begin{verbatim}
$ export FGLTRACE_FUNCTIONS="prog.incr_elems otherfunc1 otherfunc2"
\end{verbatim}

The new resulting trace will now start at \texttt{prog.incr_elems}:

\begin{verbatim}
Enter prog.incr_elems\{123,456\},5 from prog.main at line 6
   Enter base.Array.getLength\{123,456\} from prog.incr_elems at line 13
   Return 2 from base.Array.getLength
   Enter base.Array.getLength\{128,456\} from prog.incr_elems at line 13
   Return 2 from base.Array.getLength
   Enter base.Array.getLength\{128,461\} from prog.incr_elems at line 13
   Return 2 from base.Array.getLength
   Enter base.Array.getLength\{128,461\} from prog.incr_elems at line 16
   Return 2 from base.Array.getLength
   Return 2 from prog.incr_elems at line 16
Return from prog.main at line 7
\end{verbatim}

\textbf{Excluding functions from the trace}

By default, the trace includes also Genero BDL library APIs, such as \texttt{base.Array.getLength}.

Usually you are not interested in built-in function calls: You want to trace your own functions.

In order to exclude a complete set of methods, class or even a complete package, use the \texttt{FGLTRACE_EXCLUDE} environment variable, to define a space-separated list of exclude-patterns, using * and ? wildcards and [a-z] character ranges, as with the MATCHES operator.

\textbf{Note:} In \texttt{FGLTRACE_EXCLUDE}, user functions must be specified with their module prefix.

In our example, we want to exclude all methods from the build-in dynamic array class, and additionally we can also want to exclude all the calls to the classes of the om.* package (even if these are not used in this code example):

\begin{verbatim}
$ export FGLTRACE_EXCLUDE="<builtin>* base.Array.* om.*"
\end{verbatim}
Tip: To exclude all global built-in functions such as fgl_getenv(), use the <builtin>.* exclusion pattern.

The new resulting trace will now look as follows, focusing on the user function calls only:

Enter prog.incr elems((123,456),5) from prog.main at line 6
Return 2 from prog.incr elems at line 16

Mixing FGLTRACE_FUNCTIONS and FGLTRACE_EXCLUDE

When both FGLTRACE_FUNCTIONS and FGLTRACE_EXCLUDE are defined, FGLTRACE_FUNCTIONS has a higher priority than FGLTRACE_EXCLUDE:

- Any function listed in FGLTRACE_FUNCTIONS turns trace on
- Any function matching FGLTRACE_EXCLUDE turns trace off

The trace is enabled for a function listed in FGLTRACE_FUNCTIONS, when is it called from a function excluded by FGLTRACE_EXCLUDE. This might cause gaps in the output: an active call path can be disabled and enabled later.

Limited trace output

In order to avoid huge logs when the program creates large string variable or arrays, the output of values is voluntarily limited to a given size. In such case, the trace prints ellipsis (...), to indicate that the actual value is larger.

Try for example to change the MAIN block to fill the array with more elements:

```
MAIN
  DEFINE arr DYNAMIC ARRAY OF INTEGER,
    c INTEGER
  DEFINE i INTEGER
  FOR i=1 TO 100
    LET arr[i] = i
  END FOR
  LET c = incr elems(arr,5)
END MAIN
...
```

Compile and run the program again. The new trace output will show only the first elements of the array passed as parameter to the incr elems() function:

Enter prog.incr elems((1,2,3,4,5,6,7,8,9,10,11,...more...),..) from prog.main at line 8
Return 100 from prog.incr elems at line 18

Source code beautifier

Reformat the source code for better readability.

Understanding the code beautifier tool

This is an introduction to the source code beautifier tool.

The code beautifier tool built in the fglcomp compiler can be used to reformat .4gl source modules and get a readable source code, using consistent indentation and source layout rules across modules.

Note: The code beautifier supports uppercase or lowercase keyword coding conventions. See the --fo-lowercase-keywords option usage for more details.

Formatting options can be specified on the fglcomp command line, or in a configuration file (it is recommended to use the configuration file instead of commande line options).

The beautifier tool:
• Indents source lines for nested code blocks
• Converts language keywords to uppercase or lowercase
• Adds or removes space characters between tokens
• Breaks long lines
• Can use space or tab as indentation characters
• Can indent parts of the source file

By default, the new reformatted source code is written to the standard output stream (stdout). Use the `--fo-inplace` option to replace directly the original source file.

**Important:** The input source file must compile. Indenting a source file with errors will produce an unpredictable result.

In addition to the `fglcomp --format` tool, other command line tools are provided to integrate with version control systems such as GIT and SVN:

- `fglformatdiff` can reformat code line changes referenced by a `diff` command output.
- `fglitgitformat` can reformat code line changes referenced by a GIT commit. Use this tool instead of `fglformatdiff`, if GIT is your version control system.

### Syntax of the code beautifier tool

The source code beautifier option of `fglcomp` is `--format`.

#### Syntax

To beautify a source file, use the `--format` option of `fglcomp`:

```
fglcomp --format [format-option [...]]
```

1. `format-option` is a formatting option.

### Options

#### Table 457: Formatting options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`--fo-align-consecutive-assignments=0</td>
<td>1`</td>
</tr>
<tr>
<td>`--fo-align-consecutive-types=0</td>
<td>1`</td>
</tr>
<tr>
<td>`--fo-align-trailing-comments=0</td>
<td>1`</td>
</tr>
<tr>
<td><code>--fo-inplace</code></td>
<td>Write formatted output back to the provided file, instead of stdout. Creates a copy of the original file in <code>filename.4gl~</code></td>
</tr>
<tr>
<td><code>--fo-fallback-style=filename</code></td>
<td>Specify the configuration filename to be used if no <code>.fgl-format</code> file is found.</td>
</tr>
<tr>
<td><code>--fo-continuation-indent-width=integer</code></td>
<td>Define the source line width limit. Default is 80.</td>
</tr>
<tr>
<td><code>--fo-column-limit=integer</code></td>
<td>Number of columns to use for indentation. Default is 4.</td>
</tr>
<tr>
<td><code>--fo-indent-width=integer</code></td>
<td>Indent width for line continuations. Default is 4.</td>
</tr>
<tr>
<td>`--fo-label-indent=0</td>
<td>1`</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>--fo-pack=0</td>
<td>1</td>
</tr>
<tr>
<td>--fo-lowercase-keywords=0</td>
<td>1</td>
</tr>
<tr>
<td>--fo-use-tab=0</td>
<td>1</td>
</tr>
<tr>
<td>--fo-tab-width=integer</td>
<td>Defines the number of columns used for tab stop, when --fo-use-tab is specified. The default is 8 columns.</td>
</tr>
<tr>
<td>--fo-lines=start-line:end-line</td>
<td>Formats only the range of lines specified. This option can be used multiple times to specify several pieces of code to be reformatted.</td>
</tr>
</tbody>
</table>

**Configuration file**

Formatting options can be provided in a configuration file. It is strongly recommended to use a formatting configuration file instead of the command line options.

By convention, the name of the formatting configuration file is ".fgl-format".

The fglcomp --format tool will search for the .fgl-format file in the directory containing the source file passed as an argument, then in the parent directory of the source directory, and then its parent directory, and so on. The formatter will use the first .fgl-format file found. If no .fgl-format is found, fglcomp --format will read the file specified with the --fo-fallback-style=filename option.

**Tip:** Place the .fgl-format configuration file in the top directory of your project, to use the same formatting options for all sources located under this directory.

When using command line options and a configuration file, the compiler will combine formatting options from the configuration file and from the command line. When the same formatting option is specified in the configuration file and on the command line, the option of the command line takes precedence.

The formatting configuration file is a simple text file with lines defining formatting options. Lines starting with a hash character (#) are ignored.

**Note:** The --fo-inplace option must be specified as a command line parameter. It will not be read from the configuration file.

**Example:**

```
# My code formatting options
--fo-column-limit=80
--fo-indent-width=4
--fo-continuation-indent-width=4
--fo-label-indent=1
```

**Plugging the beautifier into VIM**

For vim users, it is possible to define function keys and commands to invoke the source code beautifier, and indent the current source file with a simple key stroke.

For more details about VIM configuration, see Configure VIM for Genero BDL on page 2123.
Code beautifier tool usage
This topic describes how to use the code beautifier tool.

Using the code beautifier of fglcomp
The code beautifier is a feature of the fglcomp compiler which is enabled with the --format option:

```
$ fglcomp --format module.4gl
```

Several formatting --fo-* options are available and explained below.

**Note:** The code beautifier supports uppercase or lowercase keyword coding conventions. See the --fo-lowercase-keywords option usage for more details.

Using a configuration file
All --fo-* formatting options (except --fo-inplace) can be specified in a configuration file named .fgl-format in the source directory or in one of the parent directories of the source.

If no .fgl-format is found, fglcomp --format will read the file specified with the --fo-fallback-style=filename option.

This configuration file is typically placed in the top directory of your project, to apply the same formatting rules to all sources located under this root directory:

```
$ tree -a .
.
  |-- top
  |   `-- .fgl-format
  |       |-- appl
  |          `-- main.4gl
  `-- lib
      `-- myutils.4gl
3 directories, 3 files
$ cd top
$ cat .fgl-format
--fo-indent-width=5
--fo-pack=1
$ cd lib
$ fglcomp --format myutils.4gl
PRIVATE DEFINE debug_level SMALLINT
FUNCTION set_debug_level(l SMALLINT)
    LET debug_level = l
END FUNCTION
```

Controlling the beautifier result output
By default, the reformatted source is written to the standard output stream (stdout).

To overwrite the original source file, use the --fo-inplace option:

```
$ fglcomp --format --fo-inplace module.4gl
```

Before reformatting the source file with the --fo-inplace option, fglcomp creates a copy of the original source file, by adding a tilde character after the .4gl file extension: filename.4gl~.

**Note:** The --fo-inplace option cannot be used in the .fgl-format configuration file.
Disabling formatting for a group of lines

The formatter understands special control comments that disable formatting for a block of source code lines. The code between single-line comments having "fgl-format off/on" keywords will not be formatted. The formatting control comments can use the --, # or { } comment syntax, and must appear on a line by itself.

Note: The control comment lines themselves will be formatted and aligned, depending on the surrounding source code lines.

For example:

```
MAIN
    DEFINE formatted INT = 0
    -- fgl-format off
    define unformatted int=1
    -- fgl-format on
    DEFINE formatted_again INT = 2
END MAIN
```

Tip: Make sure that the formatting control comments are placed in the same source code block and scope. For example, if you start with an fgl-format off control comment just before a FOR loop, and you put the closing fgl-format on control comment inside the FOR loop, the formatter will produce unexpected alignments.

 Formatting a range of lines

By default, the complete source file is reformatted. Use the --fo-lines=start-line:end-line option to specify a range of lines to be formatted.

The --fo-lines option is especially useful with version control systems, to limit the commit change to the piece of code that is updated and then reformatted.

To format only a range of lines of a source file, use the --fo-lines option by specifying the starting and ending line:

```
$ fglcomp --format --fo-lines=51:57 module.4gl
```

Note: The --fo-lines option can be used multiple times to specify several pieces of code to be reformatted.

Defining the maximum width for a source code line

Use the --fo-column-limit=cols option to specify the width limit for a source code line:

```
$ cat module.4gl
PUBLIC FUNCTION update_customer( id INTEGER, name VARCHAR(50) )
    UPDATE customer SET cust_name = name WHERE cust_id = id AND cust_valid = 'Y'
END FUNCTION

$ fglcomp --format --fo-column-limit=40 module.4gl
PUBLIC FUNCTION update_customer( id INTEGER, name VARCHAR(50))
    UPDATE customer
    SET cust_name = name
    WHERE cust_id = id
    AND cust_valid = 'Y'
END FUNCTION
```

The default is 80.
Defining the base indentation width

The \texttt{--fo-indent-width=cols} option can be used to set width for base indentations:

$$\text{cat module.4gl}$$
\begin{verbatim}
PUBLIC FUNCTION cleanup( mode INTEGER )
  CASE
    WHEN mode==1 DISPLAY "Mode 1"
    WHEN mode==2 DISPLAY "Mode 2"
  END CASE
END FUNCTION
\end{verbatim}

$$\text{fglcomp --format --fo-indent-width=6 module.4gl}$$
\begin{verbatim}
PUBLIC FUNCTION cleanup(mode INTEGER)
  CASE
    ^^\hphantom{6} -- 6 spaces from \texttt{--fo-indent-width}
    WHEN mode == 1
    ^^\hphantom{6} -- 6 spaces from \texttt{--fo-indent-width}
      DISPLAY "Mode 1"
    WHEN mode == 2
      DISPLAY "Mode 2"
  END CASE
END FUNCTION
\end{verbatim}

The default is 4.

Defining the continuation indentation width

The \texttt{--fo-continuation-indent-width=cols} option defines the indentation width for line-breaks. When breaking long lines, this number of spaces inserted for each extra level:

$$\text{cat module.4gl}$$
\begin{verbatim}
PUBLIC FUNCTION update_customer( id INTEGER, name VARCHAR(50) )
  UPDATE customer SET cust_name = name WHERE cust_id = id AND cust_valid = 'Y'
END FUNCTION
\end{verbatim}

$$\text{fglcomp --format --fo-column-limit=40 --fo-continuation-indent-width=2 module.4gl}$$
\begin{verbatim}
PUBLIC FUNCTION update_customer(
  id INTEGER, name VARCHAR(50))
  ^^\hphantom{2} -- 2 spaces from \texttt{--fo-continuation-indent-width}
  UPDATE customer
  ^^\hphantom{4} -- 4 spaces from \texttt{--fo-indent-width default}
    SET cust_name = name
    ^^\hphantom{2} -- 2 spaces from \texttt{--fo-continuation-indent-width}
      WHERE cust_id = id
    ^^\hphantom{2} -- 2 spaces from \texttt{--fo-continuation-indent-width}
      AND cust_valid = 'Y'
    ^^\hphantom{2} -- 2 spaces from \texttt{--fo-continuation-indent-width}
END FUNCTION
\end{verbatim}

Default is 4.

Using TAB as indentation character

By default, the indentation character is a blank space: It is used all over to produce indentation.

Specify the \texttt{--fo-use-tab=1} formatter option, to produce tabs whenever it is needed to fill whitespace that spans at least from one tab stop to the next one, based on \texttt{--fo-tab-width} and \texttt{--fo-indent-width} options.

Use the \texttt{--fo-tab-width=cols} option to define the number columns used for tab stops.
For example, with the default --fo-indent-width of 4 columns and the default --fo-tab-width of 8 columns, --fo-use-tab=1 will put tabs whenever 8 columns tab stops are found, and put 4 spaces to complete the indentation:

```sh
$ fglcomp --format --fo-use-tab=1 module.4gl
PUBLIC FUNCTION cleanup(mode INTEGER)
  CASE
    ^^^^ -- 4 spaces
    < tab > -- 1 tab
    < tab >^^^^ -- 1 tab + 4 spaces
    WHEN mode == 1
    DISPLAY "Mode 1"
    WHEN mode == 2
    END CASE
  END CASE
END FUNCTION
```

The next example produces only tabs, because --fo-tab-width is set to 4 and matches the default --fo-indent-width:

```sh
$ fglcomp --format --fo-use-tab=1 --fo-tab-width=4 module.4gl
PUBLIC FUNCTION cleanup(mode INTEGER)
  CASE
    WHEN mode == 1
    < tab > < tab > -- 2 tabs
    DISPLAY "Mode 1"
    WHEN mode == 2
    END CASE
  END CASE
END FUNCTION
```

**Controlling instruction clause indentation**

The --fo-label-indent=1 option indicates that indentation should apply to instruction sub-clauses, such as the WHEN clause of a CASE / END CASE block. When zero, no sub-clause indentation is done:

```sh
$ cat module.4gl
PUBLIC FUNCTION cleanup( mode INTEGER )
CASE
  WHEN mode==1 DISPLAY "Mode 1"
  WHEN mode==2 DISPLAY "Mode 2"
END CASE
END FUNCTION

$ fglcomp --format --fo-label-indent=0 module.4gl
PUBLIC FUNCTION cleanup(mode INTEGER)
CASE
  WHEN mode == 1
    DISPLAY "Mode 1"
  WHEN mode == 2
    DISPLAY "Mode 2"
END CASE
END FUNCTION
```

When 0, the instruction clauses are aligned with the head of the statement (CASE, INPUT etc). If 1, those "labels" are indented. This results in indenting the statements following those labels 2 levels relatively to the head of the statement.

The default is 1 (clause indentation is enabled).
Controlling how source lines are packed

The `--fo-pack=1` option can be used to pack items as much as possible together on the same line:

```
$ cat module.4gl
PUBLIC FUNCTION func1( )
CALL func2( "aaa", "bbb", "ccc", "ddd", "eee" )
END FUNCTION

$ fglcomp --format --fo-column-limit=30 module.4gl
PUBLIC FUNCTION func1()
   CALL func2( 
       "aaa",
       "bbb",
       "ccc",
       "ddd",
       "eee")
END FUNCTION

$ fglcomp --format --fo-column-limit=30 --fo-pack=1 module.4gl
PUBLIC FUNCTION func1()
   CALL func2( 
       "aaa", "bbb", "ccc",
       "ddd", "eee")
END FUNCTION
```

Using uppercase or lowercase keywords

Genero BDL uses traditionally uppercase keywords and therefore the code formatter produces uppercase keywords by default.

When the coding conventions require lowercase keywords, use the `--fo-lowercase-keywords=1` option of the formatter, to produce lower case keywords:

```
$ cat module.4gl
function myfunc(x INT) returns int
   return x + 1
end function

$ fglcomp --format --fo-lowercase-keywords=1 module.4gl
function myfunc(x int) returns int
   return x + 1
end function

$ fglcomp --format --fo-lowercase-keywords=0 module.4gl
FUNCTION myfunc(x INT) RETURNS INT
   RETURN x + 1
END FUNCTION
```

Aligning consecutive items

Several formatting options are provided to align consecutive language elements.

The `--fo-align-consecutive-assignments=1` option aligns consecutive assignments:

```
LET name = "Phil"
LET da   = get_last_date()
LET id   = 1032

LET rec.cust_id   = 1243
LET rec.cust_name = "McTough"
```
LET rec.cust_address = "5 Sunset Av."

Note: Alignment never produces additional linebreaks. The formatter produces groups of aligned lines, for example (with \texttt{--fo-align-consecutive-assignments=1} and \texttt{--fo-column-limit=40}): 

\begin{verbatim}
LET long_var_name_1 = 1 
LET very_long_var_name_2 = 1 
LET short_name = "aaaaaaaaaaaaaaaaa" 
LET tn = "aaaaaaaaaaaaaaaaa"
\end{verbatim}

The \texttt{--fo-align-consecutive-types=1} option aligns types in consecutive variable definitions:

\begin{verbatim}
DEFINE v1 INTEGER
DEFINE current_date DATE
DEFINE last_name VARCHAR(50)

DEFINE v2 INTEGER
DEFINE rate DECIMAL(10, 2)
\end{verbatim}

The \texttt{--fo-align-trailing-comments=1} option aligns consecutive comments:

\begin{verbatim}
DISPLAY a -- first comment
DISPLAY foo, -- second comment
    bar -- third comment
\end{verbatim}

The \texttt{fglformatdiff} command

The \texttt{fglformatdiff} command reads a unified diff from the (stdin) standard input stream, calculates what lines have changed and calls the formatter to beautify the modified lines.

The \texttt{fglformatdiff} command is provided to integrate with version control systems such as CVS and SVN. If GIT is your version control system, use \texttt{fglgitformat} instead.

The diff output must be unified and without any context lines (\texttt{diff -U0}).

The following example uses a diff output produced from the \texttt{svn} tool (note the options to get a unified diff format):

\begin{verbatim}
svn diff --diff-cmd=diff -x-U0 | fglformatdiff --inplace
\end{verbatim}

See also \texttt{fglformatdiff} command reference.

The \texttt{fglgitformat} command

The \texttt{fglgitformat} command can reformat code changes that are tracked in a GIT repository.

For example, the next command reformats the lines of code changes registered in the 2 last commits (\texttt{HEAD~2}), modifies directly the related sources (\texttt{--inplace}), and prints the changes to the stdout stream (\texttt{--verbose}): 

\begin{verbatim}
fglgitformat --verbose --inplace HEAD~2
\end{verbatim}

See also \texttt{fglgitformat} command reference.
Packaging web applications

Describes methods of packaging the runtime files and resources of your web applications and services using the fglgar tool.

Introducing the GAS and JGAS

The Genero Application Server (GAS) is an engine that plugs in to a Web server for the purpose of delivering Genero Web applications and services. The Genero Application Server for Java (JGAS) is designed to run your applications on the Java EE servlet. A general knowledge of how they operate can be helpful in testing and deploying Web applications.

The GAS manages the Genero Browser Client or the Genero Desktop Client application requirements of the Web server. It uses dispatchers and proxies to optimize reliability, performance, and integration in Web servers. The dispatcher handles the GAS configuration and keeps a persistent session table of all proxies for the Dynamic Virtual Machine (DVM) or runtime it starts.

In development environments, it is possible to exclude the Web Server and run applications as standalone on the GAS. The httpdispatch is the standalone dispatcher used to connect to applications from the local machine.

From version 3.10 onwards the JGAS, a GAS written in Java, can be packaged in a Java Web Archive (.war) file. Applications can be run as standalone without the need for a separate GAS installation or a Java Enterprise Edition (Java EE) server.

Gas features

These features are common to both the GAS and JGAS:

- Connections between the DVM and the front-end are handled by the GAS in a one-to-one relationship. It manages an application session and provides a debugging and logging mechanism for dispatchers and proxies. Session information is saved in case the Web server fails, which allows for the application to continue when the server restarts.
- It features a command line tool (gasadmin) that can perform several administrative tasks. This includes stopping a DVM (if required) without affecting any other applications that are running concurrently on the server.
- It supports a timeout feature called AUTO_LOGOUT which can be configured to log out a user and display a log out page after a specified time of user inactivity on a GUI client.
- It allows for user authorization and authentication to be implemented via delegation and single sign-on.
- A GAS installation is required if you provide Genero Web Services (GWS) servers. The GAS manages a pool of connections for clients accessing your Web services.
- Applications that are deployed to run on a GAS on a Web server can be easily configured based on default application configurations and resources defined in the GAS configuration file as.xcf.

The GAS is usually installed on the same machine where the Genero BDL runtime is installed. For more information, refer to the Genero Application Server User Guide.

JGAS features

The JGAS is an implementation of the GAS that is packaged with the FGLGWS product. It performs the same functions as the GAS and can be used for the development, testing, and deployment of applications.

It differs from the standard GAS in the way it uses Web server resources such as sockets and memory in the handling of HTTP communication between the DVM and the front-end. Processes are similar except that proxies are not used between the DVM and the dispatcher. HTTP requests are processed internally and one socket per DVM is all that is required to maintain the communication. For more information see the JGAS overview page in Genero Application Server for Java User Guide.
Using the JGAS

The fglgar on page 2090 tool provides you with access to the JGAS and a means to package, deploy, and run applications.

The fglgar war command provides the ability to package a Genero Archive (.gar) file with your applications, and to include the JGAS implementation, the Genero Browser Client, and any additional files in a Java Web Archive (.war) file. The Java Web Archive (.war) file can be deployed in any existing Java Enterprise Edition container such as Apache Tomcat®, Jetty, or Glassfish and the applications can be run in a browser.

**Related concepts**

Packaging war files on page 2189
Using the fglgar tool to build a Java Web Archive (.war) file allows you to deploy applications that are ready to run.

Running applications from a war file on page 2190
Applications packaged in a Java Web Archive (.war) file can be executed with the fglgar tool's run command.

**Related tasks**

Deploying applications on GAS on page 2185
Follow these procedures to build archives for deploying applications and services with fglgar.

Understanding packaging with GAS

Before deploying applications on a GAS installation, you need to package the required compiled files and resources in a Genero archive file. The fglgar command provides this function and you can find out here what is involved in this process.

Multiple applications may be bundled in a gar file. As a prerequisite, all of the required files and resources must be in a root directory. When you run the fglgar tool with the gar command a Genero archive is created.

```bash
fglgar gar --application helloWorld.42r --service welcomeService.42m
```

**Example gar file**

The gar file contains a MANIFEST file, your application modules, form files, configuration files, for example:

- MANIFEST
- helloWorld.42m
- helloWorld.42r
- helloWorld.42f
- welcomeService.42f
- helloWorld.xcf
- welcomeService.xcf

**The MANIFEST file**

The MANIFEST is an XML file that essentially provides a list of the applications and services in the gar to make available. It can be created automatically by fglgar at the command line for applications specified with the --application and/or --service option.

Or alternatively, if you have many applications to package, you may find it easier to first create the MANIFEST by hand. Running fglgar (without the --application and/or --service option) it checks if a MANIFEST file is present in the directory, and uses it to create the gar. For more information see the MANIFEST file topic in the Genero Application Server User Guide.

**Application configuration files**

You can provide the application configuration (.xcf) files if you wish but they are created automatically for you if you provide the --application and/or --service options with the name of the executable (42r or 42m) files instead of xcf files.
The xcf file is created based on default configurations in the GAS configuration file, as.xcf:

- With option `--application`, fglgar creates a default xcf based on the "defaultwa" configuration.
- With option `--service`, fglgar creates a default xcf based on "ws.default" configuration.

In both cases, the generated xcf file is given the name of the 42r or 42m provided at the command line.

**Deploy the gar file and enable its applications**

You deploy the archive on your GAS installation locally using the `gasadmin` tool, or the FGL Web services PublishGAR tool that is provided in the Genero BDL installation.

**Note:** If the GAS is on a remote Web server, you can only deploy applications with PublishGAR. You can also use other tools such as curl, etc.

These tools unpack the contents of the archive in the GAS `$(res.appdata.path)/deployment` path. The applications deployed are identified by the name of the xcf. To make the applications available to end users, enable the archive using either the `gasadmin` or the PublishGAR tool. For examples, see Deploying applications on GAS on page 2185.

**Related concepts**

Introducing the GAS and JGAS on page 2183

The Genero Application Server (GAS) is an engine that plugs in to a Web server for the purpose of delivering Genero Web applications and services. The Genero Application Server for Java (JGAS) is designed to run your applications on the Java EE servlet. A general knowledge of how they operate can be helpful in testing and deploying Web applications.

**Deploying applications on GAS**

Follow these procedures to build archives for deploying applications and services with fglgar.

As a prerequisite, you must have your applications and/or services created, compiled, and tested. Consolidate all the necessary files for your archive under a root directory.

Included in this page are some typical examples of how to build archives and deploy applications:

- Overview of the main tasks on page 2185
- Build a simple archive file on page 2186
- Build an archive file with many applications on page 2187
- Build an archive with public resources on page 2187
- Build an archive with deployment triggers on page 2188
- Deploy your application on your machine on page 2188
- Enable your application on your machine on page 2189
- Run the deployed application on page 2189

For a full understanding of what Genero archiving offers, please read all archiving topics in the Genero Application Server User Guide.

**Overview of the main tasks**

This procedure provides you with a quick overview of the main steps for archiving and deploying an application using fglgar. For more information see fglgar on page 2090.

1. Create the archive file using the `fglgar gar` command.

   If you do not have application configuration files created, they are created automatically for you. See Understanding packaging with GAS on page 2184.

   ```bash
   fglgar gar --application helloWorld.42r --service welcomeService.42m
   ```
This example shows the same command referencing xcf files you have already created for your application and service.

```
fglgar gar --application helloWorld.xcf --service welcomeService.xcf
```

A gar file is created that has the same name as your current directory. If you wish to specify a name for your archive, use the --output option.

2. Deploy the archive file.

   - If you are on the application server, you can deploy the archive with the gasadmin tool or the FGL Web services PublishGAR tool. Examples are shown for each method:

```
gasadmin gar --deploy-archive myApp.gar
```

```
fglrun
$FGLDIR/web_utilities/services/deployment/bin/PublishGAR
http://localhost:6394 deploy myApp.gar
```

**Note:** This example assumes the standalone dispatcher on your GAS installation is already started on your local machine and you have not changed the default port number.

   - If the GAS is on a remote Web server, you can deploy applications with PublishGAR. In this example, the gar is deployed at the base URL of the GAS (for example http://zeus:8090/gas) on the Web server.

```
fglrun
$FGLDIR/web_utilities/services/deployment/bin/PublishGAR
http://zeus:8090/gas deploy myApp.gar
```

The archive is unpacked in the $(res.appdata.path)/deployment directory.

**Note:** The application deployed is named with the xcf, not the gar file name.

3. Enable the archive.

   This makes applications in the archive available to end users.

   - If you are on the application server, you can enable the archive with the gasadmin tool or the FGL Web services tool, PublishGAR. Examples are shown for each method:

```
gasadmin gar --enable-archive myApp.gar
```

```
fglrun
$FGLDIR/web_utilities/services/deployment/bin/PublishGAR
http://localhost:6394 enable myApp.gar
```

```
fglrun
$FGLDIR/web_utilities/services/deployment/bin/PublishGAR
http://zeus:8090/gas enable myApp.gar
```

The applications, services, and resources included in the archive are available for your end users.

### Build a simple archive file

The fglgar gar command used with the --application option builds an archive for your application. Using fglgar as shown in this procedure creates this type of archive.

Create the archive file (gar).
For example, at the command line type the following:

```
fglgar gar --application myApp.42r
```

A `gar` file is created that has the same name as your current directory. A `MANIFEST` file and a configuration file is created automatically and included in the archive.

**Warning:** If a `MANIFEST` file already exists when you run the command, errors will be raised. For more information on the use of these options, see `fglgar` on page 2090.

**Build an archive file with many applications**

If you have many applications to package, you may find it easier to create the `MANIFEST` by hand instead of referencing each application at the command line. The `fglgar` checks if a `MANIFEST` file is present in the archive directory and uses it to create the `gar`.

1. Create your own `MANIFEST` file. See the `MANIFEST file` topic in the *Genero Application Server User Guide*.
2. Create your own application configuration files (`xcf`).
   Make sure to set the `PATH` element to `$(res.deployment.root)`. For example, if your compiled files (forms, modules, and so on) were in the `/bin` directory of your archive, you would specify the `PATH` element as shown:

   ```
   <PATH>$(res.deployment.root)/bin</PATH>
   ```

   For more information, see the *Configuring applications on GAS* chapter in the *Genero Application Server User Guide*.
3. Use the `fglgar` tool to create a Genero Archive.

   If you are in the directory containing your `MANIFEST` file and your program files:

   ```
   fglgar gar
   ```

   This creates an archive (`gar`) file with the same name as your current directory.

   If you need to specify the directory where the archive content is located, include the `--input-source` option:

   ```
   fglgar gar --input-source ./myArchiveDir
   ```

   This creates an archive file with the same name as the archive directory, drawing its content from the `./myArchiveDir` directory.

   If you wish to specify a name for your archive, use the `--output` option:

   ```
   fglgar gar --input-source ./myArchiveDir --output myApp.gar
   ```

   This creates an archive file with the name `myApp.gar`, drawing its content from the `./myArchiveDir` directory.

**Build an archive with public resources**

An archive can contain common or public resources such as images, reports, etc., that all deployed applications on the GAS can use. Using `fglgar` as shown in this procedure creates this type of archive.

1. Put your application's public images in a dedicated directory of your archive directory.
   You can name it, for example, "myAppPublicImages".
2. Create the archive file (`gar`).
For example, at the command line type the following command with the --resource option specifying the name of the resource directory in the archive directory:

```bash
fglgar gar --resource myAppPublicImages --application myApp.xcf
```

**Note:** A general knowledge of how the Genero Browser Client (GBC) operates can be helpful in the planning and deploying of Web applications. For more information, refer to the *Genero Browser Client User Guide*.

### Build an archive with deployment triggers

An archive can be defined with deployment parameters. These are commands that execute when deploying and undeploying an application on the GAS. Using `fglgar` as shown in this procedure creates the archive with deployment trigger options.

**Note:** Deployment triggers are typically not required, you can deploy your applications without them.

Create an archive file (`gar`).

For example:

```bash
fglgar gar --application myApp.xcf --trigger-component cpn.gar.execution.local --deploy-trigger "fglrun mydeploy.42r" --undeploy-trigger "fglrun myundeploy.42r"
```

Where:

- The `--trigger-component` option references a trigger component in the GAS as.xcf that defines the runtime environment where triggers are run. If not set, it defaults to "cpn.gar.execution.local".
- The `--deploy-trigger` specifies your `DEPLOY` command.
- The `--undeploy-trigger` options specifies the `UNDEPLOY` command.

These commands are saved in a `MANIFEST` file in the `gar`. See the *Triggers* page in the *Genero Application Server User Guide*.

**Warning:** If a `MANIFEST` file already exists when you run the command, errors will be raised. For more information on the use of these options, see `fglgar` on page 2090.

### Deploy your application on your machine

**Before your begin:**

Once you have created an archive for your application, you can now deploy it locally on your machine where the GAS is installed. If the standalone dispatcher on your GAS installation is not already started, run it from the command line using `httpdispatch`.

Deploy your `gar` file.

For example, to deploy an archive named `myApp.gar` using the `PublishGar` tool, type the following:

```bash
fglrun
$FGLDIR/web_utilities/services/deployment/bin/PublishGAR
http://localhost:6394  deploy myApp.gar
```

Or if using the `gasadmin` tool, type the following command:

```bash
gasadmin gar --deploy-archive myApp.gar
```

The archive is unpacked in the `$res.appdata.path)/deployment` directory.

**Note:** The application deployed is named with the `xcf`, not the `gar` file name.
Enable your application on your machine

Once you have deployed your application, you can now make it available for end users by enabling it on the machine where the GAS is installed. If the standalone dispatcher on your GAS installation is not already started, run it from the command line using httpdispatch.

Enable your application.

For example, using the PublishGAR tool, type the command

```
fglrun
$FGLDIR/web_utilities/services/deployment/bin/PublishGAR
http://localhost:6394 enable myApp.gar
```

Or if using the gasadmin tool, type the command:

```
gasadmin gar --enable-archive myApp.gar
```

Run the deployed application

Once the application is enabled, you can now run it. If the standalone dispatcher on your GAS installation is not already started, run it from the command line using httpdispatch.

In a browser enter the address of your deployed application.

```
http://localhost:6394/ua/r/myApp
```

In this example, the URL is looking for a configuration file named myApp.xcf.

If your application is displayed and you can interact with it, you have successfully deployed an application.

Packaging war files

Using the fglgar tool to build a Java Web Archive (war) file allows you to deploy applications that are ready to run.

The fglgar tool run with the war command creates a war file that embeds a Genero Java GAS (JGAS) and the Genero Archive file (gar). The JGAS allows you to run applications and services bundled in your gar file as standalone. This means that you can package applications for deployment and testing without the need to install an additional GAS package on the server side.

Note: The standalone JGAS is for development and testing only, a Java Enterprise Edition (Java EE) server is required for a production environment.

A typical fglgar war command is shown:

```
fglgar war -g MyApp.gar --gbc c:/dev/gbc_custom -o ../MyGeneroJavaApps.war
```

The example uses three of the command's main options:

- The `--input-gar (-g)` option is mandatory. It allows you to specify the Genero Archive you want to use to create the war file. This means that you need to have a gar file already created to include in the war file. See Deploying applications on GAS on page 2185.
- The `--gbc` option is optional but it allows you specify a different Genero Browser Client (GBC), for example, one that you have customized specifically for your applications. Otherwise, the default gbc installed with the FGLGWS package is embedded in the war file.
- The `--output` or `-o` option specifies the relative or absolute path to the war file to generate.

Related concepts

Introducing the GAS and JGAS on page 2183

The Genero Application Server (GAS) is an engine that plugs in to a Web server for the purpose of delivering Genero Web applications and services. The Genero Application Server for Java (JGAS) is designed to run your applications
on the Java EE servlet. A general knowledge of how they operate can be helpful in testing and deploying Web applications.

Running applications from a war file
Applications packaged in a Java Web Archive (.war) file can be executed with the fglgar tool’s run command.

What is the fglgar run command?
The fglgar run command starts the JGAS that is packaged in the specified .war file.
The command opens the port to listen for incoming requests, to launch GUI applications in a browser or to respond to Web services requests.

Important:
• FGLGWS needs to be installed on the server to interpret the Genero 4GL applications and services.
• You need a Java Runtime Environment (JRE) version that is at least version 1.8 or greater.
The next example shows fglgar run command using the MyGeneroJavaApps.war package:

```
fglgar run -w MyGeneroJavaApps.war -p 9999 -P 4444 -E log.file.path=`pwd`
```

To understand how to create a .war file, see Packaging war files on page 2189.

fglgar run options
The main fglgar run options are:
• The --war or -w option is mandatory and defines the war file.
• The --http-port or -p option defines the HTTP port. If not set, the default port is 8080.
• The --https-port or -P option defines the HTTPS port. If not set, the default port is 443.
• The --resource-overwrite or -E option defines the GAS resources you want to overwrite, such as the directory of the logs.

See fglgar command reference for more options.

Starting applications from a browser
If, for example, your MyGeneroJavaApps.war file contains two Web application, one called HelloWorld and another called MyApp, the URL requests to launch them then from the browser would look like this:

```
http://localhost:9999/MyGeneroJavaApps/HelloWorld/ua/r/HelloWorld
http://localhost:9999/MyGeneroJavaApps/MyApp/ua/r/MyApp
```

Tip: The base URL in the context of JGAS is http://server:port/war_file_name. If you type the base URL in a browser, you get an overview page where you can then access all applications and services published in your JGAS.

Getting Web Services WSDLs
To request the Web Services Descriptive Language (WSDL) of a Web service called MyWebService deployed in the MyGeneroJavaApps.war, the URL must look like this:

```
http://localhost:9999/MyGeneroJavaApps/MyWebService/ws/r/MyWebService?WSDL
```

Related concepts
Introducing the GAS and JGAS on page 2183
The Genero Application Server (GAS) is an engine that plugs in to a Web server for the purpose of delivering Genero Web applications and services. The Genero Application Server for Java (JGAS) is designed to run your applications
Programming tools | 2191

on the Java EE servlet. A general knowledge of how they operate can be helpful in testing and deploying Web applications.

Packaging examples

With the fglgar tool you can immediately see the results of a Genero Web Application or Web Service during your development stage without having to install a Web server or a GAS package.

Packaging and running applications with fglgar

If you have a directory named MyFirstApp with a Genero 4GL application and/or a service that you wish to test, then with just three fglgar commands applications are packaged and can be viewed in a browser.

```bash
$ cd MyFirstApp
$ fglcomp -M HelloWorld.4gl
$ fglcomp -M MyWebService.4gl
$ fglgar gar --application HelloWorld.42m --service MySebService.42m
$ fglgar war --input-gar MyFirstApp.gar
$ fglgar run --war MyFirstApp.war -p 9999
```

Where:

1. The application and service source files (4gl) are compiled.
2. fglgar is run with the gar command to create an archive file with the compiled applications and services.
3. fglgar is run with the war command to create a war file that embeds the gar file together with the GAS written in Java (JGAS) that allows applications to be run as standalone.
4. fglgar is run with the run command to start the JGAS packaged in the war file. It opens port 9999 to listen for incoming requests to launch applications or Web services. See Running applications from a war file on page 2190.

Adding a customized GBC to the package

If you have customized Genero Browser Client (GBC) in a directory named gbc_customized, then you can include it in the package with the --gbc option.

```bash
$ cd MyFirstApp
$ fglcomp -M HelloWorld.4gl
$ fglcomp -M MyWebService.4gl
$ fglgar gar --application HelloWorld.42m --service MySebService.42m
$ fglgar war --input-gar MyFirstApp.gar --gbc c:\dev\gbc_customized
$ fglgar run --war MyFirstApp.war -p 9999
```

Packaging multiple applications

If you have multiple applications to deploy in your MyApps directory that you wish to test, you can add them to the fglgar gar with --application options, as shown. The applications are bundled in the gar and run from the war file.

```bash
$ cd MyApps
$ fglcomp -M HelloWorld.4gl
$ fglcomp -M appl.4gl
$ fglgar gar --application HelloWorld.42m --application appl.42m
$ fglgar war --input-gar MyApps.gar
$ fglgar run --war MyApps.war -p 9999
```

Related concepts

Introducing the GAS and JGAS on page 2183

The Genero Application Server (GAS) is an engine that plugs in to a Web server for the purpose of delivering Genero Web applications and services. The Genero Application Server for Java (JGAS) is designed to run your applications
on the Java EE servlet. A general knowledge of how they operate can be helpful in testing and deploying Web applications.

Logging options

Logging solutions allow you to display exchanges between components when a program executes.

Genero provides several build-in options to get debug information, as well as logging features:

- Get the stack trace with `base.Application.getStackTrace()`
- Display the GUI protocol exchange in stderr with `FGLGUIDEBUG`.
- Display the SQL statements execution in stderr with `FGLSQLDEBUG`.
- Display Web Services API calls in stderr with `FGLWSDEBUG`.
- Produce function call stack trace, with `fglrun --trace` option.
- Produce source code coverage info with `fglrun --merge-cov / FGLCOV`.
- Log front-end protocol exchange with `fglrun --start-guilog` option.
- Produce application log files in case of runtime error, with `STARTLOG()`.

**Important:** Sensitive and personal data may be written to the output. Make sure that the log output is written to files that can only be read by application administrators.

Extending the language

These topics cover extending Genero Business Development Language with other languages and external components.

The Java interface

The *Java interface* allows you to import Java classes and instantiate Java objects in your programs.

The Java interface gives access to the huge standard Java libraries, as well as custom libraries for specific purposes.

The methods of Java objects can be called with other Java objects referenced in a program, as well as with native language data types such as `INTEGER`, `DECIMAL`, `CHAR`.

The Java interface of Genero has the following limitations:

1. It is not possible to use Java generic types such as `java.util.Vector<E>`, with a type parameter (for example, `Vector<MyClass> v = new Vector<MyClass>();`). However, it is possible to instantiate these classes without a type parameter (for example, `Vector v = new Vector();`).
2. Database connections cannot be shared between Java and Genero programs.
3. Java graphical objects cannot be used in Genero forms.

Prerequisites and installation

**Learn about Java and OOP**

Before starting with the Java interface, if you are not familiar with Java and Object Oriented Programming, we strongly recommend that you learn more about this language from the different tutorials and courses you can find on the internet.

**Related concepts**

- OOP support on page 570
Describes Object Oriented Programming basics in the language.

**Java software requirements for FGL**

In order to use the Java Interface in your application programs, you need the Java software installed and properly configured.

- Install a Java Development Kit on development sites (if you need to compile your own Java classes)
- Install a Java Runtime Environment on production sites (on the server where your programs are running)

**Note:** Over time, the Java platform has used different versioning systems, to distinguish a "Java developer version", from a "Java product version". For example, the Java developer version (JDK) 1.8 corresponds to the Java product version 8. With recent versions like Java 10, a unique version numbering convention is used.

The Java classes defined by Genero (com.fourjs.fgl.lang.*) are compiled with `javac -source 1.6 -target 1.6` options. Therefore, the minimum theoretical version is Java SE 6. However, depending on the platform, the minimum required Java version may be greater than version 6.

**Note:** As a general rule, always install the current long-term-support (LTS) Java version available on your platform, with JNI support. For example, Java SE 8 (released in March 2014) and Java SE 11 (released in September 2018) are LTS versions.

For a detailed list of supported JVMs, refer to the *Supported platforms and databases* document, available on the Products download page of the Four Js Web site.

The version of the installed Java software can be shown with the command:

```
java --version
```

In order to execute Java byte code, the Genero runtime system uses the JNI interface. The JVM is loaded as a shared library and its binary format must match the binary format of the Genero runtime system. For example, a 64-bit Genero package requires a 64-bit JVM.

When implementing Java classes for Genero Mobile for Android™ (GMA), check the JDK version required by the Android™ SDK. For more information, see the Android™ Studio web site.

**How to set up Java**

This short procedure describes how to set up a Java environment to be used with Genero.

1. Download the latest JDK from your preferred Java provider. On production sites, you only need a Java Runtime Environment (JRE).
2. Install the package by following the vendor installation notes.
3. Configure your environment for Java Native Interface (JNI) support:
   - In a development or runtime environment, on Unix-based platforms (including Mac® OS X®) and on Microsoft® Windows®, define the JAVA_HOME environment variable, to allow fglrun and fglicomp to find the Java Runtime library (libjvm.so or JVM.DLL). Defining LD_LIBRARY_PATH on Unix or PATH on Microsoft® Windows® to find the Java runtime library is not required.
   - In a development environment, set the PATH environment variable to find the Java tools (javac, java).
   - See also Platform-specific notes for the JVM on page 2194
4. Set the CLASSPATH or pass the `--java-option=-Djava.class.path=<pathlist>` option to fglrun with the directories of the Java packages you want to use. You must add `$FGLDIR/lib/fgl.jar` to the class path in order to compile Java code with language specific classes such as `com.fourjs.fgl.lang.FglDecimal` or `com.fourjs.fgl.lang.FglRecord`.
5. Try your JDK by compiling a small Java sample and executing it.
Platform-specific notes for the JVM

Unix-based platforms

The JAVA_HOME environment variable must be set to use the Java interface on Unix-based platforms.

If needed, the LD_LIBRARY_PATH environment variable can contain the path to the JVM library. This is however not required if JAVA_HOME is properly set.

Microsoft™ Windows®

The JAVA_HOME environment variable must be set to use the Java interface on Microsoft™ Windows® platforms.

If needed, the PATH environment variable can contain the path to the JVM library. This is however not required if JAVA_HOME is properly set.

Important: If PATH is defined to find the javac compiler or JVM library, make sure that it does not contain double quotes around the path to the JVM.DLL dynamic library, otherwise the DLL loader will fail to load the JVM.

See also Microsoft Windows configuration notes on page 46.

Mac® OS X®

The JAVA_HOME environment variable must be set to use the Java interface on Mac® OS X.

To find the JAVA_HOME path on Mac® OS X, use the /usr/libexec/java_home tool:

```sh
export JAVA_HOME=`/usr/libexec/java_home`
```

Note: On Mac® OS X, the usage of DYLD_LIBRARY_PATH is strongly discouraged, especially since OS X 10.11 this environment variable is no longer exported in sub processes.

The Genero runtime system uses the Java Native Interface (JNI) to interact with the JVM and execute Java code.

By default on Mac® OS X, the JNI feature may not be activated.

To ensure that Java is installed, execute the following command:

```sh
$ /usr/libexec/java_home
```

The output should look like:

```
Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home
```

To ensure that JNI is enabled, use the `-t` JNI option:

```sh
$ /usr/libexec/java_home -t JNI
```

The output should look like:

```
Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home
```

If the output is empty or reports an older version, JNI is not enabled for the default Java installation.

To enable JNI, edit the Info.plist file:

```sh
$ sudo vi `/usr/libexec/java_home`/../Info.plist
```

And add "<string>JNI</string>" in the <array> of the JVMCapabilities key:

```xml
<dict>
  <key>JVMCapabilities</key>
  <array>
    <string>JNI</string>
  </array>
</dict>
```
See also Mac OS X configuration notes on page 44.

**Android™**

The Java Interface can be used in Genero apps built for the Android™/GMA platform.

On Android™ devices, custom Java classes need to be part of the .apk package and can be used without any further configuration.

**Note:** On Android™ devices, some system functions can only be accessed in the context of a JVM. Use the Java Interface with the com.fourjs.gma.vm.FglRun class to access such system specifics.

For more details, see Executing Java code with GMA on page 2219.

**iOS**

**Important:**

The Java interface cannot be used in apps running on iOS devices: There is no standard free JVM available.

**IBM® AIX®**

Consider the following notes when using the Java Interface with the IBM® Java VM on AIX®:

If you get java.lang.UnsatisfiedLinkError exceptions, set the path to native shared libraries in the LIBPATH environment variable:

```bash
$ LIBPATH=${JAVA_HOME}/jre/bin:${JAVA_HOME}/jre/bin/j9vm:${JAVA_HOME}/jre/lib/ppc64:${LIBPATH}
$ export LIBPATH
```

This is required when using Java code that needs to access native code supplied as part of the JRE. For example, without setting LIBPATH to the appropriate path, the JVM cannot find the shared library libnet.so.

Using the -Djava.library.path=path-to-native-library java VM option does not seem to help.

See also IBM AIX configuration notes on page 44.

**HP-UX®**

Consider the following notes when using the HotSpot JVM on HP/UX:

If you get an error when fglcomp or fglrun try to load the libjvm library, define the LD_PRELOAD environment variable as follows:

```bash
$ LD_PRELOAD=libjvm.sl
$ export LD_PRELOAD
```

Using LD_PRELOAD can cause other applications fail. LD_PRELOAD should only be set for the runtime system. If you need to run other applications in the same environment as your application programs, you can set the LD_PRELOAD_ONCE or JAVA_PRELOAD_ONCE variable in the shell scripts found in FGLDIR/bin.

See also HP-UX configuration notes on page 43.
Getting started with the Java interface

**Import a Java class**

In order to use a Java class in your program code, you must first import the class with the `IMPORT JAVA` instruction:

```
IMPORT JAVA java.util.regex.Pattern
```

This will import the specified Java class into the current program module. Object references can now be defined for this class.

**Define an object reference variable**

Before creating a Java object in your program, you must declare a program variable to reference the object. The type of the variable must be the name of the Java class, and can be fully qualified if needed:

```
IMPORT JAVA java.util.regex.Pattern
MAIN
   DEFINE p1 Pattern
   DEFINE p2 java.util.regex.Pattern
END MAIN
```

The variables declared with a class are only the handles to reference an object (meaning the object is not yet created).

**Instantiate a Java class**

To create a new Java object, use `ClassName.create()`, and assign the value returned by the `create()` method to a program variable declared with the Java class name:

```
IMPORT JAVA java.lang.StringBuffer
MAIN
   DEFINE sb StringBuffer
   LET sb = StringBuffer.create()
END MAIN
```

If the Java class constructor uses parameters, pass the parameters to the `create()` method:

```
IMPORT JAVA java.lang.StringBuffer
MAIN
   DEFINE sb1, sb2 StringBuffer
   -- Next code line uses StringBuffer(String str) constructor
   LET sb1 = StringBuffer.create("abcdef")
   -- Next code line uses StringBuffer(int capacity) constructor
   LET sb2 = StringBuffer.create(2048)
END MAIN
```

**Calling a method of a class**

Class methods (static method in Java) can be called without instantiating an object of the class. Static method invocation must be prefixed with the class name. In the following example, the `compile()` class method of Pattern class returns a new instance of a Pattern object:

```
IMPORT JAVA java.util.regex.Pattern
MAIN
   DEFINE p Pattern
   LET p = Pattern.compile("[,\s]+")
END MAIN
```
If you define a variable with the same name as a Java class, you must fully qualify the class when calling static methods, as shown in this example:

```genero
IMPORT JAVA java.util.regex.Pattern
IMPORT JAVA java.util.regex.Matcher
MAIN
  DEFINE Pattern Pattern
  DEFINE Matcher Matcher
  -- static method, needs full qualifier
  LET Pattern = java.util.regex.Pattern.compile("[a-z]+")
  -- regular instance method, Pattern resolves to variable
  LET Matcher = Pattern.matcher("abcdef")
END MAIN
```

**Note:** In Genero, program variables are case-insensitive (Pattern = pattern).

### Calling a method of an object

Once the class has been instantiated as an object, and the object reference has been assigned to a variable, you can call a method of the Java object by using the variable as the prefix:

```genero
IMPORT JAVA java.util.regex.Pattern
IMPORT JAVA java.util.regex.Matcher
MAIN
  DEFINE p Pattern
  DEFINE m Matcher
  LET p = java.util.regex.Pattern.compile("[a-z]+")
  LET m = p.matcher("abcdef")
  DISPLAY m.matches()
END MAIN
```

In this example, the last line of the MAIN module calls an object method that returns a boolean value that is converted to an INTEGER and displayed.

### Advanced programming

#### Using JVM options

When using the Java interface, you can instruct `fglrun` or `fglcomp` to pass Java VM specific options during JNI initialization, by using the `--java-option` command line argument.

In the example, `fglrun` passes `-verbose:gc` to the Java Virtual Machine:

```
$ fglrun --java-option=-verbose:gc myprog.42r
```

If you want to pass several options to the JVM, repeat the `--java-option` argument as in this example:

```
$ fglrun --java-option=-verbose:gc --java-option=-esa myprog.42r
```

You may want to pass the Java class path as a command line option to `fglrun` with `-Djava.class.path` option as in this example:

```
$ fglrun --java-option=-Djava.class.path=$FGLDIR/lib/fgl.jar:$MYCLASSPATH myprog.42r
```

Regarding class path specification, the `java` runtime or `javac` compiler provides the `-cp` or `-classpath` options but when loading the JVM library from `fglrun` or `fglcomp`, only `-Djava.class.path` option is supported by the JNI interface.
**Case sensitivity with Java**

The Java language is case-sensitive. Therefore, when you write the name of a Java package, class or method in a .4gl source, it must match the exact name as if you were writing a Java program. The fglcomp compiler takes care of this, and writes case-sensitive class and method names in the .42m p-code modules.

```java
IMPORT JAVA java.util.regex.Pattern
MAIN
  DEFINE p java.util.regex.PATTERN   -- Note the case error
END MAIN
```

With this code example, fglcomp will raise error -6622 at line 3, because the "java/util/PATTERN" name cannot be found.

**Method overloading in Java**

The Java language allows method overloading; the parameter count and the parameter data types of a method are part of the method identification. Thus, the same method name can be used to implement different versions of the Java method, taking different parameters:

```java
DEFINE int2 SMALLINT, int4 INTEGER, flt FLOAT

-- Next call invokes method display(short) of the Java class
CALL myobj.display(int2)

-- Next call invokes method display(int) of the Java class
CALL myobj.display(int4)

-- Next call invokes method display(double) of the Java class
CALL myobj.display(flt)

-- Next call invokes method display(short,int) of the Java class
CALL myobj.display(int2,int4)
```

**Passing Java objects to functions**

Java objects must be instantiated and referenced by a program variable. The object reference is stored in the variable and can be passed as a parameter or returned from a program function. The Java objects are passed by reference to functions. This means that the called function does not get a clone of the object, but rather a handle to the original object. The function can then manipulate and modify the original object provided by the caller:

```java
IMPORT JAVA java.lang.StringBuffer
MAIN
  DEFINE x java.lang.StringBuffer
  LET x = StringBuffer.create()
  CALL change(x)
  DISPLAY x.toString()
END MAIN

FUNCTION change(sb)
  DEFINE sb java.lang.StringBuffer
  CALL sb.append("abc")
END FUNCTION
```

Similarly, Java object references can be returned from functions:

```java
IMPORT JAVA java.lang.StringBuffer
MAIN
  DEFINE x java.lang.StringBuffer
```
LET x = build()
DISPLAY x.toString()
END MAIN

FUNCTION build()
    DEFINE sb java.lang.StringBuffer
    LET sb = StringBuffer.create()   -- Creates a new object.
    CALL sb.append("abc")
    RETURN sb  -- Returns the reference to the object, not a copy/clone.
END FUNCTION

Garbage collection of unused objects
Java objects do not need to be explicitly destroyed; as long as an object is referenced by a variable, on the stack or in an expression, it will remain. When the last reference to an object is removed, the object is destroyed automatically.

This example shows how an unique object can be referenced twice, using two variables:

FUNCTION test()
    -- Declare 2 variables to reference a StringBuffer object
    DEFINE sb1, sb2 java.lang.StringBuffer
    -- Create object and assign reference to variable
    LET sb1 = StringBuffer.create()
    -- Same object is now referenced by 2 variables
    LET sb2 = sb1
    -- Object is modified through first variable
    CALL sb1.append("abc")
    -- Object is modified through second variable
    CALL sb2.append("def")
    -- Shows content of StringBuffer object
    DISPLAY sb1.toString()
    -- Same output as previous line
    DISPLAY sb2.toString()
    -- Object is only referenced by second variable
    LET sb1 = NULL
    -- sb2 removed from stack, object is no longer referenced and is destroyed.
END FUNCTION

Using the method return as an object
If a Java method returns an object, you can use the method call directly as an object reference to call another method:

IMPORT JAVA java.util.regex.Pattern
MAIN
    DEFINE p Pattern
    LET p = Pattern.compile("a*b")
    IF p.matcher("aaaab").matches() THEN
        DISPLAY "It matches..."
    END IF
END MAIN

In this code example, the matcher() method of object p is invoked and returns an object of type java.util.regex.Matcher. The object reference returned by the matcher() method can be directly used to invoke the matches() method of the Matcher class.

Ignorable return of Java methods
Java allows you to ignore the return value of a method (as in C/C++):

StringBuffer sb = new StringBuffer;
sb.append("abc");  -- returns a new StringBuffer object but is ignored
In programs, you can call a Java method and ignore the return value:

```java
IMPORT JAVA java.util.lang.StringBuffer
MAIN
  DEFINE sb StringBuffer
  LET sb = StringBuffer.create()
  LET sb = sb.append("abc")
  CALL sb.append("def")  -- typical usage
END MAIN
```

**Static fields of Java classes**

Java classes can have object and class ("static") fields. Java static class fields can be declared as "final" (read-only). It is not possible to change the object or class fields in programs, even if the field is not declared as "static final"; you can however read from it:

```java
IMPORT JAVA java.lang.Integer
MAIN
  DISPLAY Integer.MAX_VALUE
END MAIN
```

**Mapping native and Java data types**

Java and Genero have different primitive data types. Unlike Genero, Java is a strongly typed language: You cannot call a method with a `String` if it was defined to get an `int` parameter. To call a Java method, Genero native typed values need to be converted to/from Java types such as `byte`, `int`, `short`, `char` or data objects such as `java.lang.String`. If possible, the fglrun runtime system will do this conversion implicitly.

The `fglcomp` compiler will raise the error `-6606`, if the native data type does not match the Java (primitive) type, using Widening Primitive Conversions. For example, passing a Genero `DECIMAL` when a Java `double` is expected will fail, but passing a `SMALLFLOAT` (equivalent to `float`) when a Java `double` is expected will compile and run.

Genero has advanced native data types such as `DECIMAL`, which do not have an equivalent primitive type or class in Java. For such Genero types, you need to use a specific Java class provided in the `$FGLDIR/lib/fgl.jar` package, like `com.fourjs.fgl.lang.FglDecimal`. You can then manipulate the Genero specific value in the Java code.

Genero also implements structured types with `RECORD` definitions, converted to `com.fourjs.fgl.lang.FglRecord` objects for Java.

The Genero arrays cannot be used to call Java methods. You must use native Java arrays instead.

In some cases you need to explicitly cast with the new `CAST()` operator. See the topic about `CAST()` operator for more details.

The following tables show the implicit conversions done by the runtime system when a Java method is called, or when a Java method returns a value or object reference:

**Table 458: Implicit BDL/Java conversions using simple types**

<table>
<thead>
<tr>
<th>Genero data type</th>
<th>Java equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>long (64-bit signed integer)</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>Boolean</td>
</tr>
<tr>
<td>BYTE</td>
<td><code>com.fourjs.fgl.lang.FglByteBlob</code></td>
</tr>
<tr>
<td>CHAR</td>
<td><code>java.lang.String</code></td>
</tr>
<tr>
<td>DATE</td>
<td><code>com.fourjs.fgl.lang.FglDateTime</code></td>
</tr>
<tr>
<td>DATETIME</td>
<td><code>com.fourjs.fgl.lang.FglDateTime</code></td>
</tr>
<tr>
<td>Genero data type</td>
<td>Java equivalent</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>com.fourjs.fgl.lang.FglDecimal</td>
</tr>
<tr>
<td>FLOAT</td>
<td>double (64-bit signed floating point number)</td>
</tr>
<tr>
<td>INTEGER</td>
<td>int (32-bit signed integer)</td>
</tr>
<tr>
<td>INTERVAL</td>
<td>com.fourjs.fgl.lang.FglInterval</td>
</tr>
<tr>
<td>MONEY</td>
<td>com.fourjs.fgl.lang.FglDecimal</td>
</tr>
<tr>
<td>SMALLFLOAT</td>
<td>float (32-bit signed floating point number)</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>short (16-bit signed integer)</td>
</tr>
<tr>
<td>STRING</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>TEXT</td>
<td>com.fourjs.fgl.lang.FglTextBlob</td>
</tr>
<tr>
<td>TINYINT</td>
<td>byte (8-bit signed integer)</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>java.lang.String</td>
</tr>
</tbody>
</table>

Table 459: Implicit BDL/Java conversions using structured types

<table>
<thead>
<tr>
<th>Genero data type</th>
<th>Java equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>RECORD structure</td>
<td>com.fourjs.fgl.lang.FglRecord</td>
</tr>
<tr>
<td>Java Array</td>
<td>This is a native Java Array</td>
</tr>
</tbody>
</table>

Table 460: Native BDL types that cannot be converted to Java types

<table>
<thead>
<tr>
<th>Genero data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARRAY structures</td>
</tr>
<tr>
<td>Built-in classes</td>
</tr>
</tbody>
</table>

Using the DATE type

When calling a Java method with an expression evaluating to a DATE, the runtime system converts the DATE value to an instance of the com.fourjs.fgl.lang.FglDate class implemented in $FGLDIR/lib/fgl.jar. You can then manipulate the date from within the Java code.

You must add $FGLDIR/lib/fgl.jar to the class path in order to compile Java code with com.fourjs.fgl.lang.FglDate class.

The com.fourjs.fgl.lang.FglDate class implements following:

Table 461: Methods of the com.fourjs.fgl.lang.FglDate class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.String toString()</td>
<td>Converts the DATE value to a String object representing the date in the format: YYYY-MM-DD</td>
</tr>
<tr>
<td>static FglDate valueOf( java.lang.String val)</td>
<td>Creates a new FglDate object from a String object representing a date in the format YYYY-MM-DD.</td>
</tr>
</tbody>
</table>
In the Java code, you can convert the `com.fourjs.fgl.lang.FglDate` to a `java.util.Calendar` object as in this example:

```java
public static void useDate(FglDate d) throws ParseException {
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
    Calendar cal = Calendar.getInstance();
    cal.setTime( sdf.parse(d.toString()) );
    ...
}
```

If you need to create a `com.fourjs.fgl.lang.FglDate` object in your program, you can use the `valueOf()` class method as in this example:

```java
IMPORT JAVA com.fourjs.fgl.lang.FglDate
MAIN
    DEFINE d com.fourjs.fgl.lang.FglDate
    LET d = FglDate.valueOf("2008-12-23")
    DISPLAY d.toString()
END MAIN
```

**Using the DATETIME type**

When calling a Java method with an expression evaluating to a DATETIME, the runtime system converts the DATETIME value to an instance of the `com.fourjs.fgl.lang.FglDateTime` class implemented in `$FGLDIR/lib/fgl.jar`. You can then manipulate the DATETIME from within the Java code.

You must add `$FGLDIR/lib/fgl.jar` to the class path in order to compile Java code with `com.fourjs.fgl.lang.FglDateTime` class.

The `com.fourjs.fgl.lang.FglDateTime` class implements the following:
### Table 462: Fields of the com.fourjs.fgl.lang.FglDateTime class

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>final static int YEAR</td>
<td>Time qualifier for year</td>
</tr>
<tr>
<td>final static int MONTH</td>
<td>Time qualifier for month</td>
</tr>
<tr>
<td>final static int DAY</td>
<td>Time qualifier for day</td>
</tr>
<tr>
<td>final static int HOUR</td>
<td>Time qualifier for hour</td>
</tr>
<tr>
<td>final static int MINUTE</td>
<td>Time qualifier for minute</td>
</tr>
<tr>
<td>final static int SECOND</td>
<td>Time qualifier for second</td>
</tr>
<tr>
<td>final static int FRACTION</td>
<td>Time qualifier for fraction (start qualifier)</td>
</tr>
<tr>
<td>final static int FRACTION1</td>
<td>Time qualifier for fraction(1) (end qualifier)</td>
</tr>
<tr>
<td>final static int FRACTION2</td>
<td>Time qualifier for fraction(2) (end qualifier)</td>
</tr>
<tr>
<td>final static int FRACTION3</td>
<td>Time qualifier for fraction(3) (end qualifier)</td>
</tr>
<tr>
<td>final static int FRACTION4</td>
<td>Time qualifier for fraction(4) (end qualifier)</td>
</tr>
<tr>
<td>final static int FRACTION5</td>
<td>Time qualifier for fraction(5) (end qualifier)</td>
</tr>
</tbody>
</table>
### Table 463: Methods of the com.fourjs.fgl.lang.FglDateTime class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
</table>
| static int encodeTypeQualifier(int startUnit, int endUnit) | Returns the encoded type qualifier for a datetime with to datetime qualifiers passed: 
encoded qualifier = (length * 256) + (startUnit * 16) + endUnit 
Where length defines the total number of significant digits in this time data. 
For example, with DATETIME YEAR TO MINUTE: 
startUnit = YEAR 
length = 12 (YYYYMMDDhhmm) 
endUnit = MINUTE |
| java.lang.String toString() | Converts the DATETIME value to a String object representing a datetime in the format YYYY-MM-DD hh:mm:ss.fff. |
| static FglDateTime valueOf(long milliseconds) | Creates a new FglDateTime object representing the specified number of milliseconds since the standard base time known as "the epoch", namely January 1, 1970, 00:00:00 GMT. |
| static FglDateTime valueOf(java.lang.String val) | Creates a new FglDateTime object from a String object representing a datetime value in the format: YYYY-MM-DD hh:mm:ss.fff |
| static FglDateTime valueOf(java.lang.String val, int startUnit, int endUnit) | Creates a new FglDateTime object from a String object representing a datetime value in the format YYYY-MM-DD hh:mm:ss.fff, using the qualifiers passed as parameter. |

In the Java code, you can convert the com.fourjs.fgl.lang.FglDateTime to a java.util.Calendar object as in this example:

```java
public static void useDatetime(FglDateTime dt) throws ParseException {
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS");
    Calendar cal = Calendar.getInstance();
    cal.setTime( sdf.parse(dt.toString()) );
    ...
}
```

If you need to create a com.fourjs.fgl.lang.FglDateTime object in your program, you can use the valueOf() class method as in this example:

```java
import JAVA com.fourjs.fgl.lang.FglDateTime
main
    define dt com.fourjs.fgl.lang.FglDateTime
    let dt = FglDateTime.valueOf("2008-12-23 11:22:33.123")
    let dt = FglDateTime.valueOf("11:22:33.123", FglDateTime.HOUR, FglDateTime.FRACTION3)
    display dt.toString()
end main
```
The `valueOf()` method expects a string representing a complete date-time specification, from year to milliseconds, equivalent to a `DATETIME YEAR TO FRACTION(3)` data type.

**Using the DECIMAL type**

When calling a Java method with an expression evaluating to a `DECIMAL`, the runtime system converts the DECIMAL value to an instance of the `com.fourjs.fgl.lang.FglDecimal` class implemented in `$FGLDIR/lib/fgl.jar`. You can then manipulate the DECIMAL from within the Java code.

You must add `$FGLDIR/lib/fgl.jar` to the class path in order to compile Java code with `com.fourjs.fgl.lang.FglDecimal` class.

The `com.fourjs.fgl.lang.FglDecimal` class implements the following methods shown in the table:

**Table 464: Methods of the `com.fourjs.fgl.lang.FglDecimal` class**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>static int encodeTypeQualifier(int precision, int scale)</code></td>
<td>Returns the encoded type qualifier for this decimal based on precision and scale. The encoded qualifier is calculated as <code>(precision * 256) + scale</code>. Use 255 as scale for floating point decimal.</td>
</tr>
<tr>
<td><code>java.lang.String toString()</code></td>
<td>Converts the DECIMAL value to a String object.</td>
</tr>
<tr>
<td><code>static FglDecimal valueOf(int val)</code></td>
<td>Creates a new <code>FglDecimal</code> object from an int value.</td>
</tr>
<tr>
<td><code>static FglDecimal valueOf(java.lang.String val)</code></td>
<td>Creates a new <code>FglDecimal</code> object from a String object representing a decimal value.</td>
</tr>
</tbody>
</table>

In the Java code, you can convert the `com.fourjs.fgl.lang.FglDecimal` to a `java.lang.BigDecimal` as in the following example:

```java
public static FglDecimal divide(FglDecimal d1, FglDecimal d2){
    BigDecimal bd1 = new BigDecimal(d1.toString());
    BigDecimal bd2 = new BigDecimal(d2.toString());
    BigDecimal res = bd1.divide(bd2, BigDecimal.ROUND_FLOOR);
    return FglDecimal.valueOf(res.toString());
}
```

If you need to create a `com.fourjs.fgl.lang.FglDecimal` object in your program, you can use the `valueOf()` class method as in this example:

```java
IMPORT JAVA com.fourjs.fgl.lang.FglDecimal
MAIN
    DEFINE jdec com.fourjs.fgl.lang.FglDecimal
    LET jdec = FglDecimal.valueOf("123.45")
    DISPLAY jdec.toString()
END MAIN
```

**Using the TEXT type**

When calling a Java method with an expression evaluating to a `TEXT`, the runtime system converts the TEXT handle to an instance of the `com.fourjs.fgl.lang.FglTextBlob` class implemented in `$FGLDIR/lib/fgl.jar`. You can then manipulate the LOB from within the Java code.
You must add `$FGLDIR/lib/fgl.jar` to the class path in order to compile Java code with com.fourjs.fgl.lang.FglTextBlob class.

The com.fourjs.fgl.lang.FglTextBlob class implements the following:

**Table 465: Methods of the com.fourjs.fgl.lang.FglTextBlob class**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void dispose()</td>
<td>Dereferences the underlying TEXT variable.</td>
</tr>
<tr>
<td>java.lang.String toString()</td>
<td>Converts the large text data to a simple String.</td>
</tr>
<tr>
<td>static FglTextBlob valueOf(java.lang.String val)</td>
<td>Creates a new FglTextBlob object from a String.</td>
</tr>
</tbody>
</table>

In the Java code, you can pass a com.fourjs.fgl.lang.FglTextBlob object as in this example:

```java
public static void useByte(FglTextBlob t) throws ParseException {
    String s = t.toString();
    ...
}
```

If you need to create a com.fourjs.fgl.lang.FglTextBlob object in your program, you can use the valueOf() class method as in this example:

```java
IMPORT JAVA com.fourjs.fgl.lang.FglTextBlob
MAIN
    DEFINE jtext com.fourjs.fgl.lang.FglTextBlob
    LET jtext = FglTextBlob.valueOf("abcdef.........")
    DISPLAY jtext.toString()
END MAIN
```

**Using the BYTE type**

When calling a Java method with an expression evaluating to a BYTE, the runtime system converts the BYTE handle to an instance of the com.fourjs.fgl.lang.FglByteBlob class implemented in `$FGLDIR/lib/fgl.jar`. You can then manipulate the LOB from within the Java code.

You must add `$FGLDIR/lib/fgl.jar` to the class path in order to compile Java code with com.fourjs.fgl.lang.FglByteBlob class.

The com.fourjs.fgl.lang.FglByteBlob class implements the following:

**Table 466: Methods of the com.fourjs.fgl.lang.FglByteBlob class**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void dispose()</td>
<td>Dereferences the underlying BYTE variable.</td>
</tr>
<tr>
<td>java.lang.String toString()</td>
<td>Returns the HEX string representing the binary data.</td>
</tr>
<tr>
<td>static FglByteBlob valueOf(java.lang.String val)</td>
<td>Creates a new FglByteBlob object from a String object representing the binary data in HEX format.</td>
</tr>
</tbody>
</table>
In the Java code, you can pass a `com.fourjs.fgl.lang.FglByteBlob` object as in this example:

```java
public static void useByte(FglByteBlob b) throws ParseException {
    String s = b.toString();
    ...
}
```

If you need to create a `com.fourjs.fgl.lang.FglByteBlob` object in your program, you can use the `valueOf()` class method as in this example:

```java
IMPORT JAVA com.fourjs.fgl.lang.FglByteBlob
MAIN
    DEFINE jbyte com.fourjs.fgl.lang.FglByteBlob
    LET jbyte = FglByteBlob.valueOf("0FA5617BDE")
    DISPLAY jbyte.toString()
END MAIN
```

### Using the INTERVAL type

When calling a Java method with an expression evaluating to an `INTERVAL`, the runtime system converts the `INTERVAL` value to an instance of the `com.fourjs.fgl.lang.FglInterval` class implemented in `$FGLDIR/lib/fgl.jar`. You can then manipulate the `INTERVAL` from within the Java code.

You must add `$FGLDIR/lib/fgl.jar` to the class path in order to compile Java code with `com.fourjs.fgl.lang.FglInterval` class.

The `com.fourjs.fgl.lang.FglInterval` class implements the following:
Table 467: Fields of the com.fourjs.fgl.lang.FglInterval class

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>final static int YEAR</td>
<td>Time qualifier for year</td>
</tr>
<tr>
<td>final static int MONTH</td>
<td>Time qualifier for month</td>
</tr>
<tr>
<td>final static int DAY</td>
<td>Time qualifier for day</td>
</tr>
<tr>
<td>final static int HOUR</td>
<td>Time qualifier for hour</td>
</tr>
<tr>
<td>final static int MINUTE</td>
<td>Time qualifier for minute</td>
</tr>
<tr>
<td>final static int SECOND</td>
<td>Time qualifier for second</td>
</tr>
<tr>
<td>final static int FRACTION</td>
<td>Time qualifier for fraction (start qualifier)</td>
</tr>
<tr>
<td>final static int FRACTION1</td>
<td>Time qualifier for fraction(1) (end qualifier)</td>
</tr>
<tr>
<td>final static int FRACTION2</td>
<td>Time qualifier for fraction(2) (end qualifier)</td>
</tr>
<tr>
<td>final static int FRACTION3</td>
<td>Time qualifier for fraction(3) (end qualifier)</td>
</tr>
<tr>
<td>final static int FRACTION4</td>
<td>Time qualifier for fraction(4) (end qualifier)</td>
</tr>
<tr>
<td>final static int FRACTION5</td>
<td>Time qualifier for fraction(5) (end qualifier)</td>
</tr>
</tbody>
</table>
Table 468: Methods of the `com.fourjs.fgl.lang.FglInterval` class

<table>
<thead>
<tr>
<th>Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static int encodeTypeQualifier(</td>
<td>Returns the encoded type qualifier for an interval with to interval qualifiers and length passed:</td>
</tr>
<tr>
<td>int startUnit,</td>
<td>encoded qualifier = (<code>length</code> * 256) + (<code>startUnit</code> * 16) + <code>endUnit</code></td>
</tr>
<tr>
<td>int length,</td>
<td>Where <code>length</code> defines the total number of significant digits in this time data.</td>
</tr>
<tr>
<td>int endUnit)</td>
<td>For example, with INTERVAL DAY(5) TO FRACTION3:</td>
</tr>
<tr>
<td>static java.lang.String toString()</td>
<td>Converts the INTERVAL value to a String object representing an interval in default format.</td>
</tr>
<tr>
<td>static FglInterval valueOf(</td>
<td>Creates a new FglInterval object from a String object representing an interval value in the format:</td>
</tr>
<tr>
<td>java.lang.String val)</td>
<td>DD hh:mm:ss.fff</td>
</tr>
<tr>
<td>static FglInterval valueOf(</td>
<td>Creates a new FglDateTime object from a String object representing an interval value in standard format, using the qualifiers and precision passed as parameter.</td>
</tr>
<tr>
<td>java.lang.String val,</td>
<td></td>
</tr>
<tr>
<td>int startUnit,</td>
<td></td>
</tr>
<tr>
<td>int endUnit)</td>
<td></td>
</tr>
</tbody>
</table>

In the Java code, you can pass a `com.fourjs.fgl.lang.FglInterval` object as in this example:

```java
public static void useInterval(FglInterval inv) throws ParseException {
    String s = inv.toString();
    ...
}
```

If you need to create a `com.fourjs.fgl.lang.FglInterval` object in your program, you can use the `valueOf()` class method as in this example:

```java
IMPORT JAVA com.fourjs.fgl.lang.FglInterval
MAIN
    DEFINE inv com.fourjs.fgl.lang.FglInterval
    LET inv = FglInterval.valueOf("-510 12:33:45.123")
    DISPLAY inv.toString()
END MAIN
```

**Identifying Genero data types in Java code**

Java data types and Genero data types are different. To identify Genero types in Java code, you can use the `com.fourjs.fgl.lang.FglTypes` class implemented in `$FGLDIR/lib/fgl.jar`.

You can, for example, identify the data type of a member of an `FglRecord` object.

You must add `$FGLDIR/lib/fgl.jar` to the class path in order to compile Java code with `com.fourjs.fgl.lang.FglType` class.
The `com.fourjs.fgl.lang.FglTypes` class implements the following:
### Table 469: Fields of the com.fourjs.fgl.lang.FglTypes class

<table>
<thead>
<tr>
<th>Field</th>
<th>Corresponding data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>final static int ARRAY</td>
<td>ARRAY object</td>
</tr>
<tr>
<td>final static int BIGINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>final static int BOOLEAN</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>final static int BYTE</td>
<td>BYTE</td>
</tr>
<tr>
<td>final static int CHAR</td>
<td>CHAR</td>
</tr>
<tr>
<td>final static int DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>final static int DATETIME</td>
<td>DATETIME</td>
</tr>
<tr>
<td>final static int DECIMAL</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>final static int FGL_OBJECT</td>
<td>An FGL object like base.Channel.</td>
</tr>
<tr>
<td>final static int FLOAT</td>
<td>FLOAT</td>
</tr>
<tr>
<td>final static int INT</td>
<td>INTEGER</td>
</tr>
<tr>
<td>final static int INTERVAL</td>
<td>INTERVAL</td>
</tr>
<tr>
<td>final static int JAVA_OBJECT</td>
<td>A Java object like java.lang.String.</td>
</tr>
<tr>
<td>final static int MONEY</td>
<td>MONEY</td>
</tr>
<tr>
<td>final static int RECORD</td>
<td>RECORD structure</td>
</tr>
<tr>
<td>final static int SMALLFLOAT</td>
<td>SMALLFLOAT</td>
</tr>
<tr>
<td>final static int SMALLINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>final static int STRING</td>
<td>STRING</td>
</tr>
<tr>
<td>final static int TEXT</td>
<td>TEXT</td>
</tr>
<tr>
<td>final static int TINYINT</td>
<td>TINYINT</td>
</tr>
</tbody>
</table>
Using Genero records

When passing a RECORD to a Java method, the runtime system converts the RECORD to an instance of the com.fourjs.fgl.lang.FglRecord class implemented in $FGLDIR/lib/fgl.jar.

The FglRecord object is a copy of the RECORD variable; structure and members of the FglRecord object can be read within the Java code, but cannot be modified.

You must add $FGLDIR/lib/fgl.jar to the class path in order to compile Java code with com.fourjs.fgl.lang.FglRecord class.

The com.fourjs.fgl.lang.FglRecord class implements the following methods:
### Table 470: Methods of the com.fourjs.fgl.lang.FglRecord class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void dispose()</td>
<td>Dereferences the underlaying member variables.</td>
</tr>
<tr>
<td>double getDouble(int p)</td>
<td>Returns the double value of the record member at position p.</td>
</tr>
<tr>
<td>FglByteBlob getFglByteBlob(int p)</td>
<td>Returns the FglByteBlob value of the record member at position p.</td>
</tr>
<tr>
<td>FglDate getFglDate(int p)</td>
<td>Returns the FglDate value of the record member at position p.</td>
</tr>
<tr>
<td>FglDateTime getFglDateTime(int p)</td>
<td>Returns the FglDateTime value of the record member at position p.</td>
</tr>
<tr>
<td>FglDecimal getFglDecimal(int p)</td>
<td>Returns the FglDecimal value of the record member at position p.</td>
</tr>
<tr>
<td>FglInterval getFglInterval(int p)</td>
<td>Returns the FglInterval value of the record member at position p.</td>
</tr>
<tr>
<td>FglTextBlob getFglTextBlob(int p)</td>
<td>Returns the FglTextBlob value of the record member at position p.</td>
</tr>
<tr>
<td>int getFieldCount()</td>
<td>Returns the number of record members.</td>
</tr>
<tr>
<td>java.lang.String getFieldName(int p)</td>
<td>Returns the name of the record member at position p.</td>
</tr>
<tr>
<td>int getInt(int p)</td>
<td>Returns the int value of the record member at position p.</td>
</tr>
<tr>
<td>java.lang.String getString(int p)</td>
<td>Returns the String representation of the value of the record member at position p.</td>
</tr>
<tr>
<td>FglTypes getType(int p)</td>
<td>Returns the FglTypes constant of the record member at position p.</td>
</tr>
<tr>
<td>java.lang.String getTypeName(int p)</td>
<td>Returns the string representation of the data type of the record member at position p.</td>
</tr>
<tr>
<td>int getTypeQualifier(int p)</td>
<td>Returns the encoded type qualifier of the record member at position p.</td>
</tr>
</tbody>
</table>

In the Java code, use the query methods of the com.fourjs.fgl.lang.FglRecord to identify the members of the RECORD:

```java
public static void showMemberTypes(FglRecord rec){
    int i;
    int n = rec.getFieldCount();
    for (i = 1; i <= n; i++)
        System.out.println( String.valueOf(i) + "":" +
            rec.getFieldName(i) + " / " + rec.getTypeName(i) );
```
When assigning a RECORD to a `com.fourjs.fgl.lang.FglRecord`, *widening conversion* applies implicitly. But when assigning a `com.fourjs.fgl.lang.FglRecord` to a RECORD, *narrowing conversion* applies and you must explicitly *CAST* the original object reference to the type of the RECORD. The following example shows how to return an FglRecord object from a Java method:

```java
-- PassRecord.4gl
IMPORT JAVA com.fourjs.fgl.lang.FglRecord
IMPORT JAVA UseRecord
MAIN
    TYPE type1 RECORD
        id INTEGER,
        name VARCHAR(50)
    END RECORD
    DEFINE rec1, rec2 type1
    LET rec1.id = 123
    LET rec1.name = "McFly"
    LET rec2 = CAST(UseRecord.getRecord(rec1) AS type1)
END MAIN

-- UseRecord.java
import com.fourjs.fgl.lang.FglRecord;
public class UseRecord{
    public static FglRecord getRecord(FglRecord rec){
        ...
        return rec;
    }
}
```

**Formatting data in Java code**

To format numeric and date-time data in Java code, use the `com.fourjs.fgl.lang.FglFormat` class implemented in `$FGLDIR/lib/fgl.jar`.

You must add `$FGLDIR/lib/fgl.jar` to the class path in order to compile Java code with `com.fourjs.fgl.lang.FglFormat` class.

The `com.fourjs.fgl.lang.FglFormat` class provides an interface to the data formatting functions of the runtime system. This class is actually an equivalent of the *USING* operator in the language.

The `com.fourjs.fgl.lang.FglFormat` class implements the following:
### Methods of the com.fourjs.fgl.lang.FglFormat class

**Table 471: Methods of the com.fourjs.fgl.lang.FglFormat class**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>static String format(int v, String fmt)</code></td>
<td>Formats the integer value provided as Java int based on fmt. Here fmt must specify a numeric format with [ $@*#(&amp;&lt;&gt;()+-] characters, same as in the USING operator.</td>
</tr>
<tr>
<td><code>static String format(double v, String fmt)</code></td>
<td>Formats the FLOAT value provided as Java double based on fmt. Here fmt must specify a numeric format with [ $@*#(&amp;&lt;&gt;()+-,] characters, same as in the USING operator.</td>
</tr>
<tr>
<td><code>static String format(FglDate v, String fmt)</code></td>
<td>Formats the DATE value provided as FglDate based on fmt. Here fmt must specify a date format with [mdy] characters, same as in the USING operator.</td>
</tr>
<tr>
<td><code>static String format(FglDecimal v, String fmt)</code></td>
<td>Formats the DECIMAL value provided as FglDecimal, by using fmt. Here fmt must specify a numeric format with [ $@*# $&lt;()+-,] characters, same as in the USING operator.</td>
</tr>
</tbody>
</table>

Example of Java code using the com.fourjs.fgl.lang.FglFormat class:

```java
public static void formatDecimal(FglDecimal dec){
    System.out.println( FglFormat.format(dec,"$#####&.&&" );
}
```

### Character set mapping

Application programs use a given locale and character set, while Java uses its own charset internally for the char Java type (16-bit UNICODE).

When passing character strings to/from Java methods or when assigning program strings to `java.lang.String`, the runtime system handles character set conversion.

### Using Java arrays

Java arrays and Genero arrays are different. In order to interface with Java arrays, the Genero language has been extended with a new kind of array, called "Java arrays".

Java arrays have to be created with a given length. Like native Java arrays, the length cannot be changed after the array is created.

**Important:** In BDL Java arrays, the first element of is at index position 1, while in Java code, the first element of an array starts at index 0 (zero).

To create a Java array in Genero, you must define a TYPE in order to call the `create()` type method of Java arrays. The type of the elements in a Java array must be one of the language types that have a corresponding primitive type in Java (such as INTEGER (int), FLOAT (double)), or it must be a Java class such as `java.lang.String`.

The Java arrays are passed to Java methods by reference, so the elements of the array can be manipulated in Java. Furthermore, Java arrays can be created in Java code and returned to the Genero program.

This example shows how to create a Java array in Genero, to instantiate a Java Array of INTEGER elements:

```genero
MAIN
    TYPE int_array_type ARRAY[] OF INTEGER
    DEFINE ja int_array_type
    LET ja = int_array_type.create(100)
```
This example shows a program creating a Java array of Java strings:

```java
IMPORT JAVA java.lang.String
MAIN
    TYPE string_array_type ARRAY[] OF java.lang.String
    DEFINE names string_array_type
    LET names = string_array_type.create(100)
    LET names[1] = "aaaaaaaa"
    DISPLAY names[1].length()
END MAIN
```

To create a Java array of structured RECORD elements, use the `com.fourjs.fgl.lang.FglRecord` class:

```java
IMPORT JAVA com.fourjs.fgl.lang.FglRecord
MAIN
    TYPE record_array_type ARRAY[] OF com.fourjs.fgl.lang.FglRecord
    DEFINE ra record_array_type
    TYPE r_t RECORD
        id INTEGER,
        name VARCHAR(100)
    END RECORD
    DEFINE r r_t
    LET ra = record_array_type.create(100)
    LET r.id = 123
    LET r.name = "McFly"
    LET ra[10] = r
    INITIALIZE r TO NULL
    LET r = CAST (ra[10] AS r_t)
    DISPLAY r.*
END MAIN
```

Java arrays of Java classes can be defined. This example introspects the `java.lang.String` class by using Java array of `java.lang.reflect.Method` to query the list of methods from the `java.lang.String` class:

```java
IMPORT JAVA java.lang.Class
IMPORT JAVA java.lang.reflect.Method
MAIN
    DEFINE c java.lang.Class
    DEFINE ma ARRAY[] OF java.lang.reflect.Method
    DEFINE i INTEGER
    LET c = Class.forName("java.lang.String")
    LET ma = c.getMethods()
    FOR i = 1 TO ma.getLength()
        DISPLAY ma[i].toString()
    END FOR
END MAIN
```

Java arrays can be created in the Java code, to be returned from a method and assigned to a program variable:

```java
public static int[] createIntegerArray(int size) {
    return new int[size];
}
```
Passing variable arguments (varargs)

Java supports variable arguments in method definitions with the ellipsis notation, allowing callers to pass a different number of arguments depending on the need. A typical example is a message print method:

```java
import java.lang.String;

public class MyClass {
    public static void ShowStrings( String... sl ) {
        for ( String s : sl )
            System.out.println(s);
    }
}
```

In order to call such a method from the Genero program, create a Java array of the type of the variable argument, fill the array with objects and call the method with that array:

```java
IMPORT JAVA java.lang.String
IMPORT JAVA MyClass

MAIN
    TYPE sl_t ARRAY[] OF java.lang.String
    DEFINE sl ARRAY[] OF java.lang.String
    LET sl = sl_t.create(2)
    LET sl[1] = "Value 1"
    LET sl[2] = "Value 2"
    CALL MyClass.ShowStrings(sl)
END MAIN
```

Since Java arrays have a static size, you must create the Java array with the exact number of variable arguments to be passed to the method.

If the Java class cannot be modified, consider implementing a function to wrap calls to the Java method, with a varying number of arguments. It can for example take a BDL dynamic array as parameter, to simplify the callers code:

```java
IMPORT JAVA java.lang.String
IMPORT JAVA MyClass

MAIN
    DEFINE a DYNAMIC ARRAY OF STRING
    LET a[1] = "Value 1"
    LET a[2] = "Value 2"
    LET a[3] = "Value 3"
    CALL my_show_strings(a)
    LET a[4] = "Value 1"
    LET a[5] = "Value 2"
    CALL my_show_strings(a)
END MAIN

FUNCTION my_show_strings(sa)
    TYPE sl_t ARRAY[] OF java.lang.String
    DEFINE sa DYNAMIC ARRAY OF STRING
    DEFINE sl ARRAY[] OF java.lang.String
    DEFINE i INTEGER
    LET sl = sl_t.create(sa.getLength())
    FOR i=1 TO sa.getLength()
        LET sl[i] = sa[i]
    END FOR
    CALL MyClass.ShowStrings(sl)
END FUNCTION
```
If the Java class can be modified, a good practice is to write overloaded methods, using a static number of arguments:

```java
public class MyClass {
    private static void _ShowStrings( String... sl ) {
        for ( String s : sl )
            System.out.println(s);
    }
    public static void ShowStrings(String s1) {
        _ShowStrings(s1);
    }
    public static void ShowStrings(String s1, String s2) {
        _ShowStrings(s1, s2);
    }
    public static void ShowStrings(String s1, String s2, String s3) {
        _ShowStrings(s1, s2, s3);
    }
}
```

**Related concepts**

- Using Java arrays on page 2215

**The CAST operator**

Important consideration has to be taken when assigning object references to different target types or classes.

A *Widening Reference Conversion* occurs when an object reference is converted to a superclass that can accommodate any possible reference of the original type or class.

A *Narrowing Reference Conversion* occurs when an object reference of a superclass is converted to a subtype or subclass of the original object reference.

For example, in a vehicle class hierarchy with `Vehicle` and `Car` classes, `Car` is a subclass that inherits from the `Vehicle` superclass. When assigning a `Car` object reference to a `Vehicle` variable, *Widening Reference Conversion* takes place. When assigning a `Vehicle` object reference to a `Car` variable, *Narrowing Reference Conversion* occurs.

While widening conversion does not require casts and will not produce compilation or runtime errors, narrowing conversion needs the **CAST operator** to convert to the target type or class:

```java
CAST( object_reference AS type_or_class )
```

The following example creates a `java.lang.StringBuffer` object, and assigns the reference to a `java.lang.Object` variable (implying *Widening Reference Conversion*); then the object reference is assigned back to the `java.lang.StringBuffer` variable (implying *Narrowing Reference Conversion* and CAST operator usage):

```java
IMPORT JAVA java.lang.Object
IMPORT JAVA java.lang.StringBuffer
MAIN
    DEFINE o java.lang.Object
    DEFINE sb java.lang.StringBuffer
    LET sb = StringBuffer.create()
    -- Widening Reference Conversion
    LET o = sb
    -- Narrowing Reference Conversion needs CAST()
    LET sb = CAST( o AS StringBuffer )
END MAIN
```

**The INSTANCEOF operator**

When manipulating an object reference with a variable defined with a superclass of the real class used to instantiate the object, you sometimes need to identify the real class of the object.
This is possible with the `INSTANCEOF` operator.

This operator checks whether the left operand is an instance of the type or class specified by the right operand:

```
object_reference INSTANCEOF type_or_class
```

This example creates a `java.lang.StringBuffer` object, assigns the reference to a `java.lang.Object` variable, and tests whether the class type of the object reference is a `java.lang.StringBuffer`:

```
IMPORT JAVA java.lang.Object
IMPORT JAVA java.lang.StringBuffer
MAIN
  DEFINE o java.lang.Object
  LET o = StringBuffer.create()
  DISPLAY o INSTANCEOF StringBuffer    -- Shows 1 (TRUE)
END MAIN
```

**Java exception handling**

In order to catch Java exceptions within programs, use a `TRY/CATCH` block.

When a Java exception occurs, the runtime system sets the `STATUS` variable to the error code `-8306`.

The Java exception details (that is the name of the exception) can be found with the `ERR_GET(STATUS)` built-in function.

**Important:** To get the Java exception type with `ERR_GET()`, do not execute other instructions before querying for the error message, otherwise the `STATUS` variable may be reset to zero and the Java exception details would be lost.

To easily identify the type of the Java exceptions in your code, consider writing a library function based on `ERR_GET()`, that recognizes most common Java exceptions, and converts them to integer codes:

```
IMPORT JAVA java.lang.StringBuffer
IMPORT JAVA java.lang.Object
IMPORT JAVA java.lang.StringBuffer
IMPORT JAVA java.lang.StringBuffer
MAIN
  DEFINE sb java.lang.StringBuffer
  LET sb = StringBuffer.create("abcdef")
  TRY
    CALL sb.deleteCharAt(50) -- out of bounds!
  CATCH
    DISPLAY err_get(STATUS)
    EXIT PROGRAM 1
  END TRY
END MAIN
```

**Note:** As a general pattern, do not use `TRY/CATCH` or `WHENEVER ERROR CONTINUE` exception handlers if no exception is supposed to occur. By default the program will then stop and display the Java exception details.

**Executing Java code with GMA**

On Android™ devices, Genero apps can use the Java interface.

**Note:** The GMA executes a program in a JVM process and therefore does not require more resources to execute Java code.

We distinguish the following use cases where the Java interface of Genero can be used in GMA:

- Use classes from the standard Java or Android Java library.
- Implement and use user-defined Java classes, requiring GMA packaging.
- Implement and execute a user-defined Android activity, requiring GMA packaging.

Java may also be used to extend the GMA front-end with user-defined front calls. For details, see Implement front call modules for GMA on page 2254.
Standard Java and Android library usage
You can use Java classes that are part of the standard Java library and Android™ Java library.

Using standard Java within the GMA
Java classes provided in the standard Java library and in the Android Java library can be used directly by including the IMPORT JAVA classname keywords in the Genero code:

```
IMPORT JAVA java.lang.Runtime
IMPORT JAVA android.os.Build

MAIN
  DEFINE rtm Runtime, msg STRING
  LET rtm = java.lang.Runtime.getRuntime()
  LET msg = SFMT("Device:[%1] %2 - %3 (%4 procs)",
                  android.os.Build.MANUFACTURER,
                  android.os.Build.MODEL,
                  android.os.Build.SERIAL,
                  rtm.availableProcessors() )

  MENU "Test" ATTRIBUTES(STYLE="dialog", COMMENT=msg)
  ON ACTION ok
    EXIT MENU
  END MENU

END MAIN
```

The Android Java library does not include all the classes of a regular JRE. User interface classes are specific to the Android user interface framework. The list of standard Android Java packages can be found at http://developer.android.com/reference/packages.html.

Only non-interactive classes can be used in this context. To get a graphical user interface, you must implement an Android Activity, as described in Implement Android activities in GMA on page 2222.

Because Android apps are Java-based, the JVM and standard Java library is directly available. There is no need to bundle the Java library with your Genero program files when you deploy your app as .apk package.

When executing the Genero program on a computer in development mode, it is not possible to use classes that are specific to the Android Java library, because the Android Java library is not available in development mode at runtime.

You must compile your app code and deploy it on an Android device for execution. To compile your app code on the development platform, you need to setup the Java SDK environment and the CLASSPATH to the Android SDK library (android.jar).

**Note:** For compilation, JDK_HOME can point to a 32-bit or 64-bit Java Development Kit installation, to match the Genero BDL architecture. However, the Android SDK is only available in 32-bit.

JVM context-dependent Android API calls
On an Android device, the GMA executes a Genero program in a JVM process. Some Android system APIs cannot be directly called from the Genero runtime system context; they must be called from the JVM context.

In order to call such APIs, you must import the com.fourjs.gma.vm.FglRun class and get the Android JVM thread context by calling the getContext() method of the FglRun class.

The getContext() method will return an instance of the android.content.Context class. For more details, see http://developer.android.com/reference/android/content/Context.html

**Note:** To use this Android JVM interface, you must add the android.jar library (from the Android SDK) to the class path.
The `com.fourjs.gma.vm.FglRun` class implements the following methods:

### Table 472: Methods of the `com.fourjs.gma.vm.FglRun` class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getContext()</code></td>
<td>Returns the Android JVM context object of the runtime system.</td>
</tr>
</tbody>
</table>

In the program code, use the `getContext()` method to get the JVM context and call specific Android APIs:

```java
IMPORT JAVA android.app.Service
IMPORT JAVA android.content.Context
IMPORT JAVA android.util.DisplayMetrics
IMPORT JAVA android.view.WindowManager
IMPORT JAVA com.fourjs.gma.vm.FglRun

MAIN
    DEFINE w, h, d INT
    MENU "Java"
        ON ACTION test
            CALL android_screen_metrics() RETURNING w, h, d
            MESSAGE "Width: ", w, "\nHeight: ", h, "\nDensity: ", d
        END MENU
    END MAIN

FUNCTION android_screen_metrics()
    DEFINE ctx android.content.Context,
         dm android.util.DisplayMetrics,
         wm android.view.WindowManager

    LET ctx = com.fourjs.gma.vm.FglRun.getContext()
    LET dm = android.util.DisplayMetrics.create()
    LET wm = CAST ( ctx.getSystemService("window")
                    AS android.view.WindowManager )
    CALL wm.getDefaultDisplay().getMetrics(dm)

    RETURN dm.widthPixels,
           dm.heightPixels,
           dm.densityDpi
END FUNCTION
```

### Using front calls instead of pure Java

For maximum portability, consider implementing Android-specific extensions as custom front calls. When using the front call technology, apps can be executed in development (app running on the server) and in deployed mode (app running on the mobile device) with the same Genero code.

For more details about custom front calls with GMA, see Implement front call modules for GMA on page 2254.

### Implement Java user extensions in GMA

A GMA app can execute custom Java code.

In order to execute Java user code on the mobile device, the compiled Java classes need to be available to the Genero runtime system. They can then be imported with the `IMPORT JAVA classname` instruction.

When executing the Genero program on a computer in development mode, define the CLASSPATH to your `.jar` files. This allows the JVM loaded by the Genero runtime system find the appropriate Java classes.

When executing the Genero program on a mobile device, the compiled user Java classes must be included in the mobile app Android™ package (.apk), which is created in the Genero Studio deployment procedure.
Related concepts
Packaging custom Java extensions for GMA on page 2223
Custom Java extension must be integrated in the GMA to run on Android™ devices.

Implement Android™ activities in GMA
Android™ activities can be bundled with your GMA app and called from the Genero code.

A Java-based extension that interacts with the end user must be implemented as an Android™ Activity, by using the android.app.Android class.

In order to use your Android™ Activity from the program, it must be integrated in the mobile app Android™ package (.apk), which is created in the Genero Studio deployment procedure.

This code example implements a simple Android™ Activity:

```java
package com.myextension;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class MyActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Button button = new Button(this);
        button.setText("Quit");
        setContentView(button);
        button.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                int resultCode = RESULT_OK;
                Intent resultData = new Intent();
                resultData.putExtra("MyKey", "MyValue");
                setResult(resultCode, resultData);
                finish();
            }
        });
    }
}
```

In order to execute this activity from a Genero app, use the `startActivity` front call:

```
MAIN
    DEFINE data, extras STRING
    MENU
        ON ACTION activity ATTRIBUTES(TEXT="Call bundled activity")
            CALL ui.Interface.frontCall("android", "startActivityForResult",
                ["android.intent.action.VIEW", NULL, NULL, NULL, 
                "com.myextension.MyActivity"],
                [ data, extras ])
            MESSAGE "data="data," / extras=""extras
        ON ACTION quit
            EXIT MENU
    END MENU
END MAIN
```

Note: The component name (fifth parameter) of the startActivity front call does normally take the APK package name followed by the Java Activity class name (apk-package-name/java-class-name). The APK Android™ package name can be defined for the application project in Genero Studio. When using an user-defined activity that is part of the GMA binary archive, do not specify the APK package in the component parameter, because the Java Activity class will be included in the current APK package. This is true when using the customized GMA
front-end in development mode, and in the final application that is deployed on the device. For more details about the component parameter, see `startActivity (Android)` on page 2716.

**Related concepts**

Packaging custom Java extensions for GMA on page 2223
Custom Java extension must be integrated in the GMA to run on Android™ devices.

Packaging custom Java extensions for GMA
Custom Java extension must be integrated in the GMA to run on Android™ devices.

Genero Mobile apps for Android are created from Genero Studio, or from the command-line with `gmabuildtool` on page 2096; you need to provide the custom GMA binary archive containing your Java extensions to Genero Studio or to `gmabuildtool`.

- Genero Studio finds the GMA binary archive from the GMADIR variable defined in the configuration settings.
- The `gmabuildtool` requires the Android Studio project directory used to build the custom GMA to be specified with the `--build-project` option.

**Note:** For Genero Mobile 1.30, the minimum version of the JDK installation required is 1.8.

Along with the GMA binary archive, you must provide the `.jar` files of your Java extensions, that will be used to compile Genero application code on the development machine, as well as the `.apk` Android™ packages of GMA, to deploy the front-end part on the device for client/server development (typically with user-defined front calls).

The original GMA binary archive is a zip file containing several `.aar` Android™ libraries. A customized GMA binary archive contains the `.aar` files from the GMA core libraries, the Genero runtime system core libraries, and custom `.aar` files built from your own Java libraries. The custom `.aar` libs are created from Android Studio. The minimum Android Studio version is 0.8.9.

To create a new GMA binary archive, the `extension.jar` file, and the `.apk` packages, including your Java extensions, perform the steps described in Custom GMA binary archive build on page 2223.

After completing these steps:

- When compiling application code, Genero Studio can find your `.jar` libraries to resolve Java symbols.
- When deploying the front-end only for client/server development, Genero Studio will find the `.apk` packages to be installed on the device.
- When building an Android™ app in Genero Studio, it will be created from the custom GMA binary archive that includes your Java extensions.
- When building an Android™ app with `gmabuildtool`, it can be created by specifying the custom GMA Android™ project directory with the `--build-project` option.

**Related concepts**

Implement front call modules for GMA on page 2254
Custom front call modules for the Android™ front-end are implemented by using the API for GMA front calls in Java.

Implement Java user extensions in GMA on page 2221
A GMA app can execute custom Java code.

**Custom GMA binary archive build**

If you are planning to build Genero Mobile for Android™ extensions for your GMA project, you need to do this using Android™ Studio. Follow this procedure to extend GMA.

**Note:** When using GMA user extensions, you need to build the Java binaries for the final custom GMA `.apk`, as well as the `user-extension.jar` library, to compile Genero sources using `IMPORT JAVA user-extension`. For a complete overview, see Packaging custom Java extensions for GMA on page 2223.

**Important:** Android™ Studio must be installed, and minimum Android™ development skills are required to perform this task.

1. Locate the original GMA binary archive on your computer. If using Genero Studio, it is found in the `GMADIR/gma/artifacts` directory. The GMADIR variable is set in the Genero Studio configuration settings. If you are not using Genero Studio, the GMA binary archive is provided as a separate package.
The GMA binary archive consists of a file named fjs-gma-<version>-<build>-android-scaffolding.zip. Version 1.30.00 or later is required.

This file contains the original GMA core binary and the sub-project to build custom extensions.

2. Create a project directory, for example extension-example/gma, to where you extract the GMA scaffolding archive.

3. Unzip the fjs-gma-<version>-<build>-android-scaffolding.zip archive into extension-example/gma/project.

4. In Android™ Studio, open the project from extension-example/gma/project.

   **Note:** In the Android™ Studio Project view, under the extension library, you can find the TestExtensionActivity.java source file: This is a sample activity to test the extension bundled in the scaffold project. This class is provided for testing purpose and sample. Rename it and change the code for your own purpose, or replace it with your own extension source.

5. Define the extension module package name in the Android™ manifest file.

   In the Android™ Studio Project view, under the extension directory, open the AndroidManifest.xml file. Modify the value of the package attribute of the manifest node:

   ```xml
   <manifest xmlns:android="http://schemas.android.com/apk/res/android"
     package="com.gma.extension">
   </manifest>
   ``

   **Note:** The package name must be a Java-language-style package name, for example, in the format "com.gma.extension". The package name identifies your extension library, it is not the name used to build a final app in Genero Studio.

6. Add your Java sources to this Android™ Studio project, under the extension library.

7. If required, add external .jar libraries to the project: Copy the additional jar libraries into extension-example/gma/project/extension/libs. These jar libraries will be included when building the project.

8. In Android™ Studio, build the project in release or debug mode.

   This creates the custom extensions binaries to build the final .apk and the extension.jar file in extension-example/gma/project/extension/build/libs. The extension.jar file contains Java classes that are callable from Genero BDL. The extension.jar file is required to compile Genero source code using IMPORT JAVA extension.

   **Note:** The project build in Android™ Studio will also create the customized GMA .apk. However, this step is done explicitly later, either from Genero Studio, or by using the gmabuildtool command.

9. Build your custom GMA .apk with your custom extensions.

   If using Genero Studio, perform the following steps:

   a) Modify the CLASSPATH environment variable to include the extension.jar file.

      This is required to let the Genero compiler find your Java classes.

   b) Modify the GMADIR configuration variable to point to the extension-example/gma/project directory.

      This is required to let Genero Studio use your customized GMA binary to build apps. For more information, see the Configuration for extending Genero Mobile for Android topic in the Genero Studio User Guide.

   If using gmabuildtool to build apps from the command line:

   - Change directory to extension-example and call the gmabuildtool to build and bundle your final .apk file with the required options provided by the tool: gmabuildtool will use by default find GMA binary files and resources in the current working directory. For more details about gmabuildtool usage, see Building Android apps with Genero on page 3591.

10. Deploy the new GMA Android™ application package (.apk) on the device.

    - When using Genero Studio, the apk packages to be installed on the device are referenced from the GMADIR environment variable.
If not using Genero Studio, you can use the Android™ debug bridge (adb) command-line utility included with Android™ SDK to install the package to your device.

**Examples**

Java interface usage examples.

**Example 1: Using the regex package**

```java
IMPORT JAVA java.util.regex.Pattern
IMPORT JAVA java.util.regex.Matcher
MAIN
  DEFINE p Pattern
  DEFINE m Matcher
  LET p = Pattern.compile("[a-z]+,[a-z]+")
  DISPLAY p.pattern()
  LET m = p.matcher("aaa,bbb")
  IF m.matches() THEN
    DISPLAY "The string matches the pattern..."
  ELSE
    DISPLAY "The string does not match the pattern..."
  END IF
END MAIN
```

**Example 2: Using the Apache POI framework**

This example shows how to create a XLS file, using the Apache POI framework.

**Note:** This demo requires Apache POI version 4.0.0 or higher.

Download and install the Apache POI package, set CLASSPATH environment variable to point to the POI JAR archives.

After execution, a file named "itemlist.xls" is found in the current directory.

```java
-- Needs Apache POI 4.0.0 +
IMPORT JAVA java.io.FileOutputStream
IMPORT JAVA org.apache.poi.hssf.usermodel.HSSFWorkbook
IMPORT JAVA org.apache.poi.hssf.usermodel.HSSFSheet
IMPORT JAVA org.apache.poi.hssf.usermodel.HSSFRow
IMPORT JAVA org.apache.poi.hssf.usermodel.HSSFCell
IMPORT JAVA org.apache.poi.hssf.usermodel.HSSFCellStyle
IMPORT JAVA org.apache.poi.hssf.usermodel.HSSFFont
IMPORT JAVA org.apache.poi.ss.usermodel.IndexedColors
IMPORT JAVA org.apache.poi.ss.usermodel.HorizontalAlignment
IMPORT JAVA org.apache.poi.ss.usermodel.FillPatternType
IMPORT JAVA org.apache.poi.ss.usermodel.CellType
MAIN
  DEFINE fo FileOutputStream
  DEFINE workbook HSSFWorkbook
  DEFINE sheet HSSFSheet
  DEFINE row HSSFRow
  DEFINE cell HSSFCell
  DEFINE style HSSFCellStyle
  DEFINE headerFont HSSFFont
  DEFINE i, id INTEGER, s STRING
  LET workbook = HSSFWorkbook.create()
  LET style = workbook.createCellStyle()
  CALL style.setAlignment(HorizontalAlignment.CENTER)
  CALL style.setFillForegroundColor((
```
Example 3: Using Java on Android™

This example shows how to access Android™ components through Java, it includes:

- Access to the JDK API to get the number of cores on your device.
- Access to Android™ APIs to get the screen dimension, the device manufacturer and model (with no need for any additional authorization)
- Access to the Bluetooth stack to list the paired devices.

**Note:** In your GM project, you need to ask for BLUETOOTH authorization.

Form file `formJavaStandard.per`:

```
LAYOUT (TEXT="Access to Android API")
GROUP group1(TEXT="Using standard JDK API...")
GRID grid1
{
  [l1                        |f1                                  ]
}
END
END

ATTRIBUTES
LABEL l1 : label1, TEXT="Number of processors available";
LABEL f1 = FORMONLY.nb_proc;
END
```

Form file `formAndroidSimple.per`:

```
LAYOUT (TEXT="Access to Android API")
```
GROUP group1(TEXT="Using simple Android API...")
GRID grid1
{
[11  f1  ]
[12  f2  ]
[13  f3  ]
[14  f4  ]
}
END
END

ATTRIBUTES
LABEL l1 : label1, TEXT="Device manufacturer";
LABEL f1 = FORMONLY.manufacturer;
LABEL l2 : label2, TEXT="Device model";
LABEL f2 = FORMONLY.model;
LABEL l3 : label3, TEXT="Device serial number";
LABEL f3 = FORMONLY.serial;
LABEL l4 : label4, TEXT="Device screen dimension";
LABEL f4 = FORMONLY.diagonal;
END

Form file formAndroidBluetooth.per:

LAYOUT (TEXT="Access to Android API")
GROUP group1(TEXT="Using Bluetooth Android API...")
GRID grid1
{
[11  f1  ]
<TABLE t>
[1  c1  ]
[1  c2  ]
[1  c2  ]
<
[1  c2  ]
>
END
END

ATTRIBUTES
LABEL l1 : label1, TEXT="Bluetooth adapter name";
LABEL f1 = FORMONLY.ba_name;
LABEL c1 = FORMONLY.name;
LABEL c2 = FORMONLY.comment;
END

INSTRUCTIONS
SCREEN RECORD list(FORMONLY.name, FORMONLY.comment);
END

Program file:

IMPORT util
IMPORT JAVA java.lang.Runtime
IMPORT JAVA java.util.Iterator
IMPORT JAVA java.lang.Class
IMPORT JAVA java.lang.Math
IMPORT JAVA android.bluetooth.BluetoothAdapter
IMPORT JAVA android.bluetooth.BluetoothDevice
IMPORT JAVA android.content.Context
IMPORT JAVA android.os.Build
IMPORT JAVA android.util.DisplayMetrics
IMPORT JAVA android.view.WindowManager

IMPORT JAVA com.fourjs.gma.vm.FglRun

MAIN
MENU "Samples"
  COMMAND "Android API access"
  CALL androidApiAccess()
  COMMAND "Quit"
  EXIT MENU
ON ACTION close
  EXIT MENU
END MENU
END MAIN

FUNCTION androidApiAccess()

MENU "Android API access"
  COMMAND "Accessing Java standard API"
  CALL androidApiAccess_java_standard()
  COMMAND "Accessing simple android information"
  CALL androidApiAccess_android_simple()
  COMMAND "Accessing sophisticated APIs : bluetooth"
  CALL androidApiAccess_bluetooth()
ON ACTION CANCEL
  EXIT MENU
END MENU
END FUNCTION

FUNCTION androidApiAccess_java_standard()

DEFINE r Runtime

OPEN WINDOW w WITH FORM "formJavaStandard"

LET r = java.lang.Runtime.getRuntime()
DISPLAY r.availableProcessors() TO nb_proc

MENU
  ON ACTION QUIT
    EXIT MENU
  ON ACTION close
    EXIT MENU
END MENU

CLOSE WINDOW w
END FUNCTION

FUNCTION androidApiAccess_android_simple()

DEFINE s STRING
DEFINE dm DisplayMetrics
DEFINE c Context
DEFINE width, height, dens, wi, hi, x, y FLOAT
DEFINE screenInches FLOAT
DEFINE wm android.view.WindowManager

OPEN WINDOW w WITH FORM "formAndroidSimple"

LET s = android.os.Build.MANUFACTURER
DISPLAY s TO manufacturer
LET s = android.os.Build.MODEL
DISPLAY s TO model
LET s = android.os.Build.SERIAL
DISPLAY s TO serial
# Get the FglRun Context
LET c = com.fourjs.gma.vm.FglRun.getContext()

# Compute display dimension (diagonal)
LET dm = android.util.DisplayMetrics.create()
LET wm = CAST ( c.getSystemService("window") AS android.view.WindowManager )
CALL wm.getDefaultDisplay().getMetrics(dm)
LET width = dm.widthPixels
LET height = dm.heightPixels
LET dens = dm.densityDpi
LET wi = width/dens
LET hi = height/dens
LET x = util.Math.pow(wi,2)
LET y = util.Math.pow(hi,2);
LET screenInches = util.Math.sqrt(x+y);

DISPLAY screenInches TO diagonal
MENU
  ON ACTION QUIT
    EXIT MENU
  ON ACTION close
    EXIT MENU
END MENU

CLOSE WINDOW w
END FUNCTION

FUNCTION androidApiAccess_bluetooth()
DEFINE ba BluetoothAdapter
DEFINE sbd Iterator
DEFINE bd BluetoothDevice
DEFINE bds DYNAMIC ARRAY OF RECORD
  name STRING,
  comment STRING
END RECORD
DEFINE i INTEGER
DEFINE s STRING

OPEN WINDOW w WITH FORM "formAndroidBluetooth"
LET ba = android.bluetooth.BluetoothAdapter.getDefaultAdapter()
LET s = ba.getName()
DISPLAY s TO ba_name
LET sbd = ba.getBondedDevices().iterator()
LET i = 0
WHILE sbd.hasNext()
  LET bd = CAST(sbd.next() AS BluetoothDevice)
  LET i = i + 1
  LET bds[i].name = bd.getName()
  LET bds[i].comment = bd.getBluetoothClass().toString()
END WHILE

DISPLAY ARRAY bds TO list.*
ON ACTION QUIT
  EXIT DISPLAY
ON ACTION close
  EXIT DISPLAY
END DISPLAY

CLOSE WINDOW w
END FUNCTION
C-Extensions

With *C-Extensions*, you can bind your own C libraries in the runtime system, to call C function from the application code.

**Understanding C-Extensions**

C-Extensions allow you to write custom shared libraries in the C language.

Using *C-Extensions*, C functions implemented in shared libraries can be called from the Genero application code. This feature allows you to extend the language with custom libraries, or existing standard libraries, by writing some 'wrapper functions' to interface with the Genero language.

On regular platforms, C-Extensions are implemented with shared libraries, that are loaded by the `fglrun` program on demand.

**Note:** Platforms such as iOS mobile devices do not allow you to load shared libraries. In this case, you must re-link the virtual machine. For more details, see Implementing C-Extensions for GMI on page 2247.

Function parameters and returned values are passed/returned on the runtime stack, using *pop/push functions*. Be sure to pop and push the exact number of parameters/returns expected by the caller; otherwise, a fatal stack error will be raised at runtime.

In order to use a C-Extension in your program, you typically specify the library name with the `IMPORT` instruction at the beginning of the module calling the C-Extension functions. The compiler can then check for the existence of the functions and the library will be automatically loaded at runtime.

**Note:**
- The C code written in C-Extensions is usually platform specific, which does not ease the migration of your application to a different operating system, especially when doing a lot of system calls. Additionally, C data types are defined differently depending on the processor architecture (32 / 64 bits issues). This can also be an issue.
- Make sure that the functions defined in your C-Extensions do not conflict with program functions. In case of conflict, you will get a compiler or a runtime error, depending on the *loading technique used*.

**Related concepts**

IMPORT C-Extension on page 481

The `IMPORT` instruction imports c extension module elements to be used by the current module.

**Header files for ESQL/C typedefs**

C header files (.h) are required to define C structures for complex data types used in a C-Extension.

To compile C-Extensions using data types such as DECIMAL, DATETIME/INTERVAL or BYTE/TEXT, you need IBM® Informix® ESQL/C data type structure definitions such as `dec_t`, `dtime_t`, `intrvl_t`, as well as macros such as `TU_ENCODE()`.

These definitions are not required, if you use standard C types such as `short`, `int` or `char[]`.

The definition of the ESQL/C structures like `dec_t` are provided in individual header files, under the `$FGLDIR/include/f2c` directory: `fglDecimal.h`, `fglDatetime.h`, `fglLocator.h`.

In order to include these data type header files, simply include the `fglExt.h` header file:

```c
#include "f2c/fglExt.h"
```

The other header files are then included automatically.

**Related concepts**

Data types and structures on page 2239
C types are used to write C-Extensions.

Creating C-Extensions

Custom C-Extensions must be provided to the runtime system as Shared Objects (.so) on UNIX™, and as Dynamically Loadable Libraries (.DLL) on Windows®.

In order to create a C-Extension, you must:

1. Define the list of user functions in the C interface file, by including the fglExt.h header file.
2. Compile the C interface file with your C compiler.
3. Modify your C source modules by including the fglExt.h header file.
4. Compile the C interface file and the C modules with the position-independent code option.
5. Create the shared library with the compiled C interface file and C modules by linking with the libfgl runtime system library.

Include the fglExt.h header file in the following way:

```
#include "f2c/fglExt.h"
```

When migrating from IBM® Informix® 4GL, it is possible that existing C-Extension sources include Informix® specific headers like sqlhdr.h or decimal.h. You can either remove or keep the original includes, but if you want to keep them, the Informix® specific header files must be included before the fglExt.h header file, in order to let fglExt.h detect that typedefs such as dec_t or dt ime_t are already defined by Informix® headers. If you include Informix® headers after fglExt.h, you will get a compilation error.

**Note:** Since fglExt.h defines all Informix-like typedef structures, you can remove the inclusion of Informix® specific header files.

The C functions that are implemented in the C-Extension libraries must be known by the runtime system. To do so, each C-Extension library must publish its functions in a UsrFunction array, which is read by the runtime system when the module is loaded. The UsrFunction array describes the user functions by specifying the name of the function, the C function pointer, the number of parameters and the number of returned values. You typically define the UsrFunction array in the C interface file.

After compiling the C sources, you must link them together with the libfgl runtime system library.

Carefully read the man page of the ld dynamic loader, and any documentation of your operating system related to shared libraries. Some platforms require specific configuration and command line options when linking a shared library, or when linking a program using a shared library (+s option on HP® for example).

**Linux®** command-line example:

```
gcc -c -I $FGLDIR/include -fPIC myext.c
gcc -c -I $FGLDIR/include -fPIC cinterf.c
gcc -shared -o myext.so myext.o cinterf.o -L$FGLDIR/lib -lfgl
```

**Windows®** command-line example using Visual C 8.0 and higher (with SxS manifest for the DLL!):

```
c1 /DBUILDDLL /I%FGLDIR%/include /c myext.c
c1 /DBUILDDLL /I%FGLDIR%/include /c cintref.c
link /dll /manifest /out:myext.dll myext.obj cinterf.obj %FGLDIR%\lib\libfgl.lib
mt -manifest myext.dll.manifest -outputresource:myext.dll
```

If you build your DLL with a version of Microsoft™ Visual C++ that is different from the version used to build FGLRUN.EXE, the DLL must get private dependencies other than the process default. For example, when the C-Extension DLL needs the Visual C 9.0 runtime library MSVCR90.DLL, while the FGLRUN.EXE was built with VC 10 and needs MSVCR100.DLL. Private dependencies are specified with the resource
id ISOLATION_AWARE_MANIFEST_RESOURCE_ID, by adding the ;2 modifier at the end of the -
outputresource option, after the file name:

```
mt -manifest myext.dll.manifest -outputresource:myext.dll;2
```

To simplify compilation and linking of a C-Extension library, it is also possible to use the fglmkext command line tool:

```
fglmkext -o myext.so module_a.c module_b.c
```

**Note:** The fglmkext command line tool contains platform-specific C compiler and linker options required to build a C Extension library.

**Related concepts**

- fglmkext on page 2077
- The fglmkext tool compiles and links a user C Extension.
- Header files for ESQL/C typedefs on page 2230
- C header files (.h) are required to define C structures for complex data types used in a C-Extension.

**Creating Informix® ESQL/C Extensions**

C-Extension libraries can be created from ESQL/C sources, as long as you have an Informix® ESQL/C compiler which is compatible with your Genero runtime system.

In order to create a C-Extension from ESQL/C sources, you must:

1. Define the list of user functions in the C interface file, by including the fglExt.h header file.
2. Compile the C interface file with your C compiler.
3. Modify your ESQL/C source modules by including the fglExt.h header file.
4. Compile the ESQL/C modules with the esql compiler, with the position-independent code option.
5. Create the shared library with the compiled C interface file and ESQL/C modules by linking with the libfgl runtime system library, and with the ESQL/C libraries (esql -libs), to resolve the ESQL/C symbols.

Include the fglExt.h header file in the following way:

```
#include "f2c/fglExt.h"
```

You can compile .ec extensions with the native Informix® esql compiler. This section describes how to use the Informix® esql compiler.

The following example shows how to compile and link an extension library with Informix® esql compiler:

**Linux® command-line example:**

```
esql -c -I$FGLDIR/include myext.ec
gcc -c -I$FGLDIR/include -fPIC cinterf.c
gcc -shared -o myext.so myext.o cinterf.o -L$FGLDIR/lib -lfgl -L$INFORMIXDIR/lib/esql `esql -libs`
```

**Windows® command-line example (using Microsoft™ Visual C++):**

```
esql -c myext.ec -I%FGLDIR%/include
c /DBUILDDLL /I%FGLDIR%/include /c cintref.c
esql -target:dll -o myext.dll myext.obj cinterf.obj %FGLDIR%\lib\libfgl.lib
```

When using Informix® esql, you link the extension library with Informix® client libraries. These libraries will be shared by the extension module and the Informix® database driver loaded by the Genero runtime system. Since both the extension functions and the runtime database driver use the same functions to execute SQL queries, you can share the current SQL connection opened in the Genero program to execute SQL queries in the extension functions. However, mixing connection management instructions (DATABASE, CONNECT TO) as well as database creation
can produce unexpected results. For example you cannot do a `CREATE DATABASE` in your ESQL/C extension, and expect that the main program can use this database to execute SQL statements.

**The C interface file**

To make your C functions visible to the runtime system, you must define all the functions in the *C interface file*.

The *C interface file* is a C source file that defines the *usrFunctions* array. This array defines C functions that can be called from programs.

The last record of the *usrFunctions* array must be a line with all the elements set to NULL/0, to define the end of the list.

Each element of the *usrFunctions* array must include the following members:

1. The first member is the name of the function, provided as a *(const char *) character string.
2. The second member is the C function symbol, provided as an *(int (*function) (int)) C function pointer.*
3. The third member is the number of parameters passed to the function through the runtime stack, provided as an *(int); use a negative value like -1 to specify a variable number of arguments.*
4. The fourth member is the number of values returned by the function, provided as an *(int); use a negative value like -1 to specify a variable number of return values.*

**Note:** The third and fourth member of a *UsrFunction* element can be defined as a negative value (-1), to indicate a variable number of arguments and/or return values.

You typically do a forward declaration of your C functions, before the *usrFunctions* array initializer:

```c
#include "f2c/fglExt.h"

int c_init(int);
int c_set_trace(int);
int c_get_message(int);
int c_compare(int);
int c_generate(int);

UsrFunction usrFunctions[]={
    { "init",         c_init,         0, 0 },
    { "set_trace",    c_set_trace,    1, 0 },
    { "get_message",  c_get_message,  1, 1 },
    { "compare",      c_compare,     -1, 1 }, /* var. numb. params */
    { "generate",     c_generate,     1,-1 }, /* var. numb. returns */
    { NULL,           NULL,           0, 0 }
};
```

Note that the *UsrFunction* structure contains an additional member, dedicated for internal use. If you experience compiler warnings because of uninitialized structure members, simply complete the C function definitions with a fifth zero value:

```c
/* Avoids C compiler warnings because of un-initialized structure members */

UsrFunction usrFunctions[]={
    { "init",         c_init,         0, 0, 0 },
    /* member for internal use ---^ */
    ...
```
Linking programs using C-Extensions

When creating a 42r program or 42x library, the linker needs to resolve all function names, including C-Extension functions.

If extension modules are not specified explicitly in the source files with the IMPORT directive, you must give the extension modules with the -e option in the command line:

```
fgllink -e myext1,myext2,myext3 -o myprog.42r moduleA.42m moduleB.42m ...
```

The -e option of fgllink does not write C-Extension references into the .42r file. If you use the -e argument with the fgllink command, you must also use the -e argument with the fgIlrun command, in order to load the libraries at runtime.

The -e option is not needed when using the default userextension module, or if C-Extensions are specified with the IMPORT directive.

Related concepts

fgllink on page 2076
The fgllink tool assembles p-code modules produced with fglcomp into a .42r program or a .42x library.

Loading C-Extensions at runtime on page 2234
The runtime system can load several C-Extensions libraries, allowing you to properly split your libraries by defining each group of functions in separate C interface files.

Compiling source files on page 2106
Describes how to build the runtime files from source files.

IMPORT C-Extension on page 481
The IMPORT instruction imports c extension module elements to be used by the current module.

Loading C-Extensions at runtime

The runtime system can load several C-Extensions libraries, allowing you to properly split your libraries by defining each group of functions in separate C interface files.

Note: When running iOS platforms, the C-Extensions are linked statically to the GMI application.

Directories are searched for the C-Extensions libraries based on the FGLLDPATH on page 278 environment variable rules.

If the C-Extension library depends on other shared libraries, make sure that the library loader of the operating system can find these shared objects. You may need to set the LD_LIBRARY_PATH environment variable on UNIX™ or the PATH environment variable on Windows® to point to the directory where these other libraries are located.

There are three ways to bind a C-Extension with the runtime system:

1. Using the IMPORT instruction in sources.
2. Using the default C-Extension name: userextension(.so or .DLL)
3. Using the -e option of fgIlrun.

Using the IMPORT instruction

The IMPORT instruction allows you to declare an external module in a .4gl source file. It must appear at the beginning of the source file.

The name of the module specified after the IMPORT keyword is converted to lowercase by the compiler. Therefore it is recommended to use lowercase file names only.

The compiler and the runtime system automatically know which C-Extensions must be loaded, based on the IMPORT instruction:

```
IMPORT mylib1
MAIN
```
CALL myfunc1("Hello World")  -- C function defined in mylib1
END MAIN

When the IMPORT instruction is used, no other action has to be taken at runtime. The module name is stored in the 42m p-code and is automatically loaded when needed.

Using the default C-Extension name

It is recommended that all modules using a function from a C-Extension now use the IMPORT instruction. However, this can mean a major change to existing sources.

To simplify migration of existing C-Extensions, the runtime system loads by default a module with the name userextension. Create this shared library with your existing C-Extensions, and the runtime system will load it automatically if found in the directories specified by FGLLDPATH.

Using the -e fglrun option

In some cases you need several C-Extension libraries, which are used by different groups of programs, so you cannot use the default userextension solution. However, you don't want to review all your sources in order to use the IMPORT instruction.

You can specify the C-Extensions to be loaded by using the −e option of fglrun. The −e option takes a comma-separated list of module names, and can be specified multiple times in the command line. The following example loads five extension modules:

```
fglrn −e myext1,myext2,myext3 −e myext4,myext5 myprog.42r
```

By using the −e option, the runtime system loads the modules specified in the command line instead of loading the default userextension module.

Related concepts

IMPORT C-Extension on page 481

The IMPORT instruction imports c extension module elements to be used by the current module.

Runtime stack functions

To pass values between a C function and a program, the C function and the runtime system use the runtime stack.

Stack function basics

The parameters passed to the C function must be popped from the stack at the beginning of the C function, and the return values expected by the Genero BDL call must be pushed on the stack before leaving the C function.

The int parameter of the C function defines the number of input parameters passed on the stack, and the function must return an int value defining the number of values returned on the stack.

Note: If you don't pop / push the specified number of parameters / return values, you corrupt the stack and get a fatal error.

Pop parameters from the stack

The runtime system library includes a set of functions to retrieve the values passed as parameters on the stack. This table shows the library functions provided to pop values from the stack into C buffers:

<p>| Table 473: Library functions provided to pop values from the stack into C buffers |
|-----------------------|----------------|----------------|
| Function              | Data type      | Details            |
| void popdate(int4 *dst); | DATE          | 4-byte integer value corresponding to days since 12/31/1899. |</p>
<table>
<thead>
<tr>
<th>Function</th>
<th>Data type</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>void popbigint(bigint *dst);</td>
<td>BIGINT</td>
<td>8-byte integer value.</td>
</tr>
<tr>
<td>void popint(mint *dst);</td>
<td>INTEGER</td>
<td>System dependent integer value (int)</td>
</tr>
<tr>
<td>void popshort(int2 *dst);</td>
<td>SMALLINT</td>
<td>2-byte integer value</td>
</tr>
<tr>
<td>void poplong(int4 *dst);</td>
<td>INTEGER</td>
<td>4-byte integer value</td>
</tr>
<tr>
<td>void popflo(float *dst);</td>
<td>SMALLFLOAT</td>
<td>4-byte floating point value</td>
</tr>
<tr>
<td>void popdub(double *dst);</td>
<td>FLOAT</td>
<td>8-byte floating point value</td>
</tr>
<tr>
<td>void popdec(dec_t *dst);</td>
<td>DECIMAL</td>
<td>See structure definition in $FGLDIR/include/f2c headers</td>
</tr>
<tr>
<td>void popquote(char *dst, int size);</td>
<td>CHAR(n)</td>
<td>The size parameter defines the size of the char buffer (with the '0'). The trailing blanks are kept.</td>
</tr>
<tr>
<td>void popvchar(char *dst, int size);</td>
<td>VARCHAR(n)</td>
<td>The size parameter defines the size of the char buffer (with the '0'). The trailing blanks are kept.</td>
</tr>
<tr>
<td>void popstring(char *dst, int size);</td>
<td>VARCHAR(n)</td>
<td>The size parameter defines the size of the char buffer (with the '0'). This function trims all the trailing spaces, even the last one. There is no way to distinguish from NULL if the string has only spaces.</td>
</tr>
<tr>
<td>void popdtime(dtime_t *dst, int size);</td>
<td>DATETIME</td>
<td>See structure definition in $FGLDIR/include/f2c headers</td>
</tr>
<tr>
<td>void popinv(intrvl_t *dst, int size);</td>
<td>INTERVAL</td>
<td>See structure definition in $FGLDIR/include/f2c headers</td>
</tr>
<tr>
<td>void poplocator(ifx_loc_t **dst);</td>
<td>BYTE, TEXT</td>
<td>See structure definition in $FGLDIR/include/f2c headers</td>
</tr>
</tbody>
</table>

**Important:** This function pops the pointer of a ifx_loc_t object.

When using a pop function, the value is copied from the stack to the local C variable and the value is removed from the stack.
In a Genero program, strings (CHAR, VARCHAR) are not terminated by '\0'. Therefore, the C variable must have one additional character to store the '\0'. For example, the equivalent of a VARCHAR(100) in Genero BDL programs is a char x[101] in C.

**Stack introspection**

A set of C API functions are provided to query information on the parameters passed on the stack to a C function. Query for the parameter type and the actual size of a character string value, to adapt the buffer receiving the parameter.

**Table 474: Library functions to introspect the runtime stack**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>const char *fglcapi_peekStackType(void)</td>
<td>Returns the type name of the topmost value on the stack as a string.</td>
</tr>
<tr>
<td></td>
<td>For example, if the value on the stack is a CHAR(100), the function returns the string &quot;CHAR(100)&quot;.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> If the current value on the stack is a string literal, the returned type name is &quot;STRING&quot;.</td>
</tr>
<tr>
<td>int fglcapi_peekStackBufferSize(void)</td>
<td>Returns the size (in bytes) for the topmost value on the stack, to allocate a C char buffer, when using pop* function to get character strings.</td>
</tr>
<tr>
<td></td>
<td>String pop functions such as popvchar() and popvchar() require a C char buffer to be allocated. To allocate the buffer dynamically, use the fglcapi_peekStackBufferSize() function to get the required buffer size.</td>
</tr>
<tr>
<td></td>
<td>Allocating char buffers with the proposed size avoids truncating values returned from the stack.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> When the argument type is different from CHAR, VARCHAR and STRING, the size is the total number of bytes needed for the string representation of the maximal possible value. For example, with a DECIMAL(32, 5), the size would be 35 bytes (including sign, decimal separator and string terminator).</td>
</tr>
</tbody>
</table>

**Stack introspection example:**

```c
int my_function(int n)
{
    int sz;
    char *buf;
    sz = fglcapi_peekStackBufferSize();
    buf = malloc(sz);
    popstring(buf, sz);
    // ...
    free(buf);
    return 0;
}
```

**Push returns on the stack**

To return a value from the C function, you must use one of the functions provided in the runtime system library.
Table 475: Functions provided in the runtime system library to return a value from a C function

<table>
<thead>
<tr>
<th>Function</th>
<th>Data type</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>void pushdate(int4 val);</td>
<td>DATE</td>
<td>4-byte integer value corresponding to days since 12/31/1899.</td>
</tr>
<tr>
<td>void pushbigint(bigint val);</td>
<td>BIGINT</td>
<td>8-byte integer value.</td>
</tr>
<tr>
<td>void pushdec(const dec_t *val, const unsigned decp);</td>
<td>DECIMAL</td>
<td>See structure definition in $FGLDIR/include/f2c headers</td>
</tr>
<tr>
<td>void pushint(mint val);</td>
<td>INTEGER</td>
<td>System dependent integer value (int)</td>
</tr>
<tr>
<td>void pushlong(int4 val);</td>
<td>INTEGER</td>
<td>4-byte integer value</td>
</tr>
<tr>
<td>void pushshort(int2 val);</td>
<td>SMALLINT</td>
<td>2-byte integer value</td>
</tr>
<tr>
<td>void pushflo(float *val);</td>
<td>SMALLFLOAT</td>
<td>4-byte floating point value.</td>
</tr>
<tr>
<td>void pushdub(double *val);</td>
<td>FLOAT</td>
<td>8-byte floating point value.</td>
</tr>
<tr>
<td>void pushquote(const char *val, int len);</td>
<td>CHAR(n)</td>
<td>len = strlen(val) (without '0')</td>
</tr>
<tr>
<td>void pushvchar(const char *val, int len);</td>
<td>VARCHAR(n)</td>
<td>len = strlen(val) (without '0')</td>
</tr>
<tr>
<td>void pushdtime(const dtime_t *val);</td>
<td>DATETIME</td>
<td>See structure definition in $FGLDIR/include/f2c headers</td>
</tr>
<tr>
<td>void pushinv(const intrvl_t *val);</td>
<td>INTERVAL</td>
<td>See structure definition in $FGLDIR/include/f2c headers</td>
</tr>
</tbody>
</table>

When using a push function, the value of the C variable is copied at the top of the stack; therefore the scope and lifespan of the C variable does not matter.

To simplify migration of IBM® I4GL legacy C extensions using ret*() style functions, Genero supports the following synonyms:

**Table 476: Return value functions synonyms**

<table>
<thead>
<tr>
<th>Function</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>void retdate(int4 val)</td>
<td>pushdate</td>
</tr>
<tr>
<td>void retdec(const dec_t *val)</td>
<td>pushdec</td>
</tr>
<tr>
<td>void retmoney(const dec_t *val)</td>
<td>pushdec</td>
</tr>
<tr>
<td>void retint(int val)</td>
<td>pushint</td>
</tr>
<tr>
<td>void retlong(int4 val)</td>
<td>pushlong</td>
</tr>
<tr>
<td>void retshort(int2 val)</td>
<td>pushshort</td>
</tr>
<tr>
<td>Function</td>
<td>Equivalent</td>
</tr>
<tr>
<td>----------</td>
<td>------------</td>
</tr>
<tr>
<td>void retflo(float *val)</td>
<td>pushflo</td>
</tr>
<tr>
<td>void retdub(double *val)</td>
<td>pushdub</td>
</tr>
<tr>
<td>void retquote(const char *val)</td>
<td>pushquote</td>
</tr>
<tr>
<td>void retstring(const char *val)</td>
<td>pushquote</td>
</tr>
<tr>
<td>void retvchar(const char *val)</td>
<td>pushvchar</td>
</tr>
<tr>
<td>void retdtime(const dtime_t *val)</td>
<td>pushdtime</td>
</tr>
<tr>
<td>void retinv(const intrvl_t *val)</td>
<td>pushinv</td>
</tr>
</tbody>
</table>

**Note:** Pay attention to the retdec(), retmoney(), retquote() and retvchar() functions. These do not have the same signature as the equivalent push*() functions.

**Related concepts**

- **Runtime stack** on page 493
  The runtime stack is used to pass/return values to/from functions.
- **Header files for ESQL/C typedefs** on page 2230
  C header files (.h) are required to define C structures for complex data types used in a C-Extension.

**Data types and structures**

C types are used to write C-Extensions.

The following C types are used to write C-Extensions.

**Table 477: C types used to write C-Extensions**

<table>
<thead>
<tr>
<th>Type name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bigint</td>
<td>signed integer with a size of 8 bytes</td>
</tr>
<tr>
<td>int4</td>
<td>signed integer with a size of 4 bytes</td>
</tr>
<tr>
<td>uint4</td>
<td>unsigned integer with a size of 4 bytes</td>
</tr>
<tr>
<td>int2</td>
<td>signed integer with a size of 2 bytes</td>
</tr>
<tr>
<td>uint2</td>
<td>unsigned integer with a size of 2 bytes</td>
</tr>
<tr>
<td>int1</td>
<td>signed integer with a size of 1 byte</td>
</tr>
<tr>
<td>uint1</td>
<td>unsigned integer with a size of 1 byte</td>
</tr>
<tr>
<td>mint</td>
<td>signed machine-dependent C int</td>
</tr>
<tr>
<td>muint</td>
<td>unsigned machine-dependent C int</td>
</tr>
<tr>
<td>mlong</td>
<td>signed machine-dependent C long</td>
</tr>
<tr>
<td>ulong</td>
<td>unsigned machine-dependent C long</td>
</tr>
<tr>
<td>dec_t</td>
<td>DECIMAL data type structure</td>
</tr>
<tr>
<td>dtime_t</td>
<td>DATETIME data type structure</td>
</tr>
<tr>
<td>intrvl_t</td>
<td>INTERVAL data type structure</td>
</tr>
<tr>
<td>ifx_loc_t</td>
<td>TEXT / BYTE locator structure</td>
</tr>
</tbody>
</table>
Basic data types

Basic data types such as bigint, int4 and int2 are provided to define variables that must hold BIGINT (bigint), SMALLINT (int2), INTEGER (int4) and DATE (int4) values. Standard char array can be used to hold CHAR and VARCHAR data.

DATE

No specific typedef exists for the DATE type; you can use the int4 type to store a DATE value.

DECIMAL/MONEY

The dec_t structure is provided to hold DECIMAL and MONEY values.

The internals of dec_t structure can be ignored during C-Extension programming, because decimal API functions are provided to manipulate any aspects of a decimal.

DATETIME

The dtime_t structure holds a DATETIME value.

Before manipulating a dtime_t, you must initialize its qualifier qt_qual, by using the TU_DTENCODE macro:

```c
    dtime_t dt;
    dt.dt_qual = TU_DTENCODE(TU_YEAR, TU_SECOND);
    dtcvasc( "2004-02-12 12:34:56", &dt );
```

INTERVAL

The intrvl_t structure holds an INTERVAL value.

Before manipulating an intrvl_t, you must initialize its qualifier in_qual, by using the TU_IENCODE macro:

```c
    intrvl_t in;
    in.in_qual = TU_IENCODE(5, TU_YEAR, TU_MONTH);
    incvasc( "65234-02", &in );
```

TEXT/BYTE Locator

The ifx_loc_t structure is used to declare host variables for a TEXT/BYTE values (simple large objects). Because the potential size of the data can be quite large, this is a locator structure that contains information about the size and location of the TEXT/BYTE data, rather than containing the actual data.

Table 478: Fields of the ifx_loc_t structure

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loc_indicator</td>
<td>int4</td>
<td>Null indicator; a value of -1 indicates a null TEXT/BYTE value. Your program can set the field to indicate the insertion of a null value. Database client libraries set the value for selects and fetches.</td>
</tr>
<tr>
<td>loc_type</td>
<td>int4</td>
<td>data type - SQLTEXT (for TEXT values) or SQLBYTES (for BYTE values).</td>
</tr>
<tr>
<td>Field name</td>
<td>Data type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>loc_size</td>
<td>int4</td>
<td>Size of the TEXT/ BYTE value in bytes; your program sets the size of the large object for insertions. Database client libraries set the size for selects and fetches.</td>
</tr>
<tr>
<td>loc_loctype</td>
<td>int2</td>
<td>Location - LOCMEMORY (in memory) or LOCFNAME (in a named file). Set loc_loctype after you declare the locator variable and before this declared variable receives the large object value.</td>
</tr>
<tr>
<td>loc_buffer</td>
<td>char *</td>
<td>If loc_loctype is LOCMEMORY, this is the location of the TEXT/ BYTE value; your program must allocate space for the buffer and store its address here.</td>
</tr>
<tr>
<td>loc_bufsize</td>
<td>int4</td>
<td>If loc_loctype is LOCMEMORY, this is the size of the buffer loc_buffer. If you set loc_bufsize to -1, database client libraries will allocate the memory buffer for selects and fetches. Otherwise, it is assumed that your program will handle memory allocation and de-allocation.</td>
</tr>
<tr>
<td>loc_fname</td>
<td>char *</td>
<td>If loc_loc_type is LOCFNAME, this is the address of the path name string that contains the file.</td>
</tr>
</tbody>
</table>

Example

C Extension source (ext1.c):

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include "f2c/fglExt.h"

int lob_size(int);

int lob_size(int pc)
{
    ifx_loc_t *pb1;
    double ratio;
    char *source = NULL;
    char *psource = NULL;
```
```c
int size;

if (pc != 1) exit(1);
poplocator(&pb1);

if (pb1->loc_loctype == LOCMEMORY) {
    psource = pb1->loc_buffer;
    size = pb1->loc_size;
} else if (pb1->loc_loctype == LOCFNAME) {
    int fd;
    struct stat st;
    fd = open(pb1->loc_fname, O_RDONLY);
    fstat(fd, &st);
    size = st.st_size;
    psource = source = (char *) malloc(size);
    read(fd, source, size);
    close(fd);
}
pushint(size);
return 1;
```

Genero program (main.4gl):

```
IMPORT libext1
MAIN
  DEFINE t TEXT
  LOCATE t IN MEMORY
  LET t = "aaaaaaaaaaaaaa"
  DISPLAY lob_size(t)
END MAIN
```

Commands to compile and execute (on Linux):

```
$ gcc -fPIC -c ext1.c -I $FGLDIR/include
$ gcc --shared -o libext1.so ext1.o -L$FGLDIR/lib -lfgl
$ fglcomp main.4gl
$ fglrun main.42m
14
```

**Related concepts**

- Header files for ESQL/C typedefs on page 2230
- C header files (.h) are required to define C structures for complex data types used in a C-Extension.

**NULL handling**

Handling NULL in C-Extensions.

Variables passed from the BDL code can be checked for NULL depending on the data type.

Define macros as follows to check for NULL values:

```c
#define MY_IS_NULL_SMALLFLOAT(v) (isnan(v))
#define MY_IS_NULL_FLOAT(v) (isnan(v))
#define MY_IS_NULL_INTEGER(v) ((unsigned) v == 0x80000000)
#define MY_IS_NULL_SMALLINT(v) ((unsigned) v == 0x800)
#define MY_IS_NULL_CHAR(v) (v[0] == '\0')
#define MY_IS_NULL_DECIMAL(v) (v.dec_pos < 0)
#define MY_IS_NULL_DATETIME(v) (v.dt_dec.dec_pos < 0)
#define MY_IS_NULL_INTERVAL(v) (v.in_dec.dec_pos < 0)
#define MY_IS_NULL_LOCATOR(v) (v.loc_indicator < 0)
```
To return a NULL value from your C extension function, push an empty string on the stack as follows:

```
pushquote("", 0);
```

**Example**

C Extension source (ext1.c):

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "f2c/fglExt.h"

int check_null(int);

UsrFunction usrFunctions[]={
    { "check_null",   check_null,     1, 1 },
    { NULL,           NULL,           0, 0 }
};

#define MY_IS_NULL_DECIMAL(v) (v.dec_pos < 0)

int check_null( int pc )
{
    dec_t price;
    const char *answer;
    int len;
    if (pc != 1) exit(1);
    popdec(&price);
    if (MY_IS_NULL_DECIMAL(price)) {
        answer = "This decimal is NULL";
    } else {
        answer = "This decimal is not NULL";
    }
    len = (int) strlen(answer);
    pushquote(answer, len);
    return 1;
}
```

Genero program (main.4gl):

```4gl
IMPORT libext1
MAIN
    DEFINE d DECIMAL(10,2)
    LET d = NULL
    DISPLAY check_null(d)
    LET d = 12.34
    DISPLAY check_null(d)
END MAIN
```

Commands to compile and execute (on Linux):

```
$ gcc -fPIC -c ext1.c -I $FGLDIR/include
$ gcc --shared -o libext1.so ext1.o -L$FGLDIR/lib -lfgl
$ fglcomp main.4gl
$ fgllrun main.42m
This decimal is NULL
This decimal is not NULL
```
Calling C functions from programs

C-Extensions functions can be called from the program in the same way that you call a BDL function.

The C functions that can be called from programs must use the following signature:

```c
int function-name( int )
```

Here `function-name` must be written in lowercase letters. The `fglcomp` compiler converts all BDL function names (following a `CALL` keyword) to lowercase.

The C function must be declared in the `usrFunctions` array in the C interface file.

**Important:** Parameters and return values must be pushed/popped on the runtime stack, by using the stack functions. Parameters passed to the C function must be popped in the reverse order of the BDL call list: `CALL c_fct( A, B, C ) => pop C, B, A`. However, values returned from the C function must be pushed in the same order as in the BDL returning clause: `push A, B, C => CALL c_fct() RETURNING A, B, C`.

In this code example, the C-Extension module `mycext.c` defines the `c_fct()` function:

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "f2c/fglExt.h"

int c_fct( int n );

UsrFunction usrFunctions[] = {
   {"c_fct", c_fct, 2, 2},
   {0, 0, 0, 0}
};

int c_fct( int n )
{
   int rc;
   float price;
   char name[31];
   if (n != 2) exit(1);
   popflo(&price);
   popvchar(name, sizeof(name));
   printf(">> [%s] price:%f\n", name, price);
   pushint(strlen(name));
   price = price * 2;
   pushflo(&price);
   return 0;
}
```

The C-Extension library is imported by the BDL module with `IMPORT`:

```c
IMPORT mycext

MAIN
   DEFINE len INT, price2 FLOAT
   CALL c_fct("Hand gloves", 120.50)
   RETURNING len, price2
   DISPLAY "len = ", len
   DISPLAY "price2 = ", price2
END MAIN
```

Compilation and execution example on a Linux® system:

```bash
$ gcc -I $FGLDIR/include -shared -fPIC -o mycext.so mycext.c
```
$ fglcomp myprog.4gl

$ fglrun myprog.42m

>> [Hand gloves] price:120.500000
len =          11
price2 =                   241.0

Related concepts

Functions on page 436
Describes user defined functions.

Calling program functions from C

It is possible to call a BDL function from a C-Extension function.

To call a BDL function from a C-Extension function, use the fgl_call macro:

```c
fgl_call ( function-name, nb-params );
```

In this call, `function-name` is the name of the program function to call, and `nb-params` is the number of parameters pushed on the stack for the program function. The `function-name` must be written in lowercase letters; the fglcomp compiler converts all program function names to lowercase.

The `fgl_call()` macro is converted to a function that returns the number of values returned on the stack.

**Important:** Parameters and return values must be pushed/popped on the runtime stack, by using the stack functions. Parameters passed to the BDL function must be pushed in the same order as the BDL parameter list: push A, B, C => FUNCTION fct( A, B, C ). However, values returned from the BDL function must be popped in the reverse order of the BDL return clause: RETURN A, B, C => pop C, B, A.

The `myprog.4gl` BDL module defining the `MAIN` block and the `display_item()` function to be called from the C extension:

```c
IMPORT mycext

MAIN
  CALL c_fct() 
END MAIN

FUNCTION display_item(name, size)
  DEFINE name VARCHAR(30), size INTEGER
  DISPLAY name, size
  RETURN length(name), (size / 100)
END FUNCTION
```

The `mycext.c` C extension module calling the BDL function:

```c
#include <stdlib.h>
#include <stdio.h>
#include "f2c/fglExt.h"

int c_fct( int n );

UsrFunction usrFunctions[]={
  {"c_fct",c_fct,0,0},
  {0,0,0,0}
};

int c_fct( int n )
```
int rc, len;
float size2;
if (n != 0) exit(1);
pushquote("Hand gloves", 11);
pushint(54);
rc = fgl_call( display_item, 2 );
if (rc != 2) exit(1);
popflo(&size2);
popint(&len);
printf(">> %d %f
", len, size2);
return 0;
}

Compilation and execution example on a Linux® system:

$ gcc -I $FGLDIR/include -shared -fPIC -o mycext.so mycext.c
$ fglcomp myprog.4gl
$ fglrun myprog.42m
Hand gloves        54
>> 11 0.540000

Simple C-Extension example

This example shows how to create a C-Extension library on Linux® using gcc.
The command line options to compile and link shared libraries can change depending on the operating system and compiler/linker used.

The "splitext.c" C interface file

#include "f2c/fglExt.h"

int next_token(int);

UsrFunction usrFunctions[]= {
    { "next_token", next_token, 1, 2 },
    { 0,0,0,0 }
};

The "split.c" file

#include <string.h>
#include "f2c/fglExt.h"

int next_token( int in_num );
int next_token( int in_num )
{
    char src[513];
    char *p;
popvchar(src, sizeof(src));
    if (*src == '\0') {
        pushvchar("", 0);
pushvchar("", 0);
    } else {
        p = strchr(src, ' ');
        if (p == NULL) {
            pushvchar(src, strlen(src));
pushvchar("", 0);
        } else {
...
## Implementing C-Extensions for GMI

This section describes how to program C-Extensions for the GMI VM.

**C-Extensions for GMI**

With C-Extensions for GMI, you can address specific needs on iOS platforms, that are not available by default in the Genero language. For example, implement functions to interface with mobile specific hardware like sensors, card readers, scanners, bluetooth, etc.
The runtime system virtual machine build in the GMI for iOS platforms can be extended with the C-Extension technology. The basics to implement C-Extensions are the same for iOS as for UNIX®/Windows® platforms, but there are some differences, explained in this section.

The main difference is that user libraries cannot be loaded dynamically on iOS and thus require a re-link of the GMI binary with the user-defined C-Extension library.

**Writing C-Extension sources for GMI**

C-Extension source files can be organized in several .c or .m files, but the final library name must be userextension.

For a first test, we recommend that you group all your C-Extension functions in a single source file called userextension.m.

In the Objective C source file, add the following lines, to include typical iOS header files:

```c
#include <Foundation/Foundation.h>
#include <UIKit/UIKit.h>
```

The Genero runtime system header file must be included as well:

```c
#include "f2c/fglExt.h"
```

The C-Extension functions must be registered as usual, in a `UsrFunction` array, defining the number of input and output parameters:

```c
UsrFunction usrFunctions[]={
   {"get_user_info",get_user_info,1,1},
   ...
   {NULL,NULL,0,0}
};
```

**Using iOS C-Extensions in your program**

The application code needs to be compiled on the development platform before it is deployed on the iOS device or simulator, by using the C-Extension library build for the development platform.

In your Genero program, import the C-Extension module with `IMPORT userextension`. You can also omit this `IMPORT` instruction, because the runtime system tries to find and load the userextension library by default. Note also that C-Extension functions have a global scope, so you can omit the prefix of the function name with the lib/module name:

```c
IMPORT userextension
MAIN
   DEFINE info STRING
   LET info = get_user_info()
   ...
END MAIN
```

Compiler behavior regarding `IMPORT userextension` usage:

- **With IMPORT userextension**: The compiler can check references to functions defined in the extension. The programmer can qualify a function-name as `userextension.function-name`. But in this case, the `userextension.so` shared library must exist on the development platform.
- **Without IMPORT userextension**: The compiler can not check references to those functions. The compiler does not load the `userextension` module implicitly. C-Extension function names can not be qualified. In this case, the `userextension.so` library is not required for compilation, but it will be needed if the final program is linked, or if you want to execute/test the application in client/server development mode.
Compiling and linking with C-Extensions on the development platform

On the development machine, if you link 42r programs, or if you want the compiler to check for missing symbols (with the -r option), the userextension library must exist in the development environment.

**Note:** At runtime, on the development machine, the extension library will be loaded at first extension function call. But when the application is deployed on the iOS device, the extension library will be part of the GMI/VM binary (because it is statically linked).

To create the userextension library for the development environment, you must build an Objective-C shared library.

If the C-Extension contains iOS API calls, it will not be possible to compile the extension library as is on the development machine: Write conditional pre-processor macros to hide the iOS specific code, and simulate the function behavior for the development platform:

```c
#ifndef EMULATE_IOS
#include <Foundation/Foundation.h>
#include <UIKit/UIKit.h>
#endif
...
int get_user_info(int pc)
{
    char prop[101];
    char value[101];
    int z = (int) sizeof(prop);
    assert(pc==1);
    popvchar(prop, z);
    #ifndef EMULATE_IOS
    ... here goes the iOS specific code ...
    #else
    value[0] = '\0';
    #endif
    pushvchar(value, (int) strlen(value));
    return 1;
}
```

Command line example to create a shared library with the Xcode® environment (note that we define the NOT_IOS_IMPL constant to compile the code without iOS specific API calls):

```
$ cc -shared -o userextension.dylib userextension.c \
-D EMULATE_IOS -I $FGLDIR/include -L $FGLDIR/lib -lfgl
```

Building the iOS app with C extensions

Genero iOS apps are created with the gmibuildtool command-line tool.

In order to build your iOS app with C extensions, you need to create the static library with the staticlib target of $GMIDIR/lib/Makefile-gmi and then pass the static library to the linker with the --extension-libs option of gmibuildtool.

For more details, see Building iOS apps with Genero on page 3606

User-defined front calls

Front-ends can be extended with custom functions to access specific features.

It is possible to implement custom front-end functions to interface with platform-specific features, and use the feature from a Genero program through a front call. For example, you can implement a front-end function module interfacing with a bar code reader, to return bar codes to the Genero program.
This section describes how to implement your own front calls by front-end type. Because each front-end type uses different technologies, you must use native platform APIs to implement front calls.

**Implement front call modules for GDC**

Custom front call modules for the desktop front-end are implemented by using the API for GDC front calls in C language.

**GDC custom front call basics**

In order to extend the GDC with your own front calls, you must be familiar with C++ programming, and have a C++ compiler installed on your development platform.

GDC front call modules must be implemented as a Dynamic Linked Library (.DLL) on Windows® platforms, as a shared library (.so) on Linux®, or as a Dynamic Library (.dyLib) under Mac® Os X. This shared library must be deployed on each platform where the GDC front-end executes.

The GDC is able to automatically load the front call module and find the function, based on the module name and function name used in the Genero BDL front call (ui.Interface.frontCall).

The API for GDC front calls is based on the `frontEndInterface` front call interface structure, that is used to interface with the GDC core, in order to pass/return values to/from a front call.

Follow these steps to implement a custom front call module for the GDC:

1. Create a C source to implement your front call functions.
2. In the front call functions body:
   a. Check the number of parameters passed with the `getParamCount()` function.
   b. Pop parameter values with one of the `pop*()` functions.
   c. Perform the function task.
   d. Push the result values with one of the `push*()` functions.
   e. Return 0 on success, -1 otherwise.
3. Compile and link the shared library.
4. Deploy the shared library to the platform where GDC executes.

**The front call interface structure**

Information required to execute the front call is transmitted to the extension module through the front call interface structure. This structure contains a list of function pointers to:

- manage the stack (push or pop for each handled data type)
- get information about the function (number of in and out parameters)
- get information about the front-end (front call environment variables)

The following defines the front call interface structure:

```c
struct frontEndInterface
{
    short (* getParamCount) ();
    short (* getReturnCount) ();
    void (* popInteger) (long &, short &);
    void (* pushInteger) (const long, short);
    void (* popString) (char *, short &, short &);
    void (* pushString) (const char *, short, short);
    void (* getFrontEndEnv) (const char *, char *, short &);
    void (* popWString) (wchar_t *, short &, short &);
    void (* pushWString) (const wchar_t*, short, short);
};
```

**Important:** The front call interface structure is defined for the C++ language.
Prototype of a front call function implementation

The prototype of each front call function must be:

```c
int function_name ( const struct frontEndInterface &fci );
```

1. `function_name` is the name of your function.
2. `fci` is the front call interface structure.

The `fci` structure will be filled in by the GDC and passed to the custom function. You can then use this structure to pop/push values from/to the stack, and get environment information from the core GDC.

The function must return 0 on success, -1 otherwise.

Front call environment variables

The front call function can query the GDC for front call environment variables, to get information about the context.

The following front call environment variables are supported:

**Table 479: Supported front call environment variables for the GDC**

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>frontEndPath</td>
<td>The path where the GDC front-end is installed.</td>
</tr>
</tbody>
</table>

Module initialization and finalization

The font-call module can define initialization and finalization functions. GDC will automatically call these functions as follows:

- `void initialize();`

  This function is called when the front call module library is loaded. If needed, perform variable initialization and resource allocation in this function.

- `void finalize();`

  This function is called when the GDC front-end stops. If needed, perform resource release in this function.
The API for custom front call implementation

Table 480: Front call interface functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>short getParamCount();</td>
<td>This function returns the number of parameters given to the function called.</td>
</tr>
<tr>
<td>short getReturnCount();</td>
<td>This function returns the number of returning values of the function called.</td>
</tr>
</tbody>
</table>
| void ( * getFrontEndEnv ) (const char * name, char * value, short & length ); | This function is used to get context information from the front-end.  
  - name is the name of the front call environment variable.  
  - value is the char buffer to hold the value of the variable.  
  - length is the actual length of the value. |
| void popInteger(  
  long & value,  
  short & isNull ); | This function is used to get an integer from the stack.  
  - value is the reference to where the popped integer will be set.  
  - isNull indicates whether the parameter is null. |
| void pushInteger(  
  const long value,  
  short isNull ); | This function is used to push an integer on the stack.  
  - value is the value of the integer.  
  - isNull indicates whether the value is null. |
| void popString(  
  char * value,  
  short & length,  
  short & isNull ); | This function is used to get a string from the stack.  
  - value is the pointer where the popped string will be set.  
  - length is the length of the string.  
  - isNull indicates whether the parameter is null. |
| void pushString(  
  const char * value,  
  short length,  
  short isNull ); | This function is used to push a string on the stack.  
  - value is the value of the string.  
  - length the length of the string. A length of -1 indicates that the length is detected based on the content of the string.  
  - isNull indicates whether the parameter is null. |
| void ( * popWString ) (wchar_t *value,  
  short & length,  
  short & isNull); | This function is used to get a WideChar string from the stack.  
  - value is the pointer where the popped string will be set.  
  - length is the length of the string.  
  - isNull indicates whether the parameter is null. |
| void ( * pushWString ) (wchar_t *value,  
  short length,  
  short isNull); | This function is used to push a WideChar string on the stack.  
  - value is the value of the string.  
  - length the length of the string. A length of -1 indicates that the length is detected based on the content of the string.  
  - isNull indicates whether the parameter is null. |
Calling the custom front call from BDL

In the Genero program, use the `ui.Interface.frontCall()` API to call the front-end function. This method takes the front call module name as the first parameter and the front call function name as second parameter. The front call module name is defined by the name of the dynamic library (`module_name.DLL`, `module_name.so` or `module_name.dylib`).

For example, if you implement a front call module with the name "mymodule.so", the Genero program code must use the name "mymodule" as front call module name:

```cpp
CALL ui.Interface.frontCall("mymodule", "myfunction", ["John DOE"], [msg])
```

Deploying the custom front call module

The shared library implementing the custom front call functions must be deployed on the platform where the GDC executes. Copy your custom front call modules in the bin directory of the GDC installation directory (%GDCDIR% \bin).

Example

This example implements a simple front call function that computes the sum of two integer numbers. It takes two parameters and returns two values.

**mymodule.h**:

```c
struct frontEndInterface
{
    short (* getParamCount) ();
    short (* getReturnCount) ();
    void (* popInteger) (long &, short &);
    void (* pushInteger) (const long, short);
    void (* popString) (char *, short &, short &);
    void (* pushString) (const char *, short, short);
    void (* getFrontEndEnv) (const char *, char *, short &);
    void (* popWString) (wchar_t *, short &, short &);
    void (* pushWString) (const wchar_t*, short, short);
};

#ifdef WIN32
#define EXPORT extern "C" __declspec(dllexport)
#else
#define EXPORT extern "C"
#endif

EXPORT void initialize();
EXPORT void finalize();
EXPORT int mysum(const frontEndInterface &fx);
```

**mymodule.cpp**:

```c
#include "mymodule.h"
#include <stdio.h>
#include <string.h>

void initialize() {
}

void finalize() {
}

int mysum(const struct frontEndInterface &fci) {
    long param1, param2;
```
short isNull1, isNull2;
long sum;
char msg[255];
if (fci.getParamCount() != 2 || fci.getReturnCount() != 2) {
    return -1;
}
fci.popInteger(param2, isNull2);
fci.popInteger(param1, isNull1);
sum = param1 + param2;
if (!isNull1 && !isNull2) {
    sum = param1 + param2;
    sprintf(msg, "%d + %d = %d", param1, param2, sum);
} else {
    sum = 0;
    sprintf(msg, "Parameters are NULL");
}
fci.pushInteger(sum, 0);
fci.pushString(msg, strlen(msg), 0);
return 0;

To invoke the sum front-end function, use the ui.Interface.frontCall() method in your Genero program:

MAIN
DEFINE res INT, msg STRING
MENU
    ON ACTION frontcall ATTRIBUTES(TEXT="Call custom front call")
        CALL ui.Interface.frontCall("mymodule", "mysum",
            [100,250], [res,msg])
            DISPLAY "Result: ", res, "\n", msg
    ON ACTION quit
        EXIT MENU
END MENU
END MAIN

Related concepts
Genero Desktop Client front calls on page 2668
This section describes GDC-specific front calls.

Implement front call modules for GMA
Custom front call modules for the Android™ front-end are implemented by using the API for GMA front calls in Java.

GMA custom front call basics
In order to extend the GMA with your own front calls, you must be familiar with Java programming concepts, and if you want to interface with Android™ apps, understand concepts such as Android™ Activity and Intent.

The API for GMA front calls is based on the following Java interfaces:
• com.fourjs.gma.extension.v1.IFunctionCallController
• com.fourjs.gma.extension.v1.IFunctionCall

The front call function controller (IFunctionCallController) is implemented by the GMA, it is used to notify function call results, raise runtime exceptions and invoke activities.

The front call function body (IFunctionCall) implements the actual custom front call code.
The steps to implement an IFunctionCall class are:

1. Create a Java source file with the name of the front call function, for example: "getPhoneId.java", that implements the IFunctionCall interface.
2. Define the Java package name identifying the front call module, for example: "package com.mycompany.utilities;".
3. Define a private IFunctionCallController object reference to handle the function controller.
4. Implement the setFunctionCallController() method for the function controller registration.
5. Implement the invoke() method to perform the actual front call task. In this method, use the controller's returnValues() method to return values from the front call. If needed, you can raise runtime errors with controller's raiseError() method. It is also possible to start an Android™ Activity with the startActivity* controller methods.
6. If an activity is started with controller's startActivityForResult method, implement the onActivityResult() method in the function body class, to handle the end of the activity, and call controller's returnValues() method to return values from the front call.
7. If needed, implement the onSaveInstanceState() and the onRestoreInstanceState() methods, to respectively save and restore information when Android™ has to suspend the application.

Note: In any case, the IFunctionCall class must either call the controller's returnValues() or raiseError() methods to give the control back to the Genero program.
### The `com.fourjs.gma.extension.v1.IFunctionCall` interface

#### Table 481: Methods of the `com.fourjs.gma.extension.v1.IFunctionCall` interface

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void setFunctionCallController(IFunctionCallController controller)</code></td>
<td>This method binds the front call function controller object to the function body object. The <code>controller</code> parameter is the <code>IFunctionCallController</code> object to bind with the front call function body object.</td>
</tr>
<tr>
<td><code>abstract void invoke(Object[] args) throws IllegalArgumentException</code></td>
<td>This method performs the front call. It will be called when the front call is executed from the Genero program. The <code>args</code> parameter is a variable list of parameters passed to the front call. This corresponds to the third argument of <code>ui.Interface.frontCall</code> on page 583.</td>
</tr>
<tr>
<td><code>void onSaveInstanceState(Bundle state)</code></td>
<td>Saves the state of an ongoing function call when Android™ needs to suspend the application. The <code>state</code> parameter is the bundle to save the state to.</td>
</tr>
<tr>
<td><code>void onRestoreInstanceState(Bundle state)</code></td>
<td>Restores the state of an ongoing function call, when Android™ needs to restore the application. The <code>state</code> parameter is the bundle to restore the state from.</td>
</tr>
<tr>
<td><code>void onActivityResult(int resultCode, Intent data)</code></td>
<td>Callback invoked when an activity started through <code>IFunctionCallController.startActivityForResult</code> finishes. The <code>resultCode</code> parameter is the integer result code returned by the child activity through its <code>setResult()</code> method. The <code>data</code> parameter is an Intent object, which can return result data to the caller (various data can be attached to Intent “extras”).</td>
</tr>
</tbody>
</table>
The `com.fourjs.gma.extension.v1.IFunctionCallController` interface

Table 482: Methods of the `com.fourjs.gma.extension.v1.IFunctionCallController` interface

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void returnValues(IFunctionCall functionCall, Object...values)</td>
<td>Notifies the controller that the front call function call has finished successfully. To be called typically at the end of the <code>IFunctionCall.invoke()</code> method.</td>
</tr>
<tr>
<td></td>
<td>The <code>functionCall</code> parameter is the current <code>IFunctionCall</code> object invoked.</td>
</tr>
<tr>
<td></td>
<td>The <code>values</code> parameter defines the variable list of front call function return values. This corresponds to the fourth parameter of <code>ui.Interface.frontCall</code> on page 583.</td>
</tr>
<tr>
<td>void raiseError(IFunctionCall functionCall, String message)</td>
<td>Notifies the controller of an error in the front call function call. This leads to a BDL runtime exception. To be called if needed within the <code>IFunctionCall.invoke()</code> method.</td>
</tr>
<tr>
<td></td>
<td>The <code>functionCall</code> parameter is the current <code>IFunctionCall</code> object invoked.</td>
</tr>
<tr>
<td></td>
<td>The <code>message</code> parameter holds the error message to be returned to the Genero program in the second part of the error -6333 message (see front call error handling in <code>ui.Interface.frontCall</code> on page 583).</td>
</tr>
<tr>
<td>void startActivity(IFunctionCall functionCall, Intent intent)</td>
<td>Starts a new activity. The function call won't be notified of the end of the activity. The Genero program will run in parallel of this activity. The behavior is similar to a RUN WITHOUT WAITING.</td>
</tr>
<tr>
<td></td>
<td>The <code>functionCall</code> parameter is the current <code>IFunctionCall</code> object invoked.</td>
</tr>
<tr>
<td></td>
<td>The <code>intent</code> parameter describes the activity to start.</td>
</tr>
<tr>
<td>void startActivityForResult(IFunctionCall functionCall, Intent intent)</td>
<td>Starts a new activity. The function call won't be notified of the end of the activity. The Genero program will remain blocked as long as the started activity isn't finished. The behavior is similar to a RUN.</td>
</tr>
<tr>
<td></td>
<td>The method <code>IFunctionCall.onActivityResult</code> will be called once the activity finishes.</td>
</tr>
<tr>
<td></td>
<td>The <code>functionCall</code> parameter is the current <code>IFunctionCall</code> object invoked.</td>
</tr>
<tr>
<td></td>
<td>The <code>intent</code> parameter describes the activity to start.</td>
</tr>
<tr>
<td>IClientHandler getClientHandler()</td>
<td>Returns an <code>IClientHandler</code> object, which is able to interact with Genero applications. A client handler can post action execution, cancel action future execution and check if an action execution is going to be executed. It can also define an application state change listener which triggers callbacks, when the app goes to pause, is resumed or is stopped.</td>
</tr>
<tr>
<td></td>
<td>For more details about the <code>IClientHandler</code> interface, see The <code>com.fourjs.gma.extension.v1.IClientHandler</code> interface on page 2258</td>
</tr>
<tr>
<td>Activity getCurrentActivity()</td>
<td>Returns the current <code>Activity</code> object. Provided in case if you need to pass the current activity to an Android™ API requiring this object.</td>
</tr>
<tr>
<td></td>
<td>Important: Don't use the returned activity to start other activities (don't call <code>Activity.startActivity</code> or <code>Activity.startActivityForResult</code>), use the helpers of the current interface instead.</td>
</tr>
</tbody>
</table>
The `com.fourjs.gma.extension.v1.IClientHandler` interface

Table 483: Methods of the `com.fourjs.gma.extension.v1.IClientHandler` interface

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void postDialogAction(String action)</code></td>
<td>Post a dialog action to be triggered as soon as a BDL dialog is active and the given action name fits one of the active actions of the current dialog. The <code>action</code> parameter defines the name of the dialog action.</td>
</tr>
<tr>
<td><code>boolean isDialogActionPending(String action)</code></td>
<td>Checks if a given action is still pending (the action was posted using <code>postDialogAction()</code>, but the action was still not activated by the Genero application and thus was not sent to the runtime). The <code>action</code> parameter defines the name of the dialog action.</td>
</tr>
</tbody>
</table>

Calling the custom front call from BDL

In the Genero program, use the `ui.Interface.frontCall()` API to call the front-end function. This method takes the front call module name as first parameter and the front call function name as second parameter. The front call module name is defined by the Java package name of the custom class implementing the `IFunctionCall` interface, and the front call function name is defined by the name of the class.

For example, if you implement the following front call function:

```java
package com.mycompany.utilities;
...
public class GetPhoneId implements IFunctionCall {

...}
```

The Genero program code must pass the Java package name "com.mycompany.utilities" as front call module name and the class name "GetPhoneId" as front call function name:

```java
CALL ui.Interface.frontCall("com.mycompany.utilities", "GetPhoneId", ["John DOE"], [msg])
```

Deploying the custom front call

The compiled Java classes implementing the front calls must be included in the mobile application Android™ `.apk`, which is created in the Genero Studio deployment procedure. The same GMA package building rules apply for front calls and for simple Java extensions. See Packaging custom Java extensions for GMA on page 2223 for more details.

Example

The example implements a HelloWorld call as a front call module.

```java
package com.mycompany.testmodule;

import android.content.Intent;
import android.os.Bundle;
```
import com.fourjs.gma.extension.v1.IFunctionCall;
import com.fourjs.gma.extension.v1.IFunctionCallController;

public class HelloWorld implements IFunctionCall {
    private IFunctionCallController mController;

    @Override
    public void setFunctionCallController(IFunctionCallController controller) {
        mController = controller;
    }

    @Override
    public void invoke(Object[] args) throws IllegalArgumentException {
        if (args.length != 1) {
            throw new IllegalArgumentException("HelloWorld takes one argument");
        }
        mController.returnValues(this, "Hello " + args[0].toString());
    }

    @Override
    public void onSaveInstanceState(Bundle state) {
    }

    @Override
    public void onRestoreInstanceState(Bundle state) {
    }

    @Override
    public void onActivityResult(int returnCode, Intent data) {
    }
}

In order to invoke the HelloWorld front-end function, use the ui.Interface.frontCall() API in the Genero program:

MAIN
DEFINE msg STRING
MENU
    ON ACTION frontcall ATTRIBUTES(TEXT="Call custom front call")
        CALL ui.Interface.frontCall("com.mycompany.testmodule", "HelloWorld", ["John DOE"], [msg])
    ON ACTION quit
        EXIT MENU
END MENU
END MAIN

Related concepts
Executing Java code with GMA on page 2219

Implement front call modules for GMI
Custom front call modules for the iOS front-end are implemented by using the API for GMI front calls in Objective-C.

GMI custom front call basics
In order to extend the GMI with your own front calls, you must be familiar with Objective-C programming, and if you want to interface with iOS Apps, have a knowledge of the iOS API.
**Important:** Before starting with GMI front call implementation, you need to get the GMI package and unzip the archive into the FGLDIR directory, as described in the prerequisites sections of *Building iOS apps with Genero* on page 3606.

The API for GMI front calls is based on the `FrontCall` class and the `FrontCallHelper` and `FunctionCall` protocols. You can find these in the file `frontcall.h` in the `$FGLDIR/include/gmi` directory.

To implement custom front calls, write a class which extends `FrontCall` and implements the “moduleName” and “execute:retCount:params” methods as well as the “initWithFunctionModuleHelper:” initializer.

Follow these steps to implement a custom front call module for the GMI:

1. Import the `frontcall.h` header file in your source.
2. Define an interface (MyFrontCall) which extends `FrontCall`.
3. Create the class (MyFrontCall) which implements this interface:
   
   a. Implement the `-(instancetype)initWithFunctionModuleHelper:(id)aHelper` initializer, calling `[super initWithFunctionModuleHelper:aHelper]` to pass the `FrontCallHelper` to the base implementation.
   
   b. Implement the `-(NSString*)moduleName` method, returning the name of the front call module.
   
   c. Implement the `-(void)execute:(NSString)name retCount:(int)retCount params:(NSArray)params` method, defining the body of your front calls. See below for details about the `execute` method.

**API to implement custom front calls in GMI**

To get parameters passed from the Genero program to the front call, and return values from the front call to the Genero program, use the following macros and methods of the `FrontCall` class:
### Table 484: GMI custom front call API

<table>
<thead>
<tr>
<th>Macro / Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(void) FC_REQUIRED_PARAMS(count)</td>
<td>Checks that the number of parameters passed by the Genero program equals <code>count</code>. This macro will raise an error in the Genero program if not enough parameters were passed.</td>
</tr>
<tr>
<td>(NSString *) FC_PARAM(index)</td>
<td>Get the string parameter passed to the front call, at the given position. If the parameters are of a different type, use the <code>doubleValue</code>, <code>floatValue</code> and <code>integerValue</code> methods on <code>NSString</code> or a <code>NSScanner</code>, to convert the parameter to the expected type.</td>
</tr>
<tr>
<td>(int) FC_PARAM_INT(index)</td>
<td>Get the int parameter passed to the front call, at the given position.</td>
</tr>
<tr>
<td>(void) intResult:(int) intValue</td>
<td>Ends the front call by returning one integer to Genero.</td>
</tr>
<tr>
<td>(void) doubleResult:(double) doubleValue</td>
<td>Ends the front call by returning one double to Genero.</td>
</tr>
<tr>
<td>(void) stringResult:(NSString <em>) :stringValue</em></td>
<td>Ends the front call by returning one string to Genero.</td>
</tr>
<tr>
<td>(void) startResult</td>
<td>Initiate setting multiple result values.</td>
</tr>
<tr>
<td></td>
<td>Must be followed by <code>add*</code> function calls and ended with <code>endResult</code>.</td>
</tr>
<tr>
<td>(void) addIntResult:(int) intValue</td>
<td>Add an integer to the list of results returned.</td>
</tr>
<tr>
<td></td>
<td>To be used after a <code>startResult</code> call.</td>
</tr>
<tr>
<td>(void) addDoubleResult:(double) doubleValue</td>
<td>Add a double to the list of results returned.</td>
</tr>
<tr>
<td></td>
<td>To be used after a <code>startResult</code> call.</td>
</tr>
<tr>
<td>(void) addStringResult:(NSString <em>) :stringValue</em></td>
<td>Add a string to the list of results returned.</td>
</tr>
<tr>
<td></td>
<td>To be used after a <code>startResult</code> call.</td>
</tr>
<tr>
<td>(void) endResult</td>
<td>Finalize the setting of multiple result values and return the results to the Genero program, with front call error code zero (indicating success).</td>
</tr>
<tr>
<td>(void) ok</td>
<td>Ends the front call without returning any value to Genero, indicating that the front call execution was successful.</td>
</tr>
<tr>
<td>(void) error(FCErrorCode):error</td>
<td>Ends the front call with a specific front call error code defined in <code>FCErrorCode enum</code> in <code>frontcall.h</code>, to indicate that front call execution failed, typically because of invalid parameters or invalid function name.</td>
</tr>
<tr>
<td>(void) errorWithMessage:(NSString <em>) message</em></td>
<td>Ends the front call with front call return code -4 (maps to BDL error -6333), and a user-defined error message, that can be read with <code>ERR_GET()</code> in the Genero program.</td>
</tr>
</tbody>
</table>
Calling the custom front call from BDL

In the Genero program, use the `ui.Interface.frontCall()` API to call the front-end function. This method takes the front call module name as first parameter and the front call function name as second parameter.

The front call module name is defined by the string value returned from the `-(NSString *) moduleName` method of your front call implementation, and the front call function name is passed to the `execute` method you implemented as first parameter (name).

For example, if you implement the following class:

```objc
#import <gmi/frontcall.h>
...
@interface MyFrontCall : FrontCall
...
@end
@class MyFrontCall
-(instancetype) initWithFunctionModuleHelper:(id)aHelper
{
    if (self = [super initWithFunctionModuleHelper:aHelper]) {
        ...
        return self;
    }
-(NSString*) moduleName{
    return @"MyModule";
}
-(void)execute:(NSString)name
retCount:(int)retCount
params:(NSArray)params
{
    [super execute:name retCount:retCount params:params];
    if ([[name lowercaseString] isEqualToString:@"myfrontcall"]) {
        ...
}
```

The Genero program code must pass the module name "MyModule" as front call module name and the class name "MyFrontCall" as front call function name:

```objc
CALL ui.Interface.frontCall("MyModule", "MyFrontCall", ["John DOE"],[msg])
```

Custom front call implementation details (execute method)

First of all, call the `execute` method of the parent `FrontCall` class, right at the top of the `execute` method:

```objc
[super execute:name retCount:retCount params:params];
```

The `execute` method must check the name of the front call function passed as parameter, to perform the expected code. This is the function name passed to the `ui.Interface.frontCall()` call in the Genero program:

```objc
if([[name lowercaseString] isEqualToString:@"myfunction"]){
```

Implement the body of the front call function in the `if` block as follows:

Add an `assert()` line, to make sure that the number of return values match:

```objc
assert(retCount == 2);
```
In order to get the parameters passed from the Genero program, use the FC_* macros in the body of your front call function.

First, check that the number of parameters passed is correct, with the FC_REQUIRED_PARAMS(count) macro:

```
FC_REQUIRED_PARAMS(3);
...
```

Retrieve the parameters passed to the front call with the FC_PARAM(index) or FC_PARAM_INT(index) macros, which return a NSString* and an int respectively. If needed, use the doubleValue, floatValue and integerValue methods on NSString or a NSScanner, to convert the parameter to the expected type:

```
NSString * info = FC_PARAM(0);
int v1 = [FC_PARAM(1) integerValue];
double v2 = [FC_PARAM(2) doubleValue];
```

Implement the actual code of the front call.

To return values to Genero, use one of the helper methods such as intResult:value, if a single value must be returned to the Genero program. If more than one value must be returned, build a return set with the startResult, add*Result and endResult methods:

```
[self startResult];
[self addIntResult:isIpad];
[self addIntResult:canLocate];
[self endResult];
```

If the front call displays a UI (for example, an UIAlertController or displays a customer UIViewController), call the willSetResultLater method of the FrontCall class, to avoid having the control flow returned to the Genero program upon exit of the execute method:

```
[self willSetResultLater];
```

Additionally, if you call the willSetResultLater method, you need to call one of the result methods like stringResult at a later time.

**Deploying the custom front call**

The compiled Objective-C classes must be included in the iOS app build process.

The same app building rules apply for custom front calls as for C extensions.

See **Building iOS apps with Genero** on page 3606 for more details.

**Example**

In this example, the ExtensionFrontCall class implements two front calls: "isipad" and "logindialog".

We start by defining the interface for the custom front call module:

```
@interface ExtensionFrontCall : FrontCall<UIAlertViewDelegate>
@end
```

The ExtensionFrontCall class extends FrontCall, and implements the UIAlertViewDelegate protocol which is used by the "logindialog" front call.

Next, we start the implementation of the interface:

```
@implementation ExtensionFrontCall
-(instancetype)initWithFunctionModuleHelper:(id)aHelper
```
We use the standard initializer which will be called by GMI on start-up and define "ExtensionFrontCall" as module name by returning it from the moduleName method.

We also start the implementation of the execute method by calling the super method.

Front call modules are compiled with the help of Makefile-gmi to a static IOS library and can be tested instantly on the command line with the help of Makefile-gmi. The static library can be either passed as an command line argument to gmibuildtool or placed in the gmi sub directory of a project directory. In both cases GMI detects the extension by enumerating all front call descendant classes upon start-up.

Note: When using more than one extension module, take care that class names and object/library file names are distinct.

The isipad front call example

This front call simply returns the information on which device GMI is running. If it is an iPad, the integer 1 will be returned to the Genero program:

```objective-c
if ([[[name lowercaseString] isEqualToString:@"isipad"]]) {
    assert(retCount == 1);
    BOOL isIpad = UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad;
    [self startResult];
    [self addIntResult:isIpad];
    [self endResult];
}
```

After checking that only one return parameter was defined in Genero, the code identifies the platform with the UI_USER_INTERFACE_IDIOM() API and stores the result in the isIpad variable.

The next three lines return the result value to Genero, by starting a result block with startResult, adding an int to the return set with addIntResult, and finally calling endResult to send the result to the Genero program.

The same behavior can be achieved with a single line: [self intResult:isIpad];, since we only return one result value.

The Genero program calls the isIPad front call as follows:

```objective-c
DEFINE res INTEGER
CALL ui.Interface.frontCall( "ExtensionFrontCall", "isipad", [], [res] )
```
| Extending the language | 2265

The logindialog front call example
This front call displays a log-in dialog to the user. It expects two parameters (the title and the message for the log-in
dialog), and returns the log-in name and the password entered by the end user:
if([[name lowercaseString] isEqualToString:@“logindialog”]) {
assert(retCount == 2);
FC_REQUIRED_PARAMS(2);
NSString *title = FC_PARAM(0);
NSString *message = FC_PARAM(1);
UIAlertView *alert = [[UIAlertView alloc]
initWithTitle:title
message:message
delegate:self
cancelButtonTitle:NSLocalizedString(@"Cancel",@"Cancel")
otherButtonTitles:NSLocalizedString(@"OK",@"OK"),nil];
alert.alertViewStyle = UIAlertViewStyleLoginAndPasswordInput;
[alert show];
[self willSetResultLater];
}
We first check that two result values were set in Genero and that two parameters were supplied to the front call.
Then we use the FC_PARAM macro to fetch the parameters and assign them to NSStrings.
Then we allocate and initialize an UIAlertView with the given message and title and set the alertViewStyle
to "UIAlertViewStyleLoginAndPasswordInput", so that one plain text field and one password field will
be displayed on the alert.
In the initWithTitle call we also set "self" as the delegate of the alert so that we receive callbacks after user
input (we had added the UIAlertViewDelegate protocol to our ExtensionFrontCall interface definition).
Finally, we call willSetResultLater, to keep the control flow in iOS. If we don’t call this function, GMI
concludes the front call was not handled by the execute function (as none of the xxxResult functions was called
inside), and the front call will fail with a “Frontcall not found” error message.
The ExtensionFrontCall class implements the alertView:didDismissWithButtonIndex: method
from the UIAlertViewDelegate protocol:
pragma mark UIAlertViewDelegate(void)
alertView:(UIAlertView *)alertViewdidDismissWithButtonIndex:
(NSInteger)buttonIndex {
[self startResult];
if (buttonIndex != alertView.cancelButtonIndex) {
[self addStringResult:[alertViewtextFieldAtIndex:0].text];
[self addStringResult:[alertViewtextFieldAtIndex:1].text];
} else {
[self addStringResult:nil];
[self addStringResult:nil];
}
[self endResult];
}
This method is called after the user has tapped on one of the buttons and the view has been dismissed. Inside this
method, we first call startResult to enable adding more than one return value.
If the tapped button was not the Cancel button, we add the values of the log-in and password fields as strings to the
results and then call endResult to return the control flow to the Genero program.
The Genero program calls the log-in dialog front call as follows:
DEFINE ul, up STRING
CALL ui.Interface.frontCall( "ExtensionFrontCall", "logindialog",


Note: The file userextension.m of the GMI Extension project contains a complete example on how to write custom front calls.

Related concepts
Building iOS apps with Genero on page 3606
Genero provides a command-line tool to build applications for iOS devices.

Implement front call modules for GBC
Custom front call modules for the Genero Browser Client (GBC) front-end are implemented by using JavaScript.

GBC custom front call basics
Note: This topic is only an introduction to custom GBC front call implementation. For more details, see the Genero Browser Client User Guide.
In order to extend the GBC with your own front calls, you must be familiar with JavaScript programming concepts.
Important: Custom front call module and function names must be registered in lowercase for the GBC front-end.
With GBC, front-end calls are JavaScript functions executed locally on the workstation where the browser is running.
Note: Executing front calls in the context of a web browser is limited to the OS functions a web browser can do. For example, it will not be possible to delete a file on the computer where the browser executes.

Customizing the GBC front-end
In order to integrate your custom front calls in the GBC front-end, you need to set up the GBC customization environment.
Basically, you will have to:

1. Setup GBC customization (install Node.js).
2. Extract the GBC front-end archive into a project-dir directory,
3. Copy your custom front calls JavaScript modules in the project-dir/customization,
4. Rebuild the GBC front-end with the grunt utility.
5. Configure the GAS to use the customized GBC front-end.
For more details, see the Genero Browser Client User Guide.

Structure of a custom front call JavaScript module
One JavaScript module will define a front call module implementing several front call functions.
The .js file must be copied into the project-dir/customization directory.
A custom front call JavaScript module must have the following structure:

```
"use strict";

module('FrontCallService.modules.module-name', ['FrontCallService'],
/**
 * @param {gbc} context
 * @param {classes} cls
 */
function(context, cls) {
    context.FrontCallService.modules.module-name = {
```
function-name: function (param1, ...) {
    ... user code ...
    return [ values ... ]
},

/* More functions can be defined for this module */

Where:
1. module-name is the name of the front call module, and corresponds to the first parameter of `ui.Interface.frontCall()`.
2. function-name is the name of the front call function, and corresponds to the second parameter of `ui.Interface.frontCall()`.
3. param1, param2 ... are the input values provided as third parameter of `ui.Interface.frontCall()`.
4. values is a JavaScript array containing the values to be returned in the last parameter of `ui.Interface.frontCall()`.

GBC custom front call API

The following JavaScript functions are provided to implement your custom front calls:

Table 485: GBC custom front call API

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>this.parametersError(\message\)</code></td>
<td>This function can be invoked when an invalid number of parameters is passed to the front call, in order to raise on exception in the BDL program. The message parameter holds the error message to be returned to the Genero program in the second part of the error -6333 message (see front call error handling in <code>ui.Interface.frontCall</code> on page 583).</td>
</tr>
<tr>
<td><code>this.runtimeError(\message\)</code></td>
<td>This function can be used to raise an exception in the BDL program, when the front call needs to warn the program that an error occurred. The message parameter holds the error message to be returned to the Genero program in the second part of the error -6333 message (see front call error handling in <code>ui.Interface.frontCall</code> on page 583).</td>
</tr>
</tbody>
</table>

Example

The JavaScript code in this example implements a GBC custom front call function "myfunc" for the module "mymodule":

```
"use strict";

modulum('FrontCallService.modules.mymodule', ['FrontCallService'],
    /**
```
```javascript
* @param {gbc} context
* @param {classes} cls
*/
function(context, cls) {
    context.FrontCallService.modules.mymodule = {
        myfunc: function (name) {
            if (name === undefined) {
                this.parametersError();
                return;
            }
            if (name.length === 0) {
                this.runtimeError("name shouldn't be empty");
                return;
            }
            return ["Hello, " + name + " !"];
        }
    };
};
```

From the Genero BDL program:

```gresql
DEFINE res STRING
CALL ui.Interface.frontcall("mymodule","myfunc", ["world"], [res])
DISPLAY res
```

**Web Components**

Implement specialized form elements with Web Components.

For more details, see [Web components](#) on page 1922.

**Library reference**

Reference for classes and functions provided as built-in or extension packages.

**Built-in functions**

A *built-in function* is a predefined function that is part of the runtime system, or provided as a library function automatically loaded when a program starts. The built-in functions are part of the language.

Note that some [operators](#) such as `FIELD_TOUCHED(field-spec)` look like functions, but these are core language operators that are different in terms of semantics and order of precedence.

**Related concepts**

[Utility modules](#) on page 2301
A utility function is a function provided in a separate library; it is not built in the runtime system.
# Built-in functions

## Table 486: Built-in functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arg_val( index INTEGER ) RETURNS STRING</td>
<td>Returns a command line argument by position.</td>
</tr>
<tr>
<td>arr_count() RETURNS INTEGER</td>
<td>Returns the number of rows entered during an INPUT ARRAY statement.</td>
</tr>
<tr>
<td>arr_curr() RETURNS INTEGER</td>
<td>Returns the current row in a DISPLAY ARRAY or INPUT ARRAY.</td>
</tr>
<tr>
<td>downshift( s STRING ) RETURNS STRING</td>
<td>Converts a string to lowercase.</td>
</tr>
<tr>
<td>err_get( messageId INTEGER ) RETURNS STRING</td>
<td>Returns the text corresponding to an error number.</td>
</tr>
<tr>
<td>err_print( messageId INTEGER )</td>
<td>Prints in the error line the text corresponding to an error number.</td>
</tr>
<tr>
<td>err_quit( messageId INTEGER )</td>
<td>Prints in the error line the text corresponding to an error number and terminates the program.</td>
</tr>
<tr>
<td>errorlog( message STRING )</td>
<td>Copies the string passed as parameter into the error log file.</td>
</tr>
<tr>
<td>fgl_buffertouched() RETURNS INTEGER</td>
<td>Returns TRUE if the input buffer was modified in the current field.</td>
</tr>
<tr>
<td>fgl_db_driver_type() RETURNS CHAR(3)</td>
<td>Returns the 3-letter identifier/code of the current database driver.</td>
</tr>
<tr>
<td>fgl_decimal_truncate( x DECIMAL, scale INTEGER ) RETURNS DECIMAL</td>
<td>Returns a decimal truncated to the precision passed as parameter.</td>
</tr>
<tr>
<td>fgl_decimal_sqrt( x DECIMAL ) RETURNS DECIMAL</td>
<td>Computes the square root of the decimal passed as parameter.</td>
</tr>
<tr>
<td>fgl_decimal_exp( x DECIMAL ) RETURNS DECIMAL</td>
<td>Returns the value of Euler's constant (e) raised to the power of the decimal passed as parameter.</td>
</tr>
<tr>
<td>fgl_decimal_power( x DECIMAL ) RETURNS DECIMAL</td>
<td>Raises decimal to the power of the real exponent.</td>
</tr>
<tr>
<td>fgl_decimal_log( x DECIMAL ) RETURNS DECIMAL</td>
<td>Returns the natural logarithm of the decimal passed as parameter.</td>
</tr>
<tr>
<td>fgl_decimal_log10( x DECIMAL ) RETURNS DECIMAL</td>
<td>Returns the base-10 logarithm of the decimal passed as parameter.</td>
</tr>
<tr>
<td>fgl_decimal_round( x DECIMAL ) RETURNS DECIMAL</td>
<td>Rounds the decimal to the nearest integer.</td>
</tr>
<tr>
<td>fgl_decimal_abs( x DECIMAL ) RETURNS DECIMAL</td>
<td>Returns the absolute value of the decimal passed as parameter.</td>
</tr>
<tr>
<td>fgl_decimal_mod( x DECIMAL, y DECIMAL ) RETURNS DECIMAL</td>
<td>Returns the remainder of the division.</td>
</tr>
<tr>
<td>fgl_decimal_add( x DECIMAL, y DECIMAL ) RETURNS DECIMAL</td>
<td>Returns the sum of the two decimals.</td>
</tr>
<tr>
<td>fgl_decimal_subtract( x DECIMAL, y DECIMAL ) RETURNS DECIMAL</td>
<td>Returns the difference of the two decimals.</td>
</tr>
<tr>
<td>fgl_decimal_multiply( x DECIMAL, y DECIMAL ) RETURNS DECIMAL</td>
<td>Returns the product of the two decimals.</td>
</tr>
<tr>
<td>fgl_decimal_divide( x DECIMAL, y DECIMAL ) RETURNS DECIMAL</td>
<td>Returns the quotient of the two decimals.</td>
</tr>
<tr>
<td>fgl_decimal_max( x DECIMAL, y DECIMAL ) RETURNS DECIMAL</td>
<td>Returns the maximum of the two decimals.</td>
</tr>
<tr>
<td>fgl_decimal_min( x DECIMAL, y DECIMAL ) RETURNS DECIMAL</td>
<td>Returns the minimum of the two decimals.</td>
</tr>
<tr>
<td>fgl_decimal_compare( x DECIMAL, y DECIMAL ) RETURNS INTEGER</td>
<td>Compares two decimals.</td>
</tr>
<tr>
<td>fgl_decimal_compare_less( x DECIMAL, y DECIMAL ) RETURNS INTEGER</td>
<td>Compares if the first decimal is less than the second.</td>
</tr>
<tr>
<td>fgl_decimal_compare_greater( x DECIMAL, y DECIMAL ) RETURNS INTEGER</td>
<td>Compares if the first decimal is greater than the second.</td>
</tr>
<tr>
<td>fgl_decimal_compare_equal( x DECIMAL, y DECIMAL ) RETURNS INTEGER</td>
<td>Compares if the two decimals are equal.</td>
</tr>
<tr>
<td>fgl_decimal_compare_less_or_equal( x DECIMAL, y DECIMAL ) RETURNS INTEGER</td>
<td>Compares if the first decimal is less than or equal to the second.</td>
</tr>
<tr>
<td>fgl_decimal_compare_greater_or_equal( x DECIMAL, y DECIMAL ) RETURNS INTEGER</td>
<td>Compares if the first decimal is greater than or equal to the second.</td>
</tr>
<tr>
<td>fgl_decimal_compare_not_equal( x DECIMAL, y DECIMAL ) RETURNS INTEGER</td>
<td>Compares if the two decimals are not equal.</td>
</tr>
<tr>
<td>fgl_decimal_compare_not_less( x DECIMAL, y DECIMAL ) RETURNS INTEGER</td>
<td>Compares if the first decimal is not less than the second.</td>
</tr>
<tr>
<td>fgl_decimal_compare_not_greater( x DECIMAL, y DECIMAL ) RETURNS INTEGER</td>
<td>Compares if the first decimal is not greater than the second.</td>
</tr>
<tr>
<td>fgl_decimal_compare_not_greater_or_equal( x DECIMAL, y DECIMAL ) RETURNS INTEGER</td>
<td>Compares if the first decimal is not greater than or equal to the second.</td>
</tr>
<tr>
<td>fgl_decimal_compare_not_less_or_equal( x DECIMAL, y DECIMAL ) RETURNS INTEGER</td>
<td>Compares if the first decimal is not less than or equal to the second.</td>
</tr>
<tr>
<td>fgl_decimal_compare_not_equal_or_less( x DECIMAL, y DECIMAL ) RETURNS INTEGER</td>
<td>Compares if the first decimal is not equal and not less than the second.</td>
</tr>
<tr>
<td>fgl_decimal_compare_not_equal_or_greater( x DECIMAL, y DECIMAL ) RETURNS INTEGER</td>
<td>Compares if the first decimal is not equal and not greater than the second.</td>
</tr>
<tr>
<td>fgl_decimal_compare_not_equal_or_less_or_equal( x DECIMAL, y DECIMAL ) RETURNS INTEGER</td>
<td>Compares if the first decimal is not equal and not less than or equal to the second.</td>
</tr>
<tr>
<td>fgl_decimal_compare_not_equal_or_greater_or_equal( x DECIMAL, y DECIMAL ) RETURNS INTEGER</td>
<td>Compares if the first decimal is not equal and not greater than or equal to the second.</td>
</tr>
<tr>
<td>fgl_decimal_compare_equal_or_not_equal( x DECIMAL, y DECIMAL ) RETURNS INTEGER</td>
<td>Compares if the two decimals are equal or not equal.</td>
</tr>
<tr>
<td>fgl_decimal_compare_less_or_not_greater( x DECIMAL, y DECIMAL ) RETURNS INTEGER</td>
<td>Compares if the first decimal is less than or not greater than the second.</td>
</tr>
<tr>
<td>fgl_decimal_compare_less_or_not_greater_or_equal( x DECIMAL, y DECIMAL ) RETURNS INTEGER</td>
<td>Compares if the first decimal is less than or not greater than or equal to the second.</td>
</tr>
<tr>
<td>fgl_decimal_compare_greater_or_not_less( x DECIMAL, y DECIMAL ) RETURNS INTEGER</td>
<td>Compares if the first decimal is greater than or not less than the second.</td>
</tr>
<tr>
<td>fgl_decimal_compare_greater_or_not_less_or_equal( x DECIMAL, y DECIMAL ) RETURNS INTEGER</td>
<td>Compares if the first decimal is greater than or not less than or equal to the second.</td>
</tr>
<tr>
<td>fgl_decimal_compare_equal_or_not_less( x DECIMAL, y DECIMAL ) RETURNS INTEGER</td>
<td>Compares if the two decimals are equal or not less than the first decimal.</td>
</tr>
<tr>
<td>fgl_decimal_compare_equal_or_not_greater( x DECIMAL, y DECIMAL ) RETURNS INTEGER</td>
<td>Compares if the two decimals are equal or not greater than the first decimal.</td>
</tr>
<tr>
<td>fgl_decimal_compare_equal_or_not_less_or_greater( x DECIMAL, y DECIMAL ) RETURNS INTEGER</td>
<td>Compares if the two decimals are equal or neither less nor greater than the first decimal.</td>
</tr>
</tbody>
</table>
**arg_val()**
Returns a command line argument by position.

**Syntax**
```
arg_val(
    index INTEGER )
RETURNS STRING
```

1. *index* is an integer defining the argument position.

**Usage**
This function provides a mechanism for passing values to the program through the command line that invokes the program. You can design a program to expect or allow arguments after the name of the program in the command line.

The *index* parameter defines the argument to be returned. 0 returns the name of the program, 1 returns the first argument.

Like all built-in functions, `arg_val()` can be invoked from any program block. You can use it to pass values to `MAIN`, which cannot have formal arguments, but you are not restricted to calling `arg_val()` from the `MAIN` statement.

Use the `arg_val()` function to retrieve individual arguments during program execution. Use the `num_args()` function to determine how many arguments follow the program name on the command line.

If *index* is greater than 0, `arg_val(index)` returns the command-line argument used at a given position. The value of *index* must be between 0 and the value returned by `num_args()`, the number of command-line arguments. The expression `arg_val(0)` returns the name of the application program.

If the argument is negative or greater than `num_args()`, the method returns `NULL`.

**Related concepts**
- `base.Application.getArgument` on page 2407
  Returns the command line argument by position.

**arr_count()**
Returns the number of rows entered during an `INPUT ARRAY` statement.

**Syntax**
```
arr_count()
RETURNS INTEGER
```

**Usage**
Use `arr_count()` to determine the number of program records that are currently stored in a static program array used by the `INPUT ARRAY` instruction.

This function is typically called inside or after `INPUT ARRAY` or `DISPLAY ARRAY` statement.

`arr_count()` returns a positive integer, corresponding to the index of the furthest record within the static program array that the user accessed. Not all the rows counted by `arr_count()` necessarily contain data (for example, if the user presses the Down key more times than there are rows of data.

This function is not required when using dynamic arrays. In such case, the total number of rows in defined by the `array.getLength()` method after the dialog, or by the `ui.Dialog.getArrayLength()` method during the dialog execution.

**Related concepts**
- `Editable record list (INPUT ARRAY)` on page 1531
The **INPUT ARRAY** instruction provides always-editable record list handling in an application form.

**Record list (DISPLAY ARRAY)** on page 1501

The **DISPLAY ARRAY** instruction provides record list navigation in an application form, with optional record modification actions.

**arr_curr()** on page 2272

Returns the current row in a **DISPLAY ARRAY** or **INPUT ARRAY**.

**arr_curr()**

Returns the current row in a **DISPLAY ARRAY** or **INPUT ARRAY**.

**Syntax**

```
arr_curr()
  RETURNS INTEGER
```

**Usage**

The `arr_curr()` function returns an integer value that identifies the current row of a list of rows in an **INPUT ARRAY** or **DISPLAY ARRAY** instruction.

The first row is numbered 1.

**Important:** The row index returned by the `arr_curr()` function is constant in the context of a dialog block, even when removing rows from the array by program.

The `arr_curr()` and `scr_line()` functions can return different values if the program array is larger than the screen array.

The `arr_curr()` can be used to get the index of the last current row, after the execution of the dialog.

In multiple dialogs implementing several list controllers, consider using the `ui.Dialog.getCurrentRow()` method instead of `arr_curr()` to get the current row of a specific list identified by its screen array.

**Related concepts**

[Handling the current row](#) on page 1835

Query and control the current row in a read-only or editable list of records.

**downshift()**

Converts a string to lowercase.

**Syntax**

```
downshift(  
  s STRING  
)  
  RETURNS STRING
```

1. `s` is the character string to convert to lowercase letters.

**Usage**

The `downshift()` function returns a string value in which all uppercase characters in its argument are converted to lowercase.

The character conversion depends on locale settings (the LC_CTYPE environment variable). Non-alphabetic or lowercase characters are not altered.

**Related concepts**

[upshift()](#) on page 2299
Converts a string to uppercase.

**scr_line()**
Returns the index of the current row in the screen array.

**Syntax**
```
scr_line()
RETURNS INTEGER
```

**Usage**
The `scr_line()` function returns the index of the current row in the screen array. It is typically used inside a `DISPLAY ARRAY` or `INPUT ARRAY` statement.

**Important:** When using new graphical objects such as `TABLE` containers, this function can return an invalid screen array line number, because the current row may not be visible if the user scrolls in the list with scrollbars.

Do not confuse `scr_line()` with `arr_curr()`: the first returns the index of the current row in the form screen array, and the second returns the index of the current row in the program variable.

**Related concepts**
- Screen records / arrays on page 1251
- Form fields can be grouped in a `screen record` or `screen array` definition.

**num_args()**
Returns the number of program arguments.

**Syntax**
```
num_args()
RETURNS INTEGER
```

**Usage**
Returns the number of arguments passed to the program.
The function returns 0 if no arguments are passed to the program.

Use the `arg_val()` function to get a command line argument at a given position.

**Related concepts**
- `base.Application.getArgumentCount` on page 2408
  Returns the total number of command line arguments.

**err_get()**
Returns the text corresponding to an error number.

**Syntax**
```
err_get ( 
    messageId INTEGER 
) 
RETURNS STRING
```

1. `messageId` is a runtime error or an Informix® SQL error.
Usage

The `err_get()` function returns the error message corresponding to the number passed as parameter. This can be a Genero BDL error (like -6324), or an IBM® Informix® SQL error message.

**Important:** IBM® Informix® SQL message numbers can only be supported, if the program is connected to an Informix® database, and an SQL error occurred just before calling this function. Do not use this function in the context of SQL execution, when using different type of database servers.

Related reference

Genero BDL errors on page 3154
System error messages sorted by error number.

`err_print()`

Prints in the error line the text corresponding to an error number.

Syntax

```plaintext
err_print( messageId INTEGER )
```

1. `messageId` is a runtime error or an Informix® SQL error.

Usage

The `err_print()` function displays to the screen the error message corresponding to the number passed as parameter. The message will be displayed in the error line defined by the program.

**Important:** IBM® Informix® SQL message numbers can only be supported, if the program is connected to an Informix® database, and an SQL error occurred just before calling this function. Do not use this function in the context of SQL execution, when using different type of database servers.

Related reference

Genero BDL errors on page 3154
System error messages sorted by error number.

`err_quit()`

Prints in the error line the text corresponding to an error number and terminates the program.

Syntax

```plaintext
err_quit( messageId INTEGER )
```

1. `messageId` is a runtime error or an Informix® SQL error.

Usage

The `err_quit()` function prints the error message corresponding to the number passed as parameter. The message will be displayed in standard error stream and the program will terminate.

**Important:** IBM® Informix® SQL message numbers can only be supported, if the program is connected to an Informix® database, and an SQL error occurred just before calling this function. Do not use this function in the context of SQL execution, when using different type of database servers.

Related reference

Genero BDL errors on page 3154
System error messages sorted by error number.

**errorlog()**
Copies the string passed as parameter into the error log file.

**Syntax**

```pascal
errorlog(
    message STRING )
```

1. *message* is the character string to be inserted in the error log file.

**Usage**
The `errorlog()` function writes the passed string in the current error log file. The error log file is defined by a call to the `startlog()` function.

**Important:** Sensitive data may be written to the `startlog()` file. Make sure that the log output produced by `errorlog()` calls is written to files that can only be read by application administrators.

Use this function to identify errors in programs and to customize error handling. The error log functions can also be used to trace the way a program is used in order to improve it, record work habits, or help to detect attempts to breach security.

**Related concepts**

- `startlog()` on page 2298
  Initializes error logging and opens the error log file passed as the parameter.

**fgl_buffertouched()**
Returns `TRUE` if the input buffer was modified in the current field.

**Syntax**

```pascal
fgl_buffertouched()
RETURNS INTEGER
```

**Usage**
The function returns `TRUE` if the input buffer has been modified after the current field was selected (got the focus).

Call this function in `AFTER FIELD`, `AFTER INPUT`, `AFTER CONSTRUCT`, `ON KEY`, `ON ACTION` blocks.

This function is not equivalent to `FIELD_TOUCHED()`. The modification status of `fgl_buffertouched()` is reset when entering a new field, while `FIELD_TOUCHED()` returns `TRUE` when a field was modified during the interactive instruction.

**fgl_db_driver_type()**
Returns the 3-letter identifier/code of the current database driver.

**Syntax**

```pascal
fgl_db_driver_type()
RETURNS CHAR(3)
```

**Usage**
This function can be called after connecting to a database server with the `CONNECT` or `DATABASE` instructions, in order to identify the type of the target database with the driver type.
Returned value is a 3-letter driver code, in lower case, such as "ifx", "ora", "db2", etc.
See the drivers table for more details about the list of database driver types.
The function returns \texttt{NULL} if there is no current database driver (for example, if database connection is not yet established).

\textbf{Related concepts}
\texttt{Solution 1: Use database specific serial generators} on page 626

\texttt{fgl\_decimal\_truncate()}
Returns a decimal truncated to the precision passed as parameter.

\textbf{Syntax}

\begin{verbatim}
fgl_decimal_truncate (  
    x DECIMAL,  
    scale INTEGER )  
RETURNS DECIMAL
\end{verbatim}

1. \( x \) is the decimal to be converted.
2. \( scale \) defines the number of digits after the decimal point.

\textbf{Usage}
This function truncates the decimal to the number of decimal digits specified.
The value is not rounded, it is just truncated. For example, when truncating 12.345 to 2 decimal digits, the result will be 12.34, not 12.35.

\texttt{fgl\_decimal\_sqrt()}
Computes the square root of the decimal passed as parameter.

\textbf{Syntax}

\begin{verbatim}
fgl_decimal_sqrt (  
    x DECIMAL )  
RETURNS DECIMAL
\end{verbatim}

1. \( x \) is the decimal to be computed.

\texttt{fgl\_decimal\_exp()}
Returns the value of Euler's constant (e) raised to the power of the decimal passed as parameter.

\textbf{Syntax}

\begin{verbatim}
fgl_decimal_exp (  
    x DECIMAL )  
RETURNS DECIMAL
\end{verbatim}

1. \( x \) is the decimal to be computed.

\texttt{fgl\_decimal\_logn()}
Returns the natural logarithm of the decimal passed as parameter.

\textbf{Syntax}

\begin{verbatim}
fgl_decimal_logn (  
    x DECIMAL )
\end{verbatim}
**fgl_decimal_power()**

Raises decimal to the power of the real exponent.

**Syntax**

```
fgl_decimal_power(
    x DECIMAL,
    y DECIMAL
) RETURNS DECIMAL
```

1. `x` is the decimal to be raised to the power of `y`.
2. `y` is the exponent.

**Usage**

Unlike the ** operator, the `fgl_decimal_power()` function supports real numbers for the exponent.

**Related concepts**

- **Exponentiation (**) on page 352**

The ** operator calculates an exponentiation.

**fgl_dialog_getbuffer()**

Returns the text of the input buffer of the current field.

**Syntax**

```
fgl_dialog_getbuffer()
RETURNS STRING
```

**Usage**

The `fgl_dialog_getbuffer()` function returns the content of the input buffer of the current field. It must be used in INPUT, INPUT ARRAY and CONSTRUCT blocks.

The function is especially useful in a CONSTRUCT instruction, because there is no variable associated to fields in this case.

Consider using the `ui.Dialog.getFieldBuffer()` method instead.

**Related concepts**

- **fgl_dialog_setbuffer() on page 2277**
    Sets the input buffer of the current field.
- **GET_FLDBUF() on page 374**
    The GET_FLDBUF() operator returns as character strings the current values of the specified fields.

**fgl_dialog_setbuffer()**

Sets the input buffer of the current field.

**Syntax**

```
fgl_dialog_setbuffer(
    value STRING
)
```

1. `value` is the text to set in the current input buffer.
Usage
In the default buffered input mode, this function modifies the input buffer of the current field; the corresponding input variable is not assigned. It makes no sense to call this function in BEFORE FIELD blocks of INPUT and INPUT ARRAY. However, if the statement is using the UNBUFFERED mode, the function will set both the field buffer and the program variable. If the string set by the function does not represent a valid value that can be stored by the program variable, the buffer and the variable will be set to NULL.

The fgl_dialog_setbuffer() function must be used in INPUT, INPUT ARRAY and CONSTRUCT blocks.

This function sets the modification flag for both FIELD_TOUCHED() and fgl_buffertouched() functions. There is a slight difference between both functions. The modification flag for fgl_buffertouched() is reset to FALSE when entering the field.

The function is especially useful in a CONSTRUCT instruction, because there is no variable associated with fields in this case.

Related concepts
fgl_dialog_getbuffer() on page 2277
Returns the text of the input buffer of the current field.

The buffered and unbuffered modes on page 1720
The buffered and unbuffered mode control the synchronization of program variables and form fields.

fgl_dialog_getfieldname()
Returns the name of the current input field.

Syntax

```c
fgl_dialog_getfieldname()
RETURNS STRING
```

Usage
This function returns the name of the current input field during a dialog execution. It must be use in INPUT, INPUT ARRAY or CONSTRUCT blocks.

Only the column part of the field name is returned (screen record name is omitted).

The fgl_dialog_getfieldname() is similar to the INFIELD() operator and fgl_dialog_infield() function.

fgl_dialog_infield()
This function checks for the current input field.

Syntax

```c
fgl_dialog_infield(
    name STRING
) 
RETURNS INTEGER
```

1. name is the name if the form field.

Usage
The fgl_dialog_infield() function returns TRUE if the field name passed as the parameter is the current input field.

The function must be called in INPUT, INPUT ARRAY or CONSTRUCT blocks.
This function is the equivalent of the \texttt{INFIELD()} operator, except that the function takes a string expression as parameter, while the \texttt{INFIELD()} operator expects a hard-coded form field name.

\textbf{Related concepts}

\textit{Screen records / arrays} on page 1251
Form fields can be grouped in a \textit{screen record} or \textit{screen array} definition.

\texttt{fgl\_dialog\_setcursor()}
This function sets the position of the edit cursor in the current field.

\textbf{Syntax}

\begin{verbatim}
fgl_dialog_setcursor(
    x INTEGER
)
\end{verbatim}

1. \texttt{x} is the edit cursor position in the text.

\textbf{Usage}

The \texttt{fgl\_dialog\_setcursor()} function moves the edit cursor to the specified position in the current field. The function must be called in interactive instructions control blocks, when staying in the current field.

This function has only an effect when staying in the current field, it is not recommended to call it in an \texttt{AFTER FIELD} or \texttt{AFTER ROW} event for example.

Note that you can use \texttt{FGL\_DIALOG\_SETSELECTION()} to select a piece of text in a field.

\textbf{Important:} When using byte length semantics, the position is expressed in bytes. When using \textit{char length semantics}, the unit is characters. This matters when using a multibyte locale such as UTF-8.

\textbf{Related concepts}

\texttt{fgl\_dialog\_getcursor() / fgl\_getcursor()} on page 2281
Returns the position of the edit cursor in the current field.

\texttt{fgl\_dialog\_setselection()} on page 2282
Selects the text in the current field.

\texttt{fgl\_dialog\_setfieldorder()}
This function enables or disables field order constraint.

\textbf{Syntax}

\begin{verbatim}
fgl_dialog_setfieldorder(
    constrained INTEGER
)
\end{verbatim}

1. When \texttt{constrained} is \texttt{TRUE}, the field order is constrained. When \texttt{constrained} is \texttt{FALSE}, the field order is not constrained.

\textbf{Usage}

Typical applications control user input with \texttt{BEFORE FIELD} and \texttt{AFTER FIELD} blocks. In many cases the field order and the sequential execution of \texttt{AFTER FIELD} blocks is important in order to validate the data entered by the user. But with graphical front-ends you can use the mouse to move to a field. By default the runtime system executes all \texttt{BEFORE FIELD} and \texttt{AFTER FIELD} blocks of the fields used by the interactive instruction, from the origin field to the target field selected by mouse click. If needed, you can force the runtime system to ignore all intermediate field triggers, by calling this function with a \texttt{FALSE} attribute.

This function must be called outside interactive dialog blocks, typically at the beginning of the program.

Consider using the \texttt{Dialog\_fieldOrder} parameter when all programs are affected. The FGLPROFILE profile entry is the default when the \texttt{fgl\_dialog\_setfieldorder()} function is not used.
Consider using OPTIONS FIELD ORDER FORM for new developments with graphical rendering.

**fgl_dialog_setcurrline()**
This function moves to a specific row in a record list.

**Syntax**

```c
fgl_dialog_setcurrline(
    screenLine INTEGER,
    row INTEGER)
```

1. `screenLine` is the line number in the form screen array.
2. `row` is the row number in the program array variable.

**Usage**

Moves to the row / screen line specified. See `fgl_set_arr_curr()` for more details.

To be called during a DISPLAY ARRAY or INPUT ARRAY instruction, inside BEFORE DISPLAY / BEFORE INPUT or ON ACTION / ON KEY blocks only.

The `screenLine` parameter is ignored in GUI mode.

**Related concepts**

- **Screen records / arrays** on page 1251
- Form fields can be grouped in a screen record or screen array definition.

**fgl_dialog_getbufferstart()**

Returns the row offset of the page to feed a paged display array.

**Syntax**

```c
fgl_dialog_getbufferstart ()
RETURNS INTEGER
```

**Usage**

The `FGL_DIALOG_GETBUFFERSTART()` function returns the record list offset to be used to fill a page of a DISPLAY ARRAY running in paged mode.

This function must be called in the context of the ON FILL BUFFER trigger. The returned value is undefined if the function is used outside this trigger.

**fgl_dialog_getbufferlength()**

Returns the number of rows to feed a paged DISPLAY ARRAY.

**Syntax**

```c
fgl_dialog_getbufferlength ()
RETURNS INTEGER
```

**Usage**

The `fgl_dialog_getbufferlength()` function returns the number of rows to be provided by the program to fill a page of a DISPLAY ARRAY running in paged mode.

This function must be called in the context of the ON FILL BUFFER trigger. The returned value is undefined if the function is used outside this trigger.
fgl_dialog_getcursor() / fgl_getcursor()
Returns the position of the edit cursor in the current field.

Syntax

```plaintext
fgl_dialog_getcursor()
   RETURNS INTEGER
```

Usage

The fgl_dialog_getcursor() function can be used in conjunction with fgl_dialog_getselectionend() to get the position of the edit cursor and the piece of text that is selected in the current field.

**Important:** When using byte length semantics, the position is expressed in bytes. When using char length semantics, the unit is characters. This matters when using a multibyte locale such as UTF-8.

Related concepts

- `fgl_dialog_getselectionend()` on page 2281
  Returns the position of the last selected character in the current field.
- `fgl_dialog_setcursor()` on page 2279
  This function sets the position of the edit cursor in the current field.
- `fgl_dialog_setselection()` on page 2282
  Selects the text in the current field.

fgl_dialog_getkeylabel()
Returns the label associated to a key for the current interactive instruction.

Syntax

```plaintext
fgl_dialog_getkeylabel(  
   keyName STRING  
)  
   RETURNS STRING
```

1. `keyName` is the logical name of a key such as F11 or DELETE, INSERT, CANCEL.

Usage

The fgl_dialog_getkeylabel() function returns the label defined for the function or control key passed as parameter, for the current interactive instruction.

This function returns the key labels defined for the current dialog. There are different levels of key label definitions. This function is provided for backward compatibility, use action defaults to define action view texts.

Related concepts

- Setting action key labels on page 1775
  Labels can be defined to decorate buttons controlled by ON KEY / COMMAND KEY action handlers.
- Configuring actions on page 1744
  Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with action attributes.

fgl_dialog_getselectionend()
Returns the position of the last selected character in the current field.

Syntax

```plaintext
fgl_dialog_getselectionend()```
**Usage**

The `fgl_dialog_getselectionend()` function returns the edit cursor position of the last selected character in the text of the current field.

**Important:** When using byte length semantics, the position is expressed in bytes. When using char length semantics, the unit is characters. This matters when using a multibyte locale such as UTF-8.

The function returns zero if the complete text is selected.

The edit cursor position returned by `fgl_dialog_getcursor()` will be lower than the position returned by `fgl_dialog_getselectionend()` if the text has been selected backwards.

**Related concepts**

`fgl_dialog_getcursor() / fgl_getcursor()` on page 2281
Returns the position of the edit cursor in the current field.

`fgl_dialog_setselection()` on page 2282
Selects the text in the current field.

`fgl_dialog_setkeylabel()`
Sets the label associated to a key for the current interactive instruction.

**Syntax**

```c
fgl_dialog_setkeylabel(
    keyName STRING,
    text STRING )
```

1. `keyName` is the logical name of a key such as F11 or DELETE,INSERT, CANCEL.
2. `text` is the text associated to the key.

**Usage**

The `fgl_dialog_setkeylabel()` associates a text description to a function or control key for the current dialog. Default action views (buttons that appears in the action panel of a window) will get the label displayed instead of the function or control key name.

This function defines the key labels for the current dialog. There are different levels of key label definitions.

**Note:** This feature is supported for backward compatibility. Consider using action attributes to define accelerator keys and decorate actions.

**Related concepts**

Setting action key labels on page 1775
Labels can be defined to decorate buttons controlled by ON KEY / COMMAND KEY action handlers.

Configuring actions on page 1744
Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with action attributes.

`fgl_dialog_setselection()`
Selects the text in the current field.

**Syntax**

```c
fgl_dialog_setselection(
    start INTEGER,
    end INTEGER )
```
1. **start** defines the edit cursor position.
2. **end** defines the selection end position.

**Usage**

A call to `fgl_dialog_setselection(cursor, end)` sets the text selection in the current form field. The `start` parameter defines the character position of the edit cursor (equivalent to `fgl_dialog_getcursor()` position), while `end` defines the character position of the end of the text selection (equivalent to `fgl_dialog_getselectionend()` position).

**Important:** When using byte length semantics, the positions are expressed in bytes. When using char length semantics, the unit is characters. This matters when using a multibyte locale such as UTF-8.

`start` can be lower, greater or equal to `end`.

This function has only an effect when staying in the current field, it is not recommended to call it in an **AFTER FIELD** or **AFTER ROW** event for example.

**Related concepts**

- `fgl_dialog_getcursor()` / `fgl_getcursor()` on page 2281
  Returns the position of the edit cursor in the current field.
- `fgl_dialog_getselectionend()` on page 2281
  Returns the position of the last selected character in the current field.

**fgl_drawbox()**

Draws a rectangle in the current window.

**Syntax**

```c
fgl_drawbox(
    height INTEGER,
    width INTEGER,
    posY INTEGER,
    posX INTEGER,
    color INTEGER )
```

1. **height** is the height of the rectangle.
2. **width** is the width of the rectangle.
3. **posY** is the vertical coordinate (line) of the upper side of the rectangle.
4. **posX** is the horizontal coordinate (column) of the left side of the rectangle.
5. **color** is the color number (ignored).

**Usage**

The `fgl_drawbox()` function draws a rectangle based on the character terminal coordinates in the current open window.

Dimensions and coordinates are specified in grid cell units (characters).

This function is provided for backward compatibility. A call to this function will be ignored if the current window is not SCREEN based. The function is supported to draw rectangles in text mode applications.

**Related concepts**

- `fgl_drawline()` on page 2284
Draws a line in the current window (TUI and traditional mode).

**fgl_drawline()**
Draws a line in the current window (TUI and traditional mode).

**Syntax**

```c
fgl_drawline(
    posY INTEGER,
    posX INTEGER,
    width INTEGER,
    type CHAR(1),
    color INTEGER)
```

1. `posY` is the vertical coordinate (line) of the start of the line.
2. `posX` is the horizontal coordinate (column) of the start of the line.
3. `width` is the width of the line.
4. `type` (ignored).
5. `color` is the color number (ignored).

**Usage**
The `fgl_drawline()` function draws a line based on the character terminal coordinates in the current open window.

Dimensions and coordinates are specified in grid cell units (characters).

This function is provided for backward compatibility. A call to this function will be ignored if the current window is not SCREEN based. The function is supported to draw lines in text mode applications.

**Related concepts**
`fgl_drawbox()` on page 2283
Draws a rectangle in the current window.

`fgl_eventloop()`
Waits for a user interaction event.

**Syntax**

```c
fgl_eventloop()
RETURNS BOOLEAN
```

**Usage**
The `fgl_eventloop()` function is used to implement the parallel dialog main event loop, in conjunction with `START DIALOG / TERMINATE DIALOG` instructions, used to register and end parallel dialogs.

The function waits until a user interaction event occurs and returns TRUE or FALSE to indicate if the event loop must continue or stop. It is typically used in a WHILE / END WHILE loop:

```
WHILE fgl_eventLoop() END WHILE
```

Parallel dialogs are registered with the `START DIALOG` instruction. As long as at least one parallel dialog is registered, the `fgl_eventloop()` function returns TRUE. When the last parallel dialog is ended with a `TERMINATE DIALOG` instruction, the `fgl_eventloop()` function returns FALSE and the even loop is stopped.

**Related concepts**
`WHILE` on page 389
The WHILE statement executes a block of statements until the specified condition becomes false.

Understanding parallel dialogs on page 1700

Parallel dialogs allow the control of several forms simultaneously.

Parallel dialogs for split views on page 1902

In order to control the left-hand and right-hand split view content, you must implement two parallel dialogs, each dedicated to a pane.

fgl_getenv()

Returns the value of the environment variable.

Syntax

```c
fgl_getenv(
    name STRING )
RETURNS STRING
```

1. `name` is the name of the environment variable.

Usage

The argument of `fgl_getenv()` must be the name of an environment variable.

If the requested value exists in the current user environment, the function returns the value of that variable. If the specified environment variable is not defined, the function returns a `NULL` value. If the environment variable is defined but does not have a value assigned to it, the function returns blank spaces.

Related concepts

fgl_setenv() on page 2294

Sets the value of an environment variable.

fgl_gethelp()

Reads the current help file, returning help text based on the help identifier.

Syntax

```c
fgl_gethelp(
    id INTEGER )
RETURNS STRING
```

1. `id` is the help text identifier.

Usage

The `fgl_gethelp()` function returns the text corresponding to the help message number passed as parameter. The text is read from the current help file. The current help file is defined by the `OPTIONS HELP FILE` instruction.

fgl_getpid()

Returns the system process identifier.

Syntax

```c
fgl_getpid()
RETURNS INTEGER
```
**Usage**
The `fgl_getpid()` function returns the current process identifier. The process identifier is provided by the operating system.

**Related concepts**

- `fgl_system()` on page 2296
  Runs a command on the application server.

- `fgl_getfile()`
  Retrieves a file from the front-end context to the virtual machine context.

**Syntax**

```
fgl_getfile(
    remotePath STRING,
    localPath STRING)
```

1. `remotePath` is the path of the file to retrieve from the front-end context.
2. `localPath` is the path of the file to write in the virtual machine context.

**Usage**
The `fgl_getfile()` function uploads a file from the front-end workstation disk to the application server disk where `fglr` is executed.

**Important:** Using this function can result in a security hole if you allow the end user to specify the file paths without control. There is no limitation on the file content or file paths. If the user executing the application on the server side is allowed to write critical server files, the program could transfer files from the client workstation and overwrite critical server files. On the other hand, critical files can be read from the client workstation and copied on the application server. It is in the hands of the programmer to implement file path and/or file content restrictions in the programs using `fgl_getfile()`.

When the front-end is located on a mobile device (GMA or GMI), the `fgl_getfile()` function can take an opaque file path as first argument, to identify a local device resource returned from a front call such as `choosePhoto` on page 2699, `takeVideo` on page 2712. This allows you to retrieve the media file into the virtual machine context, for persistent storage, and to share it with applications running on other devices. This `fgl_getfile()` feature can be used with a standalone app running on the device, or a client/server app executing on a server and displaying on the device. For more details, see Runtime images on page 1154.

**Related concepts**

- Types of Genero Mobile apps on page 3565
  Genero supports different types of mobile app architectures: development mode, standalone apps, partially-connected apps, and client-server apps.

- `fgl_putfile()` on page 2291
  Transfers a file from the virtual machine context to the front-end context.

- `fgl_getkey()`
  Waits for a keystroke and returns the key number.

**Syntax**

```
fgl_getkey()
```

**Usage**

`fgl_getkey()` waits for a keystroke and returns the key code corresponding to the pressed physical key.
It is recommended to only use this function in text mode.

Unlike fgl_lastkey(), which can return a value indicating the logical effect of whatever key the user pressed, fgl_getkey() returns an integer representing the key code of the physical key that the user pressed. The fgl_getkey() function recognizes the same codes for keys that the fgl_keyval() function returns. Unlike fgl_keyval(), which can only return keystrokes that are entered during dialogs, fgl_getkey() can be called outside a dialog context.

**Related concepts**

fgl_lastkey() on page 2290
Returns the key code corresponding to the logical key that the user most recently typed in the form.

**fgl_getkeylabel()**
Returns the default label associated to a key.

**Syntax**

```c
fgl_getkeylabel(
    keyName STRING )
RETURNS STRING
```

1. *keyName* is the logical name of a key such as F11 or DELETE, INSERT, CANCEL.

**Usage**

The fgl_getkeylabel() function returns the default label defined for the function or control key passed as parameter.

This function returns the default key labels defined for all dialogs. There are different levels of key label definitions.

This function is provided for backward compatibility, use action defaults to define action view texts.

**Related concepts**

Setting action key labels on page 1775
Labels can be defined to decorate buttons controlled by ON KEY / COMMAND KEY action handlers.

Configuring actions on page 1744
Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with action attributes.

**fgl_getresource()**
Returns the value of an FGLPROFILE entry.

**Syntax**

```c
fgl_getresource(
    name STRING )
RETURNS STRING
```

1. *name* is the FGLPROFILE entry name to be read.

**Usage**

The fgl_getresource() function reads the FGLPROFILE file(s) and returns the value defined for the entry passed as parameter.

If the entry does not exist in the configuration file, the function returns NULL.

**Important:** FGLPROFILE entry names are not case sensitive.

If multiple entries are defined with the same name (this can happen especially when several profile files are defined in the FGLPROFILE environment variable), the last entry found wins.
`fgl_getresource()` is a global built-in function, consider using the new `base.Application.getResourceEntry()` method instead.

**Related concepts**
- `base.Application.getResourceEntry` on page 2409
  Returns the value of a FGLPROFILE entry.

`fgl_getversion()`
Returns the product version number of Genero.

**Syntax**

```
fgl_getversion()
RETURNS STRING
```

**Usage**
The `fgl_getversion()` function returns the product version number of the Genero Business Development Language runtime system.

**Important:** This function is provided for debugging only; do not write business code dependent on the build number. The format of the returned value is subject of change in future versions.

`fgl_getwin_height()`
Returns the number of rows of the current window.

**Syntax**

```
fgl_getwin_height()
RETURNS INTEGER
```

**Usage**
The `fgl_getwin_height()` function returns the height of the current window, in character units.

This function is provided for text mode applications, in GUI mode, windows are resizable and thus their height is variable.

**Related concepts**
- `fgl_getwin_width()` on page 2288
  Returns the width of the current window as a number of columns.

`fgl_getwin_width()`
Returns the width of the current window as a number of columns.

**Syntax**

```
fgl_getwin_width()
RETURNS INTEGER
```

**Usage**
The `fgl_getwin_width()` function returns the width of the current window, in character units.

This function is provided for text mode applications, in GUI mode, windows are resizable and thus their width is variable.
**Related concepts**

**fgl_getwin_height()** on page 2288  
Returns the number of rows of the current window.

**fgl_getwin_x()**  
Returns the horizontal position of the current window.

**Syntax**

```plaintext
fgl_getwin_x()
   RETURNS INTEGER
```

**Usage**

The `fgl_getwin_x()` function returns the horizontal coordinate of the top/left corner of the current window. This function is provided for text mode applications, in GUI mode, windows are movable and thus their position is variable.

**Related concepts**

**fgl_getwin_y()** on page 2289  
Returns the vertical position of the current window.

**fgl_getwin_y()**  
Returns the vertical position of the current window.

**Syntax**

```plaintext
fgl_getwin_y()
   RETURNS INTEGER
```

**Usage**

The `fgl_getwin_y()` function returns the vertical coordinate of the top/left corner of the current window. This function is provided for text mode applications, in GUI mode, windows are movable and thus their position is variable.

**Related concepts**

**fgl_getwin_x()** on page 2289  
Returns the horizontal position of the current window.

**fgl_keyval()**  
Returns the key code of a logical or physical key.

**Syntax**

```plaintext
fgl_keyval (  
   keyValue STRING )
   RETURNS INTEGER
```

1. `keyValue` can be a single character, a digit, a printable symbol like @, #, $ or a special keyword such as ACCEPT.

**Usage**

`fgl_keyval()` can be used in form-related statements to examine a value returned by the `fgl_lastkey()` and `fgl_getkey()` functions.
Key names recognized by fgl_keyval() are: ACCEPT, DELETE, DOWN, END, ESC/ESCAPE, HELP, HOME, INSERT, INTERRUPT, LEFT, NEXT/NEXTPAGE, PREVIOUS/PREVPAGE, RETURN, RIGHT, SPACE, TAB, UP, F1 through F64, CONTROL-character (where character can be any letter except A, D, H, I, J, L, M, R, or X).

The function returns NULL if the parameter does not correspond to a valid key.

If you specify a single character, fgl_keyval() considers the case and returns the ASCII value of the character. In all other instances, the function ignores the case of its argument, which can be uppercase or lowercase letters.

To determine whether the user has performed an action, such as inserting a row, specify the logical name of the action (such as INSERT) rather than the name of the physical key (such as F1). For example, the logical name of the Accept action is ACCEPT, while the default physical key is ESCAPE. To test if the key most recently pressed by the user corresponds to the Accept action, specify fgl_keyval("ACCEPT") rather than fgl_keyval("ESCAPE") or fgl_keyval("ESC"). Otherwise, if a key other than ESCAPE is set as the Accept key and the user presses that key, fgl_lastkey() does not return a code equal to fgl_keyval("ESCAPE").

This function is provided for backward compatibility especially for TUI mode applications. fgl_keyval() is well supported in text mode, but this function can only be emulated in GUI mode, because the front-ends communicate with the runtime system with other events as keystrokes.

**Related concepts**

fgl_lastkey() on page 2290

Returns the key code corresponding to the logical key that the user most recently typed in the form.

fgl_getkey() on page 2286

Waits for a keystroke and returns the key number.

**fgl_lastkey()**

Returns the key code corresponding to the logical key that the user most recently typed in the form.

**Syntax**

```
fgl_lastkey()
RETURNS INTEGER
```

**Usage**

The fgl_lastkey() function returns a numeric code corresponding to the user's last keystroke before the function was called. For example, if the last key that the user pressed was a lowercase a, the function returns the code 61.

**Note:** The value of fgl_lastkey() is undefined in a MENU statement.

It is not required to know the specific key codes returned by fgl_lastkey(): The fgl_keyval() function can be used to compare the key code of the last key pressed. The fgl_keyval() function allows you to compare the last key pressed with a logical of physical key. For example, you do not need to know the physical key defined to validate a dialog, you can use the logical name "accept" instead. For a complete list of key codes and logical key names, see the Key code table.

Pay attention to the fact that this function is provided for backward compatibility: The abstract user interface protocol is based on logical events, not only key events. For example, in GUI mode, when selecting a new row with the mouse in a table, there is no key press as when moving in a screen array in TUI mode. However, the runtime system tries to emulate as much as possible keystrokes from non-keystroke events like mouse clicks. Consider reviewing the code logic, in order to use control events of the dialog instruction such as BEFORE FIELD, AFTER FIELD, BEFORE ROW, AFTER ROW, to detect field and record list navigation events.

**Related concepts**

fgl_keyval() on page 2289
Returns the key code of a logical or physical key.

**fgl_mblen()**

Returns the number of bytes of the first character in a string.

**Syntax**

```plaintext
fgl_mblen(
    str STRING )
RETURNS INTEGER
```

1. *str* is any valid string expression.

**Usage**

The `fgl_mblen()` function returns the number of bytes used to encode the first character of the specified string, in the current application locale.

When using a multibyte character set like UTF-8, you can use this function to compute the size in bytes of the first character in the string. Then, you can use this size to identify the position of the next character in the string. This function is mandatory to scan multibyte encoded strings.

If the parameter is NULL, the function returns zero.

If the parameter starts with an invalid multibyte character, the function returns -1.

**Related concepts**

- `length()` on page 2297
  Returns the number of characters in a string passed as parameter.
- `fgl_width()` on page 2296
  Returns the number of columns needed to represent the printed version of the expression.

**fgl_putfile()**

Transfers a file from the virtual machine context to the front-end context.

**Syntax**

```plaintext
fgl_putfile( 
    localPath STRING,  
    remotePath STRING)
```

1. *localPath* is the path to the file to transmit from the virtual machine context.
2. *remotePath* is the path to the file to write on the front-end workstation file system. This parameter is ignored when the front-end is GBC in a web browser.

**Usage**

The `fgl_putfile()` function downloads a file from the application server disk where fglrun is executed to the front-end workstation disk.

**Important:** Using this function can result in a security hole if you allow the end user to specify the file paths without control. There is no limitation on the file content or file paths. If the user executing the application on the server side is allowed to read critical server files, the program could transfer these files on the client workstation. On the other hand, critical files can be written on the client workstation. It is in the hands of the programmer to implement file path and/or file content restrictions in the programs using `fgl_putfile()`.

When using the GBC front-end, the *remotePath* parameter is ignored, because browsers cannot directly write to the disk for security reasons. The basename of the filename specified in the *localPath* parameter will be the name of the file to be downloaded.
For example, when performing the following call (with GBC):

```
CALL fgl_putfile("/var/myapp/files/myfile.txt", "dummy")
```

The "myfile.txt" basename of the source filename will be used to create the local file, in the browser download directory.

**Related concepts**

*fgl_getfile()* on page 2286
Retrieves a file from the front-end context to the virtual machine context.

*fgl_report_print_binary_file()*
Prints a file containing binary data during a report.

**Syntax**

```
fgl_report_print_binary_file(
    path STRING )
```

1. *path* is the name of the binary file.

**Usage**

This function prints a file containing binary data during a report.

This function is provided for backward compatibility and must only be using inside a REPORT routine.

*fgl_report_set_document_handler()*
Redirects the next report to an XML document handler.

**Syntax**

```
fgl_report_set_document_handler(
    handler om.SaxDocumentHandler )
```

1. *handler* is the document handler variable.

**Usage**

This function attaches the specified XML document handler to the next executed report.

The function must be called before the execution of a START REPORT instruction not using the TO XML HANDLER clause.

**Note:** The *fgl_report_set_document_handler()* function is provided for backward compatibility. Use the TO XML HANDLER of START REPORT instead. See XML output for reports on page 2027 for more details.

*fgl_setkeylabel()*
Sets the default label associated to a key.

**Syntax**

```
fgl_setkeylabel(
    keyName STRING,
    text STRING )
```

1. *keyName* is the logical name of a key such as F11 or DELETE, INSERT, CANCEL.
2. *text* is the text associated to the key.
Usage

fgl_setkeylabel() associates a text description to a function or control key. Default action views (buttons that appears in the action panel of a window) will get the label displayed instead of the function or control key name.

This function defines the default key labels for all dialogs. There are different levels of key label definitions.

Note: This feature is supported for backward compatibility. Consider using action attributes to define accelerator keys and decorate actions.

Related concepts

Setting action key labels on page 1775
Labels can be defined to decorate buttons controlled by ON KEY / COMMAND KEY action handlers.

Configuring actions on page 1744
Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with action attributes.

fgl_scr_size()
Returns the size of the specified screen array in the current form.

Syntax

```
fgl_scr_size (  
    name STRING  )  
RETURNS INTEGER
```

1. `name` is the name of a screen-array in the current displayed form.

Usage

The fgl_scr_size() function takes the name of a screen array as parameter identifying an array in the currently-opened form and returns an integer that corresponds to the number of screen records in that screen array.

This function is typically used with traditional text mode forms having screen arrays with a constant size, to display data in screen array rows with the DISPLAY TO instruction.

For modern GUI applications, consider using the UNBUFFERED mode in dialogs, to get automatic form field synchronization with program variables.

Error -1108 will be raised if the passed screen-array does not exits in the current form, and error -1114 is returned if no form is currently displayed.

fgl_setsize()
Sets the size of the main application window.

Syntax

```
fgl_setsize (  
    height INTEGER,  
    width INTEGER  )
```

1. `height` is the number of lines of the window.
2. `width` is the number of columns of the window.

Usage

This function defines the size of the main window when using the traditional GUI mode.

Related concepts

fgl_settitle() on page 2294
Sets the title of the current application window.

**ui.Interface.setSize** on page 2486
Specify the initial size of the parent container window.

### fgl_settitle()
Sets the title of the current application window.

#### Syntax

```
fgl_settitle(
  newTitle STRING )
```

1. `newTitle` is the text of the title.

#### Usage

The `fgl_settitle()` function defines the title of the current window, as well as the default title for new created windows.

This function is provided for backward compatibility, the title of a window can be defined with the `TEXT` attribute of a LAYOUT section.

**Related concepts**

**ui.Interface.setText** on page 2487
 Defines the title for the program.

### fgl_setenv()
Sets the value of an environment variable.

#### Syntax

```
fgl_setenv(
  name STRING,
  value STRING )
```

1. `name` is the name of the environment variable.
2. `value` is the value to be set.

#### Usage

The `fgl_setenv()` function sets or modifies the value of an environment variable.

There is a little difference between Windows® and UNIX™ platforms when passing a `NULL` as the `value` parameter. On Windows® platforms, the environment variable is removed, while on UNIX, the environment variable gets an empty value (it is not removed from the environment).

**Important:** You may experience unexpected results if you change environment variables that are already used by the current program - for example, when you are connected to Informix® and you change the INFORMIXDIR environment variable.

**Related concepts**

**fgl_getenv()** on page 2285
Returns the value of the environment variable.

**fgl_set_arr_curr()**
Moves to a specific row in a record list.

**Syntax**

```plaintext
fgl_set_arr_curr(  
  row INTEGER )
```

1. *row* is the row number is the program array variable.

**Usage**

This function is typically used to control navigation in a **DISPLAY ARRAY** or **INPUT ARRAY**, within an **ON ACTION** or **ON KEY** block. The function can also be used inside **BEFORE DISPLAY** or **BEFORE INPUT** blocks, to jump to a specific row when the dialog starts. It is not recommended to use this function in an other context.

Control blocks like **BEFORE ROW** and field/row validation in **INPUT ARRAY** are performed, as if the user moved to another row, except when the function is called in **BEFORE DISPLAY** or **BEFORE INPUT**.

When a new row is reached using this function, the first editable field gets the focus.

An alternative to the `fgl_set_arr_curr()` function is the `ui.Dialog.setCurrentRow()` method; however, the dialog class method will be used in a different programming pattern, as it does not trigger the control blocks like the built-in function.

**Related concepts**

- **Editable record list (INPUT ARRAY)** on page 1531
  The **INPUT ARRAY** instruction provides always-editable record list handling in an application form.

- **Record list (DISPLAY ARRAY)** on page 1501
  The **DISPLAY ARRAY** instruction provides record list navigation in an application form, with optional record modification actions.

- arr_curr() on page 2272
  Returns the current row in a **DISPLAY ARRAY** or **INPUT ARRAY**.

**fgl_sqldebug()**
Sets the SQL debug level from program code.

**Syntax**

```plaintext
fgl_sqldebug(  
  level INTEGER )
```

1. *level* is the SQL debug level to set. Zero resets the level defined by FGLSQLDEBUG. -1 prints SQL debug information only for SQL statements producing an error.

**Usage**

The **FGLSQLDEBUG** environment variable can be set, to get SQL debug information for all SQL statements executed by a program. The SQL debug output is written to the **stderr** stream.

When program code can be modified and recompiled, the `fgl_sqldebug()` function can be used to force SQL debug log to isolate a specific set of SQL instructions:

- When calling `fgl_sqldebug()` with a value different from zero, it has the same effect as setting FGLSQLDEBUG.
- When calling `fgl_sqldebug()` with zero, it resets the SQL debug level defined by FGLSQLDEBUG.
Related concepts

Debugging SQL statements on page 603

The runtime system can display debug information for SQL statements executed by the program.

**fgl_system()**

Runs a command on the application server.

**Syntax**

```
fgl_system(
  program STRING )
```

1. `program` is the command line to be executed on the server.

**Usage**

The `fgl_system()` function suspends the execution of the program and executes the command passed as parameter on the application server where fglrun is executed. The command is executed in a new shell and the program is suspended until the command terminates.

When running the program in TUI mode, the terminal is switched to line mode before executing the command passed to the `fgl_system()` function.

This function is provided for backward compatibility. In older versions, the function could raise a terminal emulator on the front-end to show the command output on the workstation. This feature is no longer supported.

**fgl_width()**

Returns the number of columns needed to represent the printed version of the expression.

**Syntax**

```
fgl_width(
  str STRING )
```

1. `str` is any valid string expression.

**Usage**

The `fgl_width()` function returns the number of columns that will be used if you display `str` on a text terminal.

If the parameter is `NULL`, the function returns zero.

The number of columns used by a character depends on the glyph (that is the graphical symbol used to draw the character on the screen). For example, an ASCII character like "A" uses one column, while one Chinese character uses 2 columns (this means that on a text terminal, the size of one Chinese character takes the same size as "AB").

Trailing blanks are counted in the length of the string.

Related concepts

length() on page 2297

Returns the number of characters in a string passed as parameter.

fgl_mblen() on page 2291
Returns the number of bytes of the first character in a string.

**fgl_window_getoption()**

Returns attributes of the current window.

**Syntax**

```
fgl_window_getoption(
    name STRING )
RETURNS STRING
```

1. `name` is an window attribute name.

**Usage**

The `fgl_window_getoption()` function returns the value of the window attribute passed as parameter.

Possible parameters are: `name`, `x`, `y`, `width`, `height`, `formline`, `messageline`.

This function is provided for backward compatibility, do not use this function in modern GUI applications.

**length()**

Returns the number of characters in a string passed as parameter.

**Syntax**

```
length(
    str STRING )
RETURNS INTEGER
```

1. `str` is any valid character string expression supported by the language.

**Usage**

The `length()` function counts the length of a character string.

**Note:** Unlike the `STRING.getLength()` method, the `LENGTH()` function does not count the trailing blanks.

If the parameter is `NULL`, the function returns zero.

**Important:** When using byte length semantics, the length is expressed in bytes. When using char length semantics, the unit is characters. This matters when using a multibyte locale such as UTF-8.

**Related concepts**

- `fgl_width()` on page 2296
  Returns the number of columns needed to represent the printed version of the expression.

- `fgl_mblen()` on page 2291
  Returns the number of bytes of the first character in a string.

**set_count()**

Defines the number of rows containing explicit data in a static array used by the next dialog.

**Syntax**

```
set_count(
    count INTEGER )
```

1. `count` defines the number of explicit rows in the static array.
Usage

When using a static array in an `INPUT ARRAY` (with `WITHOUT DEFAULTS` clause) or a `DISPLAY ARRAY` statement, you must specify the number of rows in the array which contain explicit data. In typical applications, these array elements contain the values fetched from a `SELECT` statement.

`set_count()` must be called before a `DISPLAY ARRAY` or `INPUT ARRAY` statement.

The number of rows can also specified with the `COUNT` attribute of `INPUT ARRAY` and `DISPLAY ARRAY` statements.

When using a dynamic array, the number of rows is implicitly defined by the array.

Related concepts

`DYNAMIC ARRAY.getLength` on page 2396
Returns the length of the array.

`arr_curr()` on page 2272
Returns the current row in a `DISPLAY ARRAY` or `INPUT ARRAY`.

`fgl_set_arr_curr()` on page 2295
Moves to a specific row in a record list.

`showhelp()`
Displays a runtime help text.

Syntax

```
showhelp(
    number INTEGER )
```

1. `number` is the help message number in the current help file.

Usage

The `showhelp()` function displays a runtime help text, corresponding to its specified argument, from the current help file defined by the `OPTIONS HELP FILE` instruction.

In GUI mode, the help text will be displayed in a new pop-up window. In TUI mode, the help text is displayed in the whole screen.

Related concepts

`Message files` on page 1160
Message files centralize strings and larger texts identified by a number, that can be used in programs.

`startlog()`
Initializes error logging and opens the error log file passed as the parameter.

Syntax

```
startlog(
    path STRING )
```

1. `path` is the name of the error log file.

Usage

Call `startlog()` in the `MAIN` program block to open or create an error log file. After `startlog()` has been invoked, a record of every subsequent error that occurs during the program execution is written in the error log file.

**Important:** Sensitive and personal data may be written to the output. Make sure that the log output is written to files that can only be read by application administrators.
The format of the error records appended to the error log file after each subsequent error is as follows:

```
Date: 03/06/99 Time: 12:20:20
Program error at "stock_one.4gl", line number 89.
SQL statement error number -239.
Could not insert new row - duplicate value in a UNIQUE INDEX column.
SYSTEM error number -100
ISAM error: duplicate value for a record with unique key.
```

To report specific application errors, use the `errorlog()` function to make an entry in the error log file.

If the argument of `startlog()` is not the name of an existing file, `startlog()` creates a new one. If the file already exists, `startlog()` opens it and positions the file pointer so that subsequent error messages can be appended to this file.

**Example**

```fql
MAIN
    CALL startlog("/tmp/error-" || fgl_getpid() || ".log")
    CALL errorlog("Current user is not allowed to perform this operation")
END MAIN
```

**Related concepts**

- `errorlog()` on page 2275
  Copies the string passed as parameter into the error log file.
- `Exceptions` on page 502
  Describes exception (error) handling in the programs.

**Related reference**

- `Genero BDL errors` on page 3154
  System error messages sorted by error number.

- `upshift()`
  Converts a string to uppercase.

**Syntax**

```fql
upshift(
    s STRING
) RETURNS STRING
```

1. `s` is the character string to convert to uppercase letters.

**Usage**

The `upshift()` function returns a string value in which all lowercase characters in its argument are converted to uppercase.

The character conversion depends on `locale settings` (the LC_CTYPE environment variable). Non-alphabetic or uppercase characters are not altered.

**Related concepts**

- `downshift()` on page 2272
Converts a string to lowercase.

### List of desupported built-in functions

#### Table 487: Desupported built-in functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FGL_FORMFIELD_GETOPTION()</td>
<td>Returns attributes of a specified form field.</td>
</tr>
<tr>
<td>FGL_GETUITYPE()</td>
<td>Returns the type of the front-end.</td>
</tr>
<tr>
<td>FGL_WINDOW_OPEN()</td>
<td>Opens a new window with coordinates and size.</td>
</tr>
<tr>
<td>FGL_WINDOW_OPENWITHFORM()</td>
<td>Opens a new window with coordinates and form.</td>
</tr>
<tr>
<td>FGL_WINDOW_CLEAR()</td>
<td>clears the window having the name that is passed as a parameter.</td>
</tr>
<tr>
<td>FGL_WINDOW_CLOSE()</td>
<td>Closes the window having the name that is passed as a parameter.</td>
</tr>
<tr>
<td>FGL_WINDOW_CURRENT()</td>
<td>Makes current the window having the name that is passed as a parameter.</td>
</tr>
</tbody>
</table>

**Related reference**

[Built-in functions](#) on page 2270

### The key code table

This table lists internal key codes. Avoid hard-coding these numbers in your sources; otherwise the source will not be compatible with future versions of the language.

Always use the `fgl_keyval()` function instead.

#### Table 488: Internal key codes

<table>
<thead>
<tr>
<th>Value</th>
<th>Key name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 to 26</td>
<td>Control-x</td>
<td>Control key, where x is the any letter from A to Z. The key code corresponding to Control-A is 1, Control-B is 2, etc.</td>
</tr>
<tr>
<td>others &lt; 256</td>
<td>ASCII chars</td>
<td>Other codes correspond to the ASCII characters set.</td>
</tr>
<tr>
<td>2000</td>
<td>up</td>
<td>The up-arrow logical key.</td>
</tr>
<tr>
<td>2001</td>
<td>down</td>
<td>The down-arrow logical key.</td>
</tr>
<tr>
<td>2002</td>
<td>left</td>
<td>The left-arrow logical key.</td>
</tr>
<tr>
<td>2003</td>
<td>right</td>
<td>The right-arrow logical key.</td>
</tr>
<tr>
<td>2005</td>
<td>nextpage</td>
<td>The next-page logical key.</td>
</tr>
<tr>
<td>2006</td>
<td>prevpage</td>
<td>The previous-page logical key.</td>
</tr>
<tr>
<td>2008</td>
<td>help</td>
<td>The help logical key.</td>
</tr>
<tr>
<td>2011</td>
<td>interrupt</td>
<td>The interrupt logical key.</td>
</tr>
<tr>
<td>2020</td>
<td>home</td>
<td>The home logical key.</td>
</tr>
<tr>
<td>2021</td>
<td>end</td>
<td>The end logical key.</td>
</tr>
<tr>
<td>2016</td>
<td>accept</td>
<td>The accept logical key.</td>
</tr>
<tr>
<td>2017</td>
<td>backspace</td>
<td>The backspace logical key.</td>
</tr>
<tr>
<td>Value</td>
<td>Key name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>----------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>3000 to 3255</td>
<td>Fx</td>
<td>Function key, where x is the number of the function key. The key code corresponding to a function key Fx is 3000+x-1, for example, 3011 corresponds to F12.</td>
</tr>
</tbody>
</table>

**Utility modules**

A utility function is a function provided in a separate library; it is not built in the runtime system.

To use a utility function, declare the module where the function is defined with the IMPORT FGL instruction:

```plaintext
IMPORT FGL fgldialog
...
CALL fgl_winmessage(...)
```

For backward compatibility, utility functions are also grouped in a 42x library named libfgl4js.42x, which can be linked to your programs.

The 42x library file, 42m modules and 42f forms are located in $FGLDIR/lib. The sources of the utility functions and form files are provided in the $FGLDIR/src directory.
**fgldialog: Common dialog functions**

**Table 489: Common dialog functions (fgldialog.4gl)**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fgl_winbutton</code></td>
<td>Displays an interactive message box containing multiple choices, in a pop-up window.</td>
</tr>
<tr>
<td><code>fgl_winmessage</code></td>
<td>Displays an interactive message box containing text and OK button.</td>
</tr>
<tr>
<td><code>fgl_winprompt</code></td>
<td>Displays a dialog box containing a field that accepts a value.</td>
</tr>
<tr>
<td><code>fgl_winquestion</code></td>
<td>Displays an interactive message box with configurable Ok/Yes/No/Cancel/Ignore/Abort/Retry buttons.</td>
</tr>
<tr>
<td><code>fgl_winwait</code></td>
<td>Displays an interactive message box and waits for user validation.</td>
</tr>
</tbody>
</table>

**fgl_winbutton()**
Displays an interactive message box containing multiple choices, in a pop-up window.

**Syntax**

```c
fgl_winbutton(
    title STRING,
    text STRING,
    default STRING,
    buttons STRING,
    icon STRING,
    danger SMALLINT )
RETURNS STRING
```
1. *title* defines the title of the message window.
2. *text* specifies the string displayed in message window.
3. *default* indicates the default button to be pre-selected.
4. *buttons* defines a set of button labels separated by "|".
5. *icon* is the name of the icon to be displayed.
6. *danger* (for X11 only), number of warning items. Otherwise, this parameter is ignored.

**Usage**

Use the `fgl_winbutton()` function to open a message box and let the end user select an option in a set of buttons. The function returns the label of the button which has been selected by the user.

Use "\n" in *text* to separate lines (this does not work in TUI mode).

Supported names for the *icon* parameter are: `information`, `exclamation`, `question`, `stop`.

You can define up to 7 buttons that each have 10 characters.

If two buttons start with the same letter, the user will not be able to select one of them in the TUI mode.

The "&" before a letter for a button is displayed in TUI mode, or underlines the next letter in graphical front-ends.

This function is provided for backward compatibility, use a menu with "dialog" style instead.

**Example**

```plaintext
IMPORT FGL fgldialog
MAIN
  DEFINE answer STRING
  LET answer = fgl_winbutton( "Media selection", "What is your favorite media?", "Lynx", "Floppy Disk|CD-ROM|DVD-ROM|Other", "question", 0)
  DISPLAY "Selected media is: " || answer
END MAIN
```

`fgl_winmessage()`

Displays an interactive message box containing text and OK button.

**Syntax**

```
fgl_winmessage(
  title STRING,
  text STRING,
  icon STRING )
```

1. *title* defines message box title.
2. *text* is the text displayed in the message box. Use ‘\n’ to separate lines.
3. *icon* is the name of the icon to be displayed.

**Usage**

The `fgl_winmessage()` function displays a message box to the end user.

**Important:** With front-ends implementing this function with a system dialog box API creating a modal window, the end user will have to close the modal window first, before continuing within the window of another program. Consider using a menu with "dialog" style instead so as not to block other programs.

Supported names for the *icon* parameter are: `information`, `exclamation`, `question`, `stop`. Note that on some front-ends such as iOS devices, the native message pop-up window does not display an image.
On front-ends using a system dialog box API, the OK buttons will be automatically localized based on the operating system language settings. On other front-ends, the option buttons will be decorated depending on action default settings.

**Example**

```fgl
IMPORT FGL fgldialog
MAIN
    CALL fgl_winmessage( "Title", "This is a critical message.", "stop")
END MAIN
```

**Related concepts**

- [Configuring actions](#) on page 1744

Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with *action attributes*.

### fgl_winprompt()

Displays a dialog box containing a field that accepts a value.

#### Syntax

```fgl
fgl_winprompt ( 
    x INTEGER,
    y INTEGER,
    text STRING,
    default STRING,
    length INTEGER,
    type INTEGER )
RETURNS STRING
```

1. `x` is the column position in characters.
2. `y` is the line position in characters.
3. `text` is the message shown in the box.
4. `default` is the default value.
5. `length` is the maximum length of the input value.
6. `type` is the data type of the return value.

#### Usage

The *fgl_winprompt()* function allows the end user to enter a value.

This function is provided for backward compatibility. You can also use your own input dialog with a customized *form* to get a value from the user. Or use the standard *PROMPT* instruction.

Possible values for the type parameter are: 0=CHAR, 1=SMALLINT, 2=INTEGER, 7=DATE, 255=invisible

Avoid passing NULL values.

#### Example

```fgl
IMPORT FGL fgldialog
MAIN
    DEFINE answer DATE
    LET answer = fgl_winprompt( 10, 10, "Today", DATE, 10, 7 )
    DISPLAY "Today is " || answer
END MAIN
```
fgl_winquestion()
Displays an interactive message box with configurable Ok/Yes/No/Cancel/Ignore/Abort/Retry buttons.

Syntax

```
fgl_winquestion(
    title STRING,
    text STRING,
    default STRING,
    buttons STRING,
    icon STRING,
    danger SMALLINT )
RETURNS STRING
```

1. `title` is the message box title.
2. `text` is the message displayed in the message box. Use '\n' to separate lines (does not work on ASCII client).
3. `default` defines the default button that is preselected.
4. `buttons` defines the options. Must be a pipe-separated list of 2 or three options: ok, yes, no, cancel, abort, retry, ignore.
5. `icon` is the name of the icon to be displayed.
6. `danger` is supported for backward compatibility. This parameter is ignored.

Usage

The `fgl_winquestion()` function shows a question message box to the end user and waits for an answer.

**Important:** With front-ends implementing this function with a system dialog box API creating a modal window, the end user will have to close the modal window first, before continuing within the window of another program. Consider using a menu with 'dialog' style instead so as not to block other programs.

The function returns the label of the option which has been selected by the user.

Supported names for the `icon` parameter are: information, exclamation, question, stop. Note that on some front-ends such as iOS devices, the native message pop-up window does not display an image.

The `buttons` parameter defines the list of options that the user can select. Possible values are: ok, yes, no, cancel, abort, retry, ignore. You must specify a pipe-separated list of options, with a maximum of 3 options. For example: "ok", "yes|no", "yes|no|cancel", "abort|retry|ignore".

**Important:** To display the pop-up window of this API, desktop and mobile front-ends use the platform specific message box API, with a predefined set of buttons. Some non-standard option combinations may not be supported, such as "ok|yes|abort". Furthermore, the order of the buttons depends also on platform standards. For example, with "abort|retry|ignore", the buttons can appear in the following order: [Retry] [Ignore] [Abort].

On front-ends using a system dialog box API, the option buttons will be automatically localized based on the operating system language settings. On other front-ends, the option buttons will be decorated to action default settings.

Example

```
IMPORT FGL fgldialog
MAIN
    DEFINE answer STRING
    LET answer = "yes"
    WHILE answer = "yes"
        LET answer = fgl_winquestion(
            "Procedure", "Would you like to continue ? ",
            "cancel", "yes|no|cancel", "question", 0)
    END WHILE
    IF answer = "cancel" THEN
```
Related concepts
Configuring actions on page 1744
Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with action attributes.

`fgl_winwait()`
Displays an interactive message box and waits for user validation.

**Syntax**

```plaintext
fgl_winwait(
    text STRING
)
```

1. `text` is the message displayed in the message box. Use '\n' to separate lines (not working on ASCII client).

**Usage**
The `fgl_winwait()` function displays a message to the end user and waits until the user presses the OK button.

**Important:** With front-ends implementing this function with a system dialog box API creating a modal window, the end user will have to close the modal window first, before continuing within the window of another program. Consider using a menu with "dialog" style instead so as not to block other programs.
### fgldbutl: Database utility module

#### Table 490: Database utility functions (fgldbutl.4gl)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Important: This feature is deprecated, its use is discouraged although not prohibited.</td>
<td>Returns the database type for the current connection.</td>
</tr>
<tr>
<td><code>db_get_database_type()</code></td>
<td>[RETURNS STRING]</td>
</tr>
<tr>
<td><code>db_get_last_serial()</code></td>
<td>[emultype STRING, tablename STRING ] RETURNS BIGINT \ Returns the last generated serial for a given serial emulation and database table.</td>
</tr>
<tr>
<td><code>db_get_sequence()</code></td>
<td>[id STRING ] RETURNS BIGINT \ Generates a new sequence for a given identifier.</td>
</tr>
<tr>
<td><code>db_start_transaction()</code></td>
<td>[RETURNS INTEGER] \ Starts a nested transaction call.</td>
</tr>
<tr>
<td><code>db_finish_transaction()</code></td>
<td>[commit INTEGER ] RETURNS INTEGER \ Terminates a nested transaction call.</td>
</tr>
<tr>
<td><code>db_is_transaction_started()</code></td>
<td>[RETURNS INTEGER] \ Indicates whether a nested transaction call is started.</td>
</tr>
</tbody>
</table>

**db_get_database_type()**

Returns the database type for the current connection.

**Syntax**

**Important:** This feature is deprecated, its use is discouraged although not prohibited.

**Usage**

After connecting to the database, you can get the type of the database server with this function.

**Important:** This function is deprecated, use the `fgl_dbdriver_type()` function instead.

#### Table 491: Codes returned by `db_get_database_type()` per database type

<table>
<thead>
<tr>
<th>DB Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASE</td>
<td>SAP® ASE</td>
</tr>
</tbody>
</table>
### db_get_last_serial()

Retrieves the last generated serial for a given serial emulation and database table.

**Syntax**

```sql
db_get_last_serial(
    emultype STRING,
    tabname STRING )
RETURNS BIGINT
```

1. `emultype` is the serial emulation type ("native", "native2", "regtable", "trigseq").
2. `tabname` is the name of the database table (case sensitive, use lowercase).

**Usage**

This function executes the SQL query to retrieve the last generated auto-incremented columns, for the given serial emulation type and database table.

This function must be used after an `INSERT` statement producing a new serial value, especially when the serial is based on the `BIGINT` type (because `SQLCA.SQLERRD[2]` can only hold `INTEGER` values).

**Note:**

When using this function, configure your connection to avoid the automatic retrieval of the last generated serial to fill `SQLCA.SQLERRD[2]`, by setting the following FGLPROFILE entry:

```sql
dbi.database.dbname.ifxemul.datatype.serial.sqlerrd2 = false
```

**Example**

```plaintext
IMPORT FGL fgldbutl
MAIN
    DEFINE ns BIGINT
    DATABASE mydb
    INSERT INTO mytable VALUES ( ns, 'a new sequence' )
    LET ns = db_get_last_serial("native","mytable")
END MAIN
```

**Related concepts**

*Auto-incremented columns (serials) on page 626*
How to implement automatic record keys.

**db_get_sequence()**
Generates a new sequence for a given identifier.

**Syntax**

```plaintext
db_get_sequence(  
id STRING  )  
RETURNS BIGINT
```

1. `id` is the identifier of the sequence.

**Usage**
This function generates a new sequence from a register table created in the current database.

**Important:**
1. Needs a database table called SEQREG.
2. The function must be called inside a transaction block.

The table must be created as follows:

```sql
CREATE TABLE seqreg (  
    sr_name VARCHAR(30) NOT NULL,  
    sr_last BIGINT NOT NULL,  
    PRIMARY KEY (sr_name)  
)
```

Each time you call this function, the sequence is incremented in the database table and returned by the function.

It is mandatory to use this function inside a transaction block, in order to generate unique sequences.

**Example**

```plaintext
IMPORT FGL fgldbut1  
MAIN  
    DEFINE ns BIGINT, s INTEGER  
    DATABASE mydb  
    BEGIN WORK  
    LET ns = db_get_sequence("mytable")  
    INSERT INTO mytable VALUES ( ns, 'a new sequence' )  
    COMMIT WORK  
END MAIN
```

**Related concepts**

- [Auto-incremented columns (serials)](page 626) on page 626
- How to implement automatic record keys.

**db_start_transaction()**
Starts a nested transaction call.

**Syntax**

```plaintext
db_start_transaction()  
RETURNS INTEGER
```
## Usage

On most database engines, you can only have a unique transaction, that is started with `BEGIN WORK` and ended with `COMMIT WORK` or `ROLLBACK WORK`. But in some cases, you may need to do complex nested function calls, executing several SQL instructions that must all be grouped in a single transaction. The nested transaction utility functions help you to implement this.

With this nested transaction technique, you encapsulate transaction start and end within the utility function. Custom functions doing SQL operations can then be reused in different parts of your application. If the caller does not start the transaction, the called function will automatically start and end the transaction.

The `db_start_transaction()` function encapsulates the `BEGIN WORK` instruction to start a transaction, in order to implement nested transactions.

**Note:** These transaction encapsulation functions are provided for special cases, where the function call graph is complex. In general, it is recommended that you simply use the standard `BEGIN WORK / COMMIT WORK / ROLLBACK WORK` instructions to implement transaction blocks.

These transaction management functions execute a real transaction instruction at the boundaries of the subsequent start/finish calls.

### Example

```fortran
IMPORT FGL fgldbutl

MAIN
  DEFINE s INTEGER
  DATABASE mydb
  LET s = db_start_transaction() -- real BEGIN WORK
  IF s != 0 THEN DISPLAY "error 1" END IF
  WHENEVER ERROR CONTINUE
  UPDATE customer SET cust_name = 'Undef'
  WHENEVER ERROR STOP
  LET s = SQLCA.SQLCODE
  IF s != 0 THEN
    DISPLAY "error 2"
  ELSE
    LET s = do_update()
    IF s != 0 THEN DISPLAY "error 3" END IF
    END IF
  LET s = db_finish_transaction(s==0) -- real COMMIT or ROLLBACK WORK
  IF s != 0 THEN DISPLAY "error 4" END IF
END MAIN

FUNCTION do_update()
  DEFINE s INTEGER
  LET s = db_start_transaction() -- no SQL command (nested)
  IF s != 0 THEN
    DISPLAY "error 1.1"
  ELSE
    WHENEVER ERROR CONTINUE
    UPDATE customer SET cust_status = 'X'
    WHENEVER ERROR STOP
    LET s = SQLCA.SQLCODE
    IF s != 0 THEN
      DISPLAY "error 1.2"
    END IF
    END IF
  LET s = db_finish_transaction(s==0) -- no SQL command (nested)
  IF s != 0 THEN DISPLAY "error 1.3" END IF
  RETURN s
END FUNCTION
```
Related concepts

**Database transactions** on page 688
Database transaction concepts and handling.

**db_finish_transaction()**
Terminates a nested transaction call.

**Syntax**

```plaintext
db_finish_transaction(
    commit INTEGER )
RETURNS INTEGER
```

1. *commit* is a boolean that indicates whether the transaction must be committed.

**Usage**

This function encapsulates the **COMMIT WORK** or **ROLLBACK WORK** instructions to end a transaction.

When the number of calls to `DB_START_TRANSACTION()` matches, this function executes a **COMMIT WORK** if the passed parameter is TRUE; if the passed parameter is FALSE, it executes a **ROLLBACK WORK**.

If the number of start/finish calls does not match, the function does nothing.

**Related concepts**

**Database transactions** on page 688
Database transaction concepts and handling.

**db_is_transaction_started()**
Indicates whether a nested transaction call is started.

**Syntax**

```plaintext
db_is_transaction_started()
RETURNS INTEGER
```

**Usage**

The function returns TRUE if a transaction was started with `db_start_transaction()`, and was not yet finished with a call to the `db_finish_transaction()` function.

**Related concepts**

**Database transactions** on page 688
**fglwinexec: Front-end dialogs module**

**Table 492: Front-end-side dialog functions (fglwinexec.4gl) (deprecated: use ui.Interface.frontCall() instead)**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>winopendir</strong> (dirname STRING, caption STRING) RETURNS STRING</td>
<td>Opens a dialog window to get a directory path on the front-end workstation.</td>
</tr>
<tr>
<td><strong>winopenfile</strong> (dirname STRING, typename STRING, extlist STRING, caption STRING) RETURNS STRING</td>
<td>Opens a dialog window to get a file to be read on the front-end workstation.</td>
</tr>
<tr>
<td><strong>winsavefile</strong> (dirname STRING, typename STRING, extlist STRING, caption STRING) RETURNS STRING</td>
<td>Opens a dialog window to get a file path to save data on the front-end workstation.</td>
</tr>
<tr>
<td><strong>winshellexec</strong> (filename STRING)</td>
<td>Opens a document on the workstation where the Windows® front-end runs. <strong>Note:</strong> Microsoft™ Windows® only!</td>
</tr>
<tr>
<td><strong>winexecwait</strong> (command STRING) RETURNS INTEGER</td>
<td>Executes a program on the workstation where the Windows® front-end runs and waits for termination. <strong>Note:</strong> Microsoft™ Windows® only!</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>Important:</strong> This feature is deprecated, its use is discouraged although not prohibited.</td>
<td>Executes a program on the workstation where the Windows® front-end runs and returns immediately. <strong>Note:</strong> Microsoft™ Windows® only!</td>
</tr>
</tbody>
</table>
| **winexec(** | \[
\text{command STRING}
\]
RETURNS INTEGER |\| Executes a program on the workstation where the Windows® front-end runs and returns immediately. **Note:** Microsoft™ Windows® only! |

**winopendir()**

Opens a dialog window to get a directory path on the front-end workstation.

**Syntax**

**Important:** This feature is deprecated, its use is discouraged although not prohibited.

\[
\text{winopendir(}
\text{dirname STRING,}
\text{caption STRING }
\text{)}
\text{RETURNS STRING}
\]

1. `dirname` is the default path to be displayed in the dialog window.
2. `caption` is the label to be displayed.

**Usage**

This function opens a dialog window to let the user select a directory path on the front-end workstation file system.

The function returns the directory path on success.

The function returns **NULL** if a problem has occurred or if the user canceled the dialog.

**Note:** The function must be called after the front-end connection has been established. Do not use this function when using another front-end as the Genero Desktop Client (GDC).

**Related concepts**

`openDir` on page 2658
Displays a file dialog window to get a directory path on the local file system.

**winopenfile()**

Opens a dialog window to get a file to be read on the front-end workstation.

**Syntax**

**Important:** This feature is deprecated, its use is discouraged although not prohibited.

\[
\text{winopenfile(}
\text{dirname STRING,}
\text{typename STRING,}
\text{extlist STRING,}
\text{caption STRING }
\text{)}
\text{RETURNS STRING}
\]

1. `dirname` is the default path to be displayed in the dialog window.
2. `typename` is the name of the file type to be displayed.
3. `extlist` is a blank-separated list of file extensions defining the file type.
4. `caption` is the label to be displayed.
**Usage**

This function opens a dialog window to let the user select a file path on the front-end workstation file system, in order to open the file.

The function returns the file path on success.

The function returns `NULL` if a problem has occurred or if the user canceled the dialog.

**Note:** The function must be called after the front-end connection has been established. Do not use this function when using another front-end as the Genero Desktop Client (GDC).

**Related concepts**

- **openFile** on page 2659
  Displays a file dialog window to let the user select a single file path on the local file system.

- **winsavefile()**
  Opens a dialog window to get a file path to save data on the front-end workstation.

**Syntax**

**Important:** This feature is deprecated, its use is discouraged although not prohibited.

```plaintext
winsavefile ( 
  dirname STRING, 
  typename STRING, 
  extlist STRING, 
  caption STRING ) 
RETURNS STRING
```

1. `dirname` is the default path to be displayed in the dialog window.
2. `typename` is the name of the file type to be saved.
3. `extlist` is a blank separated list of file extensions defining the file type.
4. `caption` is the label to be saved.

**Usage**

This function opens a dialog window to let the user select a file path on the front-end workstation file system, in order to save the file.

The function returns the file path on success.

The function returns `NULL` if a problem has occurred or if the user canceled the dialog.

**Note:** The function must be called after the front-end connection has been established. Do not use this function when using another front-end as the Genero Desktop Client (GDC).

**Related concepts**

- **saveFile** on page 2662
  Displays a file dialog window to get a path to save a file on the local file system.

- **winexec() MS Windows® FE Only!**
  Executes a program on the workstation where the Windows® front-end runs and returns immediately.

**Syntax**

**Important:** This feature is deprecated, its use is discouraged although not prohibited.

```plaintext
winexec ( 
  command STRING ) 
RETURNS INTEGER
```
1. *command* is the command to be executed on the front-end.

**Usage**

The function executes the program on the Windows® front end and returns the control to the program without waiting.

**Note:** The function must be called after the front-end connection has been established. Do not use this function when using another front-end as the Genero Desktop Client (GDC).

**winexecwait() MS Windows® FE Only!**

Executes a program on the workstation where the Windows® front-end runs and waits for termination.

**Syntax**

**Important:** This feature is deprecated, its use is discouraged although not prohibited.

```plaintext
winexecwait(
    command STRING
) RETURNS INTEGER
```

1. *command* is the command to be executed on the front-end.

**Usage**

The function executes the program on the Windows® front end and waits for its termination.

**Note:** The function must be called after the front-end connection has been established. Do not use this function when using another front-end as the Genero Desktop Client (GDC).

**winshellexec() MS Windows® FE Only!**

Opens a document on the workstation where the Windows® front-end runs.

**Syntax**

**Important:** This feature is deprecated, its use is discouraged although not prohibited.

```plaintext
winshellexec(
    filename STRING
) RETURNS INTEGER
```

1. *filename* is the file to be opened on the front-end.

**Usage**

The function opens a document on the Windows® front-end without waiting.

**Note:** The function must be called after the front-end connection has been established. Do not use this function when using another front-end as the Genero Desktop Client (GDC).
**VCard: VCF file format module**

**Table 493: vCard types (VCard.4gl)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUBLIC TYPE VCAAddress RECORD</td>
<td></td>
</tr>
<tr>
<td>PostOfficeBox, ExtendedAddress, -- apartment</td>
<td></td>
</tr>
<tr>
<td>Street, City, State, ZIP, Country STRING --, CountryCode STRING -- X-ABADR:de</td>
<td></td>
</tr>
<tr>
<td>END RECORD</td>
<td></td>
</tr>
</tbody>
</table>

| PUBLIC TYPE VCName RECORD |
| FirstName, LastName, MiddleName, Prefix, Suffix STRING |
| --, FormattedName STRING |
| END RECORD             |

| PUBLIC TYPE VCPerson RECORD |
| FirstName STRING, -- N[1] |
| LastName STRING, -- N[2] |
| MiddleName STRING, -- N[3] |
| Prefix STRING, -- N[4] |
| Suffix STRING, -- N[5] |
| formattedName STRING, -- FN |
| nickname STRING, -- NICKNAME |
| jobTitle STRING, -- TITLE |
| organization STRING, -- ORG.value[1] |
| department STRING, -- ORG.value[2] |
| birthday STRING, -- BDAY |
| note STRING, -- NOTE |
| address DYNAMIC ARRAY OF RECORD |
| type STRING, PostOfficeBox, -- ADR[1] |
| ExtendedAddress, -- ADR[2] |
| Street, -- ADR[3] |
| City, -- ADR[4] |
| State, -- ADR[5] |
| ZIP, -- ADR[6] |
| Country STRING -- ADR[7] |
| END RECORD |
| phone DYNAMIC ARRAY OF RECORD |
| type STRING, number STRING -- TEL |
| END RECORD |
| email DYNAMIC ARRAY OF RECORD |
| type STRING, value STRING -- EMAIL |
| END RECORD |

The VCAAddress structured type holds vCard address data.

The VCName structured type holds vCard data related to the person's name.

The VCPerson structured type holds vCard data.
### Table 494: vCard functions (VCard.4gl)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>format_person</code></td>
<td>Converts a VCPerson record to a vCard string representation vCard.</td>
</tr>
<tr>
<td><code>scan_address</code></td>
<td>Extracts an address from a string representing a vCard.</td>
</tr>
<tr>
<td><code>scan_email</code></td>
<td>Extracts an email from a string representing a vCard.</td>
</tr>
<tr>
<td><code>scan_person</code></td>
<td>Extracts person's data from a string representing a vCard.</td>
</tr>
<tr>
<td><code>scan_phone</code></td>
<td>Extracts a phone number from a string representing a vCard.</td>
</tr>
</tbody>
</table>

#### VCAddress type

The VCAddress structured type holds vCard address data.

#### Syntax

```plaintext
PUBLIC TYPE VCAddress RECORD
    PostOfficeBox,
    ExtendedAddress, -- apartment or suite number
    Street,
    City,
    State,
    ZIP,
    Country STRING
    -- , CountryCode STRING -- X-ABADR:de
END RECORD
```

#### Usage

This type defines a record structure to hold vCard address information. It is used for values returned by the `scan_address()` function.

#### Example

```plaintext
IMPORT FGL VCard
MAIN
    DEFINE a VCard.VCAddress
    LET a.Street = "Sunset Bld"
```
VCName type
The VCName structured type holds vCard data related to the person's name.

Syntax

PUBLIC TYPE VCName RECORD
    FirstName,
    LastName,
    MiddleName,
    Prefix,
    Suffix STRING
    --, FormattedName STRING
END RECORD

Usage
This type defines a record structure to hold vCard information related to the person's name. It is used for values returned by the scan_name() function.

Example

IMPORT FGL VCard
MAIN
    DEFINE n VCard.VCName
    LET n.FirstName = "Hans"
    LET n.LastName = "Mustermann"
END MAIN

VCPerson type
The VCPerson structured type holds vCard data.

Syntax

PUBLIC TYPE VCPerson RECORD
    FirstName STRING, -- N[1]
    LastName STRING, -- N[2]
    MiddleName STRING, -- N[3]
    Prefix STRING, -- N[4]
    Suffix STRING, -- N[5]
    formattedName STRING, -- FN
    nickname STRING, -- NICKNAME
    jobTitle STRING, -- TITLE
    organization STRING, -- ORG.value[1]
    department STRING, -- ORG.value[2]
    birthday STRING, -- BDAY
    note STRING, -- NOTE
    address DYNAMIC ARRAY OF RECORD
        type STRING,
        PostOfficeBox, -- ADR[1]
        ExtendedAddress, -- ADR[2]
        Street, -- ADR[3]
        City, -- ADR[4]
        State, -- ADR[5]
        ZIP, -- ADR[6]
        Country STRING -- ADR[7]
    END RECORD,
    phone DYNAMIC ARRAY OF RECORD
        type STRING,
Usage
This type defines a record structure to hold vCard information. It is used by VCard functions such as format_person().

Example

```fgl
IMPORT FGL VCard
MAIN
  DEFINE p VCard.VCPerson
  LET p.FirstName = "Hans"
  LET p.LastName = "Mustermann"
END MAIN
```

format_person()
Converts a VCPerson record to a vCard string representation vCard.

Syntax

```
format_person(
  person VCPerson )
RETURNS STRING
```

1. `person` is a VCPerson record.

Usage
This function converts a record defined with the VCPerson type, to a string representing a vCard. The returned value is a version 3.0 vCard formatted string.

Example

```fgl
IMPORT FGL VCard
MAIN
  DEFINE p VCard.VCPerson
  LET p.FirstName = "Hans"
  LET p.LastName = "Mustermann"
  LET p.email[1].VALUE = "hans@nomail.com"
  LET p.phone[1].TYPE = "HOME"
  LET p.phone[1].number = "+49 123 4567 8901"
  LET p.phone[2].TYPE = "WORK"
  LET p.phone[2].number = "+49 123 9876 5431"
  DISPLAY VCard.format_person(p.*)&
END MAIN
```

Output:

```
BEGIN:VCARD
VERSION:3.0
N:Hans;Mustermann;;
FN:Hans Mustermann
TEL;TYPE=HOME:+49 123 4567 8901
TEL;TYPE=WORK:+49 123 9876 5431
EMAIL:hans@nomail.com
```
scan_address()  
Extracts an address from a string representing a vCard.

Syntax

```plaintext
scan_address(
  source STRING,
  type STRING
) RETURNS VCAddress
```

1. `source` is the vCard string (version 3.0).
2. `type` is the type of address (HOME, WORK, pref).

Usage

This function parses the vCard string passed as parameter to find address data based on a type, and returns address information in a record defined with the VCAddress type.

The function looks for lines starting with the "ADR" keyword.

The second parameter (`type`) defines the value of the "TYPE" attribute in an "ADR" line. Values can, for example, be "HOME", "WORK", "pref". If this parameter is `NULL`, the address with TYPE=pref will be returned. If no preferred address exists, the first address will be returned.

Example

```plaintext
IMPORT FGL VCard
MAIN
  DEFINE a VCard.VCAddress,
    f TEXT
  LOCATE f IN FILE arg_val(1)
  CALL VCard.scan_address(f,"WORK") RETURNING a.*
  DISPLAY a.*
END MAIN
```

scan_email()  
Extracts an email from a string representing a vCard.

Syntax

```plaintext
scan_email(
  source STRING,
  type STRING
) RETURNS STRING
```

1. `source` is the vCard string (version 3.0).
2. `type` is the type of email (HOME, WORK, pref).

Usage

This function parses the vCard string passed as parameter to find "EMAIL" data based on a type, and returns the email address as a string.

The function looks for lines starting with the "EMAIL" keyword.

The second parameter (`type`) defines the value of the "TYPE" attribute in an "EMAIL" line. Values can, for example, be "HOME", "WORK", "pref". If this parameter is `NULL`, the email with TYPE=pref will be returned. If no preferred email exists, the first email will be returned.
Example

```fgl
IMPORT FGL VCard
MAIN
  DEFINE m STRING,
       f TEXT
  LOCATE f IN FILE arg_val(1)
  CALL VCard.scan_email(f,NULL) RETURNING m
  DISPLAY m
END MAIN
```

**scan_name()**

Extracts name information from a string representing a vCard.

**Syntax**

```fgl
scan_name(
  source STRING)
RETURNS VCName
```

1. *source* is the vCard string (version 3.0).

**Usage**

This function parses the vCard string passed as parameter to find a person's name data, and returns name information in a record defined with the **VCName** type.

**Example**

```fgl
IMPORT FGL VCard
MAIN
  DEFINE n VCard.VCName,
       f TEXT
  LOCATE f IN FILE arg_val(1)
  CALL VCard.scan_name(f) RETURNING n.*
  DISPLAY n.*
END MAIN
```

**scan_person()**

Extracts person's data from a string representing a vCard.

**Syntax**

```fgl
scan_person(
  source STRING)
RETURNS VCPerson
```

1. *source* is the vCard string (version 3.0).

**Usage**

This function parses the vCard string passed as parameter, extracts all information, and returns a record defined with the **VCPerson** type.

**Example**

```fgl
IMPORT FGL VCard
MAIN
  DEFINE p VCard.VCPerson,
```
f TEXT
LOCATE f IN FILE arg_val(1)
CALL VCard.scan_person(f) RETURNING p.*
DISPLAY p.*
END MAIN

scan_phone()
Extracts a phone number from a string representing a vCard.

Syntax

scan_phone(
    source STRING,
    type STRING )
RETURNS STRING

1. source is the vCard string (version 3.0).
2. type is the type of phone number (HOME, WORK, TEXT, VOICE, FAX, CELL, VIDEO, PAGER, TEXTPHONE, pref).

Usage
This function parses the vCard string passed as parameter to find phone data based on a type, and returns the phone number in a string.

The function looks for lines starting with the "TEL" keyword.

The second parameter (type) defines the value of the "TYPE" attribute in a "TELs" line. Values can, for example, be "HOME", "WORK", "TEXT", "VOICE", "FAX", "CELL", "VIDEO", "PAGER", "TEXTPHONE", "pref". If this parameter is NULL, the phone number with TYPE=pref will be returned. If no preferred phone number exists, the first phone number will be returned.

Example

IMPORT FGL VCard
MAIN
    DEFINE n STRING,
    f TEXT
    LOCATE f IN FILE arg_val(1)
    CALL VCard.scan_phone(f,NULL) RETURNING n
    DISPLAY n
END MAIN

fglgallery: Image gallery module

Table 495: fglgallery types (fglgallery.4gl)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUBLIC TYPE t_struct_value RECORD</td>
<td>The t_struct_value type holds image selection data.</td>
</tr>
<tr>
<td>current INTEGER,</td>
<td></td>
</tr>
<tr>
<td>selected DYNAMIC ARRAY OF INTEGER</td>
<td></td>
</tr>
<tr>
<td>END RECORD</td>
<td></td>
</tr>
</tbody>
</table>
Table 496: fglgallery functions (fglgallery.4gl)

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addImage(id SMALLINT, path STRING, title STRING)</td>
<td>Adds a picture resource to an fglgallery.</td>
</tr>
<tr>
<td>clean(id SMALLINT)</td>
<td>Removes all pictures from an fglgallery.</td>
</tr>
<tr>
<td>create(name STRING)</td>
<td>Creates a new fglgallery handle.</td>
</tr>
<tr>
<td>deleteImages(id SMALLINT, indexes DYNAMIC ARRAY OF INTEGER)</td>
<td>Deletes pictures used in an fglgallery.</td>
</tr>
<tr>
<td>destroy(id SMALLINT)</td>
<td>Frees resources allocated for an fglgallery.</td>
</tr>
<tr>
<td>display(id SMALLINT, type INTEGER, size INTEGER)</td>
<td>Displays an fglgallery to the end user.</td>
</tr>
<tr>
<td>finalize()</td>
<td>Releases the fglgallery library.</td>
</tr>
<tr>
<td>flush(id SMALLINT)</td>
<td>Displays new added images to the end user.</td>
</tr>
<tr>
<td>getImageCount(id SMALLINT)</td>
<td>Returns the number of pictures in an fglgallery.</td>
</tr>
<tr>
<td>getPath(id SMALLINT, index STRING)</td>
<td>Returns the URL of a picture in an fglgallery.</td>
</tr>
<tr>
<td>getTitle(id SMALLINT, index STRING)</td>
<td>Returns the description of a picture in an fglgallery.</td>
</tr>
<tr>
<td>initialize()</td>
<td>Prepares the fglgallery library for use.</td>
</tr>
<tr>
<td>setAspectRatio(id SMALLINT, ratio DECIMAL(5,2))</td>
<td>Prepares the fglgallery library for use.</td>
</tr>
<tr>
<td>setMultipleSelection(id SMALLINT)</td>
<td>Prepares the fglgallery library for use.</td>
</tr>
</tbody>
</table>
**t_struct_value type**
The *t_struct_value* type holds image selection data.

**Syntax**

```pl
PUBLIC TYPE t_struct_value RECORD
  current INTEGER,
  selected DYNAMIC ARRAY OF INTEGER
END RECORD
```

**Usage**

This user-defined type defines a record structure to hold image selection information, to be used in fglgallery web components.

Each picture is identified by its ordinal position in the image gallery.

- The *current* member defines the current picture in the gallery.
- The *selected* member is a dynamic array of integers, defining the selected images. This member is used, when multiple image selection is enabled with `setMultipleSelection()`. The current image can be different from a selected image. The current image may not be selected.

Image selection information is stored in the `WEBCOMPONENT` field value, as a JSON formatted string similar to: `{"current":5,"selected":[4,5]}`. The variable bound to the fglgallery `WEBCOMPONENT` form field must be defined as `VARCHAR` or `STRING`.

The *t_struct_value* structure is typically used to get the selected image ids, when the image selection action is fired by the fglgallery web component. The web component field value can be parsed with the `util.JSON.parse()` method, to fill a variable defined with the *t_struct_value* type.

The web component field value can also be used to control the image selection from the program code. To select specific images in the gallery, fill the *t_struct_value* variable with image ids, then stringify the structure to JSON with the `util.JSON.stringify()` method, and set the web component field value with this JSON string.

**Example**

In this code example, `rec.gallery_wc` is the name of the fglgallery web component field:

```pl
DEFINE struct_value fglgallery.t_struct_value
...
  ON ACTION image_selection
    CALL util.JSON.parse( rec.gallery_wc, struct_value )
    DISPLAY struct_value.current,
    struct_value.selected.getLength()
...
  LET struct_value.current = 3
  LET struct_value.selected.clear()
  LET struct_value.selected[1] = 2
  LET struct_value.selected[2] = 7
  LET struct_value.selected[3] = 15
  LET rec.gallery_wc = util.JSON.stringify(struct_value)
...
```

**addImage()**

Adds a picture resource to an fglgallery.

**Syntax**

```pl
addImage ( id SMALLINT, path STRING,
```
Example

CALL fglgallery.clean( id )

Related concepts
The fglgallery web component on page 1979
The fglgallery built-in web component implements a picture gallery, to let the end user select pictures and photos.

create()
Creates a new fglgallery handle.

Syntax

```sql
create( name STRING )
    RETURNS SMALLINT
```

1. `name` is the name of the WEBCOMPONENT form field.

Usage
This function creates a new fglgallery handler.

Note: The window/form containing the fglgallery web component field must be created before calling this function.

The id returned by this function identifies the web component field in subsequent fglgallery API calls.

When the fglgallery web component is no longer required, use the `destroy()` function, to free the resources allocated for this fglgallery object.

Example

```sql
DEFINE id SMALLINT
LET id = fglgallery.create("formonly.gallery")
...
CALL fglgallery.destroy(id)
```

Related concepts
The fglgallery web component on page 1979
The fglgallery built-in web component implements a picture gallery, to let the end user select pictures and photos.

deleteImages()
Deletes pictures used in an fglgallery.

Syntax

```sql
deleteImages(
    id SMALLINT,
    indexes DYNAMIC ARRAY OF INTEGER )
    RETURNS STRING
```

1. `id` is the fglgallery identifier, as returned by the `create()` function.
2. `indexes` is the dynamic array of indexes of the pictures to remove from the gallery.

Usage
This function deletes the specified pictures from the fglgallery.

The `id` parameter is the fglgallery handler returned by the `create()` function.
The `indexes` parameter is a `DYNAMIC ARRAY OF INTEGER` that defines a set of picture indexes. Each corresponding picture reference will be removed from the image gallery.

The new gallery is automatically displayed to the end user.

**Example**

```
DEFINE todel DYNAMIC ARRAY OF INTEGER
LET todel[1] = 3
LET todel[2] = 6
CALL fglgallery.deleteImages(id, todel)
```

**Related concepts**

The `fglgallery` web component on page 1979

The `fglgallery` built-in web component implements a picture gallery, to let the end user select pictures and photos.

**destroy()**

Frees resources allocated for an `fglgallery`.

**Syntax**

```
destroy( id SMALLINT )
```

1. `id` is the `fglgallery` identifier, as returned by the `create()` function.

**Usage**

This function releases the `fglgallery` identified by the specified handler.

The `id` parameter is the `fglgallery` handler returned by the `create()` function.

Call this method after closing the window/form displaying the gallery web component.

**Example**

```
CALL fglgallery.destroy( id )
```

**Related concepts**

The `fglgallery` web component on page 1979

The `fglgallery` built-in web component implements a picture gallery, to let the end user select pictures and photos.

**display()**

Displays an `fglgallery` to the end user.

**Syntax**

```
display(
    id SMALLINT,
    type INTEGER,
    size INTEGER )
```

1. `id` is the gallery identifier, as returned by the `create()` function.
2. `type` is the type of rendering to be used for the gallery (`FGLGALLERY_TYPE_*`).
3. `size` is the size hint for the gallery rendering (`FGLGALLERY_SIZE_*`).
**Usage**

This function displays the fglgallery in the corresponding web component field.

The *id* parameter is the fglgallery handler returned by the `create()` function.

The function can be called several times. It is typically used to refresh the image gallery, when the type or the size of the gallery needs to be changed.

The gallery *type* can be one of:
- `FGLGALLERY_TYPE_MOSAIC`
- `FGLGALLERY_TYPE_LIST`
- `FGLGALLERY_TYPE_THUMBNAILS`

The gallery *size* can be one of:
- `FGLGALLERY_SIZE_XSMALL`
- `FGLGALLERY_SIZE_SMALL`
- `FGLGALLERY_SIZE_NORMAL`
- `FGLGALLERY_SIZE_LARGE`
- `FGLGALLERY_SIZE_XLARGE`

**Note:** If you only need to add images to the gallery without changing the gallery type or size, there is no need to redisplay the gallery: Use the `addImage()` and `flush()` functions instead.

**Example**

```plaintext
CALL fglgallery.display( id, FGLGALLERY_TYPE_MOSAIC,
                        FGLGALLERY_SIZE_NORMAL )
```

**Related concepts**

The fglgallery web component on page 1979

The fglgallery built-in web component implements a picture gallery, to let the end user select pictures and photos.

**finalize()**

Releases the fglgallery library.

**Syntax**

```plaintext
finalize( )
```

**Usage**

This function terminates the fglgallery module usage.

It must be performed when the fglgallery library is no longer needed by the caller.

**Related concepts**

The fglgallery web component on page 1979

The fglgallery built-in web component implements a picture gallery, to let the end user select pictures and photos.

**flush()**

Displays new added images to the end user.

**Syntax**

```plaintext
flush( id SMALLINT )
```
1. *id* is the *fglgallery* identifier, as returned by the *create()* function.

**Usage**

This function flushes the new images to the *fglgallery* web component. The *flush()* method is used after adding a set of new images with the *addImage()* function, to show the images when the image gallery is already displayed. A *flush()* call is not required before calling the *display()* method.

The *id* parameter is the *fglgallery* handler returned by the *create()* function.

**Example**

```plaintext
CALL fglgallery.addImage(id, "http://...", "Picture 1")
CALL fglgallery.addImage(id, "http://...", "Picture 2")
CALL fglgallery.addImage(id, "http://...", "Picture 3")
...
CALL fglgallery.flush()
```

**Related concepts**

The *fglgallery* web component on page 1979

The *fglgallery* built-in web component implements a picture gallery, to let the end user select pictures and photos.

**getImageCount()**

Returns the number of pictures in an *fglgallery*.

**Syntax**

```plaintext
getImageCount ( id SMALLINT )
RETURNS INTEGER
```

1. *id* is the *fglgallery* identifier, as returned by the *create()* function.

**Usage**

This function counts the number of pictures in the specified *fglgallery* web component.

The *id* parameter is the *fglgallery* handler returned by the *create()* function.

In an *fglgallery* web component, pictures are identified by their ordinal position. This function can be used to loop through the picture indexes.

**Example**

```plaintext
FOR idx = 1 TO fglgallery.getImageCount(id)
    DISPLAY fglgallery.getPath(id, idx)
END FOR
```

**Related concepts**

The *fglgallery* web component on page 1979

The *fglgallery* built-in web component implements a picture gallery, to let the end user select pictures and photos.

**getPath()**

Returns the URL of a picture in an *fglgallery*.

**Syntax**

```plaintext
getPath( 
```
### getPath

**Syntax**

```sql
getPath ( id SMALLINT, index STRING ) RETURNS STRING
```

1. `id` is the fglgallery identifier, as returned by the `create()` function.
2. `index` is the picture index in the image gallery.

**Usage**

This function returns the path (URL) of a picture in an image gallery. The value returned by this function corresponds to the path provided as second parameter to the `addImage()` function.

The `id` parameter is the fglgallery handler returned by the `create()` function.

The `index` parameter identifies the picture by its ordinal position in the image gallery.

**Example**

```javascript
DISPLAY fglgallery.getPath( id, 10 )
```

**Related concepts**

The fglgallery web component on page 1979

The `fglgallery` built-in web component implements a picture gallery, to let the end user select pictures and photos.

### getTitle

**Syntax**

```sql
getTitle ( id SMALLINT, index STRING ) RETURNS STRING
```

1. `id` is the fglgallery identifier, as returned by the `create()` function.
2. `index` is the picture index in the image gallery.

**Usage**

This function returns the title of a picture in the fglgallery.

The `id` parameter is the fglgallery handler returned by the `create()` function.

The `index` parameter identifies the picture by its ordinal position in the image gallery.

**Example**

```javascript
DISPLAY fglgallery.getTitle( id, 10 )
```

**Related concepts**

The fglgallery web component on page 1979
The fglgallery built-in web component implements a picture gallery, to let the end user select pictures and photos.

initialize()
Prepares the fglgallery library for use.

Syntax

```
initialize()
```

Usage

This function initializes the fglgallery module usage. When the fglgallery library is no longer needed, call the finalization function `fglgallery.finalize()`.

Note: Initialization and finalization functions can be called several times by different modules using the fglgallery library.

Example

```
IMPORT FGL fglgallery
FUNCTION show_image_gallery()
  ...
  CALL fglgallery.initialize()
  ...
  CALL fglgallery.finalize()
END FUNCTION
```

Related concepts

The fglgallery web component on page 1979
The fglgallery built-in web component implements a picture gallery, to let the end user select pictures and photos.

setMultipleSelection()

Enables/disables multiple picture selection in an fglgallery.

Syntax

```
setMultipleSelection(
  id SMALLINT,
  on BOOLEAN
)
```

1. *id* is the fglgallery identifier, as returned by the `create()` function.
2. *on* can be `TRUE` (to enable multi-selection) or `FALSE` (to disable multi-selection).

Usage

This function controls multiple picture selection in the specified image gallery.

The *id* parameter is the fglgallery handler returned by the `create()` function.

When multiple selection is enabled, the end user can mark pictures as selected.

Indexes of selected images are provided in the web component field value, using the following JSON notation: `{"current":5,"selected":[4,5]}`. Convert this JSON string to a `fglgallery.t_struct_value` record to get the list of picture indexes in the "selected" member.
Example

CALL fglgallery.setMultipleSelection( id, TRUE )

Related concepts

The fglgallery web component on page 1979
The fglgallery built-in web component implements a picture gallery, to let the end user select pictures and photos.

setAspectRatio()
Displays an fglgallery to the end user.

Syntax

```
setAspectRatio(
  id SMALLINT,
  ratio DECIMAL(5,2) )
```

1. *id* is the gallery identifier, as returned by the *create()* function.
2. *ratio* is the aspect ratio to be used (for example, 1.77 or 16/9).

Usage

In order to align image elements properly when image resources have different sizes, the fglgallery uses the same size for each image.

**Note:** The size of image elements is defined by the FGLGALLERY_SIZE_* parameter passed to the *display()* function. This size parameter is relative to the current font size (em unit is used in HTML).

The *setAspectRatio()* function defines the aspect ratio for fglgallery image elements.

By default, a square aspect ration (1:1) is used.

The *id* parameter is the fglgallery handler returned by the *create()* function.

The *display()* function has to used after *setAspectRatio()* to have the changes in the fglgallery take effect.

Example

CALL fglgallery.setAspectRatio( id, 16/9 )

Related concepts

The fglgallery web component on page 1979
The fglgallery built-in web component implements a picture gallery, to let the end user select pictures and photos.

fglsvgcanvas: SVG drawing module

Table 497: fglsvgcanvas types (fglsvgcanvas.4gl)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUBLIC TYPE t_svg_rect RECORD</td>
<td></td>
</tr>
<tr>
<td>x DECIMAL,</td>
<td></td>
</tr>
<tr>
<td>y DECIMAL,</td>
<td></td>
</tr>
<tr>
<td>width DECIMAL,</td>
<td></td>
</tr>
<tr>
<td>height DECIMAL</td>
<td></td>
</tr>
</tbody>
</table>
END RECORD | The t_svg_rect type defines the position and dimensions of a rectangle. |
Table 498: fglsvgcanvas constants (fglsvgcanvas.4gl)

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVGATT_* constants</td>
<td>List of predefined SVG attributes.</td>
</tr>
</tbody>
</table>
Table 499: fglsvgcanvas functions (fglsvgcanvas.4gl)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>animateTransform(</td>
<td>Produces an SVG &quot;animateTransform&quot; element.</td>
</tr>
<tr>
<td>attributeName</td>
<td></td>
</tr>
<tr>
<td>attributeType</td>
<td></td>
</tr>
<tr>
<td>type</td>
<td></td>
</tr>
<tr>
<td>from</td>
<td></td>
</tr>
<tr>
<td>to</td>
<td></td>
</tr>
<tr>
<td>by</td>
<td></td>
</tr>
<tr>
<td>begin</td>
<td></td>
</tr>
<tr>
<td>dur</td>
<td></td>
</tr>
<tr>
<td>repeatCount</td>
<td></td>
</tr>
<tr>
<td>RETURNS om.DomNode</td>
<td></td>
</tr>
<tr>
<td>circle(</td>
<td>Produces an SVG &quot;circle&quot; element.</td>
</tr>
<tr>
<td>cx</td>
<td></td>
</tr>
<tr>
<td>cy</td>
<td></td>
</tr>
<tr>
<td>r</td>
<td></td>
</tr>
<tr>
<td>RETURNS om.DomNode</td>
<td></td>
</tr>
<tr>
<td>clean( cid SMALLINT)</td>
<td>Delete all SVG elements inside the SVG canvas.</td>
</tr>
<tr>
<td>clipPath_rect(</td>
<td>Produces an SVG &quot;clipPath&quot; element with a &quot;rect&quot; element.</td>
</tr>
<tr>
<td>id</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td></td>
</tr>
<tr>
<td>width</td>
<td></td>
</tr>
<tr>
<td>height</td>
<td></td>
</tr>
<tr>
<td>RETURNS om.DomNode</td>
<td></td>
</tr>
<tr>
<td>color_shade(</td>
<td>Applies a shade factor to an RGB color.</td>
</tr>
<tr>
<td>source</td>
<td></td>
</tr>
<tr>
<td>factor</td>
<td></td>
</tr>
<tr>
<td>RETURNS STRING</td>
<td></td>
</tr>
<tr>
<td>color_tint(</td>
<td>Applies a tint factor to an RGB color.</td>
</tr>
<tr>
<td>source</td>
<td></td>
</tr>
<tr>
<td>factor</td>
<td></td>
</tr>
<tr>
<td>RETURNS STRING</td>
<td></td>
</tr>
<tr>
<td>create( name STRING)</td>
<td>Creates a new SVG canvas handler.</td>
</tr>
<tr>
<td>RETURNS SMALLINT</td>
<td></td>
</tr>
<tr>
<td>createChars(</td>
<td>Produces an SVG DOM text node.</td>
</tr>
<tr>
<td>value</td>
<td></td>
</tr>
<tr>
<td>RETURNS om.DomNode</td>
<td></td>
</tr>
<tr>
<td>createElement(</td>
<td>Produces an SVG DOM element with the tag name specified as parameter.</td>
</tr>
<tr>
<td>tagName</td>
<td></td>
</tr>
<tr>
<td>id</td>
<td></td>
</tr>
<tr>
<td>RETURNS om.DomNode</td>
<td></td>
</tr>
<tr>
<td>defs( id STRING )</td>
<td>Produces an SVG &quot;defs&quot; element.</td>
</tr>
<tr>
<td>RETURNS om.DomNode</td>
<td></td>
</tr>
</tbody>
</table>
t_svg_rect type
The t_svg_rect type defines the position and dimensions of a rectangle.

Syntax

PUBLIC TYPE t_svg_rect RECORD
  x DECIMAL,
  y DECIMAL,
  width DECIMAL,
  height DECIMAL
END RECORD

Usage
This user-defined type defines a record structure with the position (x, y) and size (width, height) of a rectangle.
The structure is for example used with the getBBox() function.

SVGATT_ constants
List of predefined SVG attributes.

Syntax

PUBLIC CONSTANT SVGATT_TRANSFORM = "transform"
PUBLIC CONSTANT SVGATT_CLASS = "class"
PUBLIC CONSTANT SVGATT_STYLE = "style"
PUBLIC CONSTANT SVGATT_ONCLICK = "onclick"
PUBLIC CONSTANT SVGATT_ONMOUSEOVER = "onmouseover"
PUBLIC CONSTANT SVGATT_ONMOUSEOUT = "onmouseout"
PUBLIC CONSTANT SVGVAL_ELEM_CLICKED = "elem_clicked(this)"
PUBLIC CONSTANT SVGVAL_ELEM_MOUSE_OVER = "elem_mouse_over(this)"
PUBLIC CONSTANT SVGVAL_ELEM_MOUSE_OUT = "elem_mouse_out(this)"
PUBLIC CONSTANT SVGATT_FILL = "fill"
PUBLIC CONSTANT SVGATT_FILL_OPACITY = "fill-opacity"
PUBLIC CONSTANT SVGATT_FILL_RULE = "fill-rule"
PUBLIC CONSTANT SVGATT_STROKE = "stroke"
PUBLIC CONSTANT SVGATT_STROKE_WIDTH = "stroke-width"
PUBLIC CONSTANT SVGATT_STROKE_OPACITY = "stroke-opacity"
PUBLIC CONSTANT SVGATT_STROKE_LINECAP = "stroke-linecap"
PUBLIC CONSTANT SVGATT_STROKE_LINEJOIN = "stroke-linejoin"
PUBLIC CONSTANT SVGATT_STROKE_MITERLIMIT = "stroke-miterlimit"
PUBLIC CONSTANT SVGATT_STROKE_DASHARRAY = "stroke-dasharray"
PUBLIC CONSTANT SVGATT_STROKE_DASHOFFSET = "stroke-dashoffset"
PUBLIC CONSTANT SVGATT_FONT_FAMILY = "font-family"
PUBLIC CONSTANT SVGATT_FONT_SIZE = "font-size"
PUBLIC CONSTANT SVGATT_FONT_SIZE_ADJUST = "font-size-adjust"
PUBLIC CONSTANT SVGATT_FONT_STRETCH = "font-stretch"
PUBLIC CONSTANT SVGATT_FONT_STYLE = "font-style"
PUBLIC CONSTANT SVGATT_FONT_VARIANT = "font-variant"
PUBLIC CONSTANT SVGATT_FONT_WEIGHT = "font-weight"
PUBLIC CONSTANT SVGATT_TEXT_ANCHOR = "text-anchor"
PUBLIC CONSTANT SVGATT_MARKER_START = "marker-start"
PUBLIC CONSTANT SVGATT_MARKER_MID = "marker-mid"
PUBLIC CONSTANT SVGATT_MARKER_END = "marker-end"
Usage
The SVGATT pre-defined constants are available from the fglsvgcanvas module. Such constants are typically used to build an SVG attribute list for `setAttributes()` or `styleAttributeList()`.

animateTransform()
Produces an SVG "animateTransform" element.

Syntax
```
animateTransform(  
    attributeName STRING,  
    attributeType STRING,  
    type STRING,  
    from STRING,  
    to STRING,  
    by STRING,  
    begin STRING,  
    dur STRING,  
    repeatCount STRING )  
RETURNS om.DomNode
```

1. The `attributeName`, `attributeType`, `type`, `from`, `to`, `by`, `begin`, `dur`, and `repeatCount` parameters are used to set the corresponding SVG attributes for the `animateTransform` element. See SVG specification for details.

Usage
This function creates an "animateTransform" SVG DOM element from the parameters passed.

For more details about SVG animation, refer to the SVG specification.

Example
```
DEFINE n om.DomNode
LET n = fglsvgcanvas.animateTransform("transform", "XML",  
    "rotate", NULL, NULL,  
    "360", NULL, "12h", "indefinite")
```

Related concepts
The fglsvgcanvas web component on page 1988
The fglsvgcanvas built-in web component implements a drawing canvas for Scalable Vector Graphics content.

circle()
Produces an SVG "circle" element.

Syntax
```
circle(  
   cx DECIMAL,  
   cy DECIMAL,  
   r DECIMAL )  
RETURNS om.DomNode
```

1. `cx` defines the X coordinate of the center point of the circle.
2. `cy` defines the Y coordinate of the center point of the circle.
3. `r` defines the radius of the circle.
Usage
This function creates a "circle" SVG DOM element from the parameters.

Example

```om
DEFINE n om.DomNode
LET n = fglsvgcanvas.circle(100,100,50)
CALL n.setAttribute(SVGATT_STYLE,"stroke:#006600;")
```

Related concepts
The fglsvgcanvas web component on page 1988
The fglsvgcanvas built-in web component implements a drawing canvas for Scalable Vector Graphics content.

clean()
Deletes all SVG elements inside the SVG canvas.

Syntax

```om
clean( cid SMALLINT )
```

1. `cid` is the SVG canvas id, as returned by `fglsvgcanvas.create()`.

Usage
This function cleans the SVG canvas.

Note: The clean() function will not automatically display the cleaned SVG canvas. To see a visual result, re-display the SVG content with the `display()` function.

Related concepts
The fglsvgcanvas web component on page 1988
The fglsvgcanvas built-in web component implements a drawing canvas for Scalable Vector Graphics content.

clipPath_rect()
Produces an SVG "clipPath" element with a "rect" element.

Syntax

```om
clipPath_rect( 
    id STRING,
    x DECIMAL,
    y DECIMAL,
    width DECIMAL,
    height DECIMAL 
) 
RETURNS om.DomNode
```

1. `id` is the SVG object identifier.
2. `x` is the X coordinate of the top-left corner of the clip rectangle.
3. `y` is the Y coordinate of the top-left corner of the clip rectangle.
4. `width` is the width of the clip rectangle.
5. `height` is the height of the clip rectangle.

Usage
This function creates a "clipPath" SVG DOM element using a rectangle as clipping shape.

A clip path can be used in a text or image element, to define the clipping borders.

The clip path element must be added to the `defs` node of the `svg` element.
To use a clip path in an SVG DOM element, set the "clip-path" attribute with the string generated by the `url()` function.

**Example**

```plaintext
DEFINE defs om.DomNode,
    clip_id STRING
...
LET clip_id = "clip_gh_1"
CALL defs.appendChild(
    fglsvgcanvas.clipPath_rect( clip_id, tx, ty, -1, sy )
)
...
LET n = fglsvgcanvas.text( tx + 3, (ty + (sy/2) + 1), "Some text", NULL )
CALL n.setAttribute("clip-path", fglsvgcanvas.url("clip_gh_1") )
```

**Related concepts**
The [fglsvgcanvas web component](#) on page 1988
The [fglsvgcanvas](#) built-in web component implements a drawing canvas for Scalable Vector Graphics content.

color_shade()
Applies a shade factor to an RGB color.

**Syntax**

```plaintext
color_shade(
    source STRING,
    factor DECIMAL )
RETURNS STRING
```

1. `source` is the source RGB color.
2. `factor` is the shade factor.

**Usage**
This function modifies the RGB color specification by applying the shade factor passed as parameter.

**Example**

```plaintext
LET m = fglsvgcanvas.color_shade( "#FFAA34", 0.42 )
```

**Related concepts**
The [fglsvgcanvas web component](#) on page 1988
The [fglsvgcanvas](#) built-in web component implements a drawing canvas for Scalable Vector Graphics content.

color_tint()
Applies a tint factor to an RGB color.

**Syntax**

```plaintext
color_tint(
    source STRING,
    factor DECIMAL )
RETURNS STRING
```

1. `source` is the source RGB color.
2. `factor` is the tint factor.
Usage
This function modifies the RGB color specification by applying the tint factor passed as parameter.

Example

| LET m = fglsvgcanvas.color_tint( "#FFAA34", 0.42 ) |

Related concepts
The fglsvgcanvas web component on page 1988
The fglsvgcanvas built-in web component implements a drawing canvas for Scalable Vector Graphics content.

destroy()
Creates a new SVG canvas handler.

Syntax

create( name STRING )
RETURNS SMALLINT

1. name is the name of the form field defined as a WEBCOMPONENT.

Usage
This function creates a new SVG canvas handle by using the form field name passed as parameter.

Note: The window/form containing the fglsvgcanvas web component field must be created before calling this function.

The name is used to bind the WEBCOMPONENT form field with the SVG canvas, to display SVG content.

The function returns the id of the newly-created SVG canvas. This id must be used in subsequent calls to fglsvgcanvas functions.

When the SVG canvas is no longer needed, free the allocated resources with the destroy() function.

Example

Form file:

```plaintext
... ATTRIBUTES
WEBCOMPONENT cv = FORMONLY.canvas,
    COMPONENTTYPE = "fglsvgcanvas",
    PROPERTIES = (selection="item_selection"),
    SIZEPOLICY = FIXED,
    STRETCH = BOTH,
    SCROLLBARS = NONE
;
...
```

Program code:

```plaintext
DEFINE cid SMALLINT
LET cid = fglsvgcanvas.create("formonly.canvas")
...
CALL fglsvgcanvas.destroy( cid )
```

Related concepts
The fglsvgcanvas web component on page 1988
The fglsvgcanvas built-in web component implements a drawing canvas for Scalable Vector Graphics content.

createChars()
Produce an SVG DOM text node.

Syntax

```
createChars ( value STRING )
RETURNS om.DomNode
```

1. `value` is the value to be set in the text node.

Usage

This function creates an SVG DOM text node from the value passed as parameter.

Use the createChars() function when you need to create an SVG text node that cannot be created with one of the fglsvgcanvas functions (such as `title()`).

Example

```
DEFINE t, c om.DomNode
LET t = fglsvgcanvas.createElement( "title", "id1" )
LET c = fglsvgcanvas.createChars( "my value" )
CALL t.appendChild( c )
```

Related concepts

The fglsvgcanvas web component on page 1988

The fglsvgcanvas built-in web component implements a drawing canvas for Scalable Vector Graphics content.

createElement()
Produce an SVG DOM element with the tag name specified as parameter.

Syntax

```
createElement ( 
  tagName STRING,
  id STRING )
RETURNS om.DomNode
```

1. `tagName` is the DOM tag name.
2. `id` is the SVG object identifier.

Usage

This function creates an SVG DOM element with the specified tag name.

Use the createElement() function when you need to create an SVG element that cannot be created with one of the fglsvgcanvas functions.

Example

```
DEFINE n om.DomNode
LET n = fglsvgcanvas.createElement( "feBlend", "myblend" )
```

Related concepts

The fglsvgcanvas web component on page 1988
The `fglsvgcanvas` built-in web component implements a drawing canvas for Scalable Vector Graphics content.

**defs()**

Produces an SVG "defs" element.

**Syntax**

```
defs(
    id STRING )
RETURNS om.DomNode
```

1. *id* is the SVG object identifier.

**Usage**

This function creates a "defs" SVG DOM element from the parameters.

The `defs` element can be used to centralize SVG drawing elements such as SVG patterns or CSS styles, that can be reused in other SVG elements.

**Example**

Creating a pattern:

```
DEFINE root_svg, defs, pattern, n om.DomNode
...
LET defs = fglsvgcanvas.defs( NULL )
CALL root_svg.appendChild( defs )

LET pattern = fglsvgcanvas.pattern( "pattern1", 0, 0, 20, 20,
    "userSpaceOnUse",
    NULL, "rotate(45)" ), NULL )
CALL pattern.appendChild( n:=fglsvgcanvas.rect(0,0,1000,10,NULL,NULL) )
CALL n.setAttribute(SVGATT_STYLE, 'stroke:none; fill:blue;' )
CALL pattern.appendChild( n:=fglsvgcanvas.rect(0,10,1000,10,NULL,NULL) )
CALL n.setAttribute(SVGATT_STYLE, 'stroke:none; fill:navy;' )
CALL defs.appendChild( pattern )
```

Creating CSS styles:

```
CONSTANT STYLE_1 = 1
CONSTANT STYLE_2 = 2
CONSTANT STYLE_3 = 3
DEFINE attr DYNAMIC ARRAY OF om.SaxAttributes
DEFINE root_svg, defs om.DomNode
...
LET defs = fglsvgcanvas.defs( NULL )
CALL defs.appendChild( fglsvgcanvas.styleList(
    fglsvgcanvas.styleDefinition(".style_1",attr[STYLE_1])
    ||
    fglsvgcanvas.styleDefinition(".style_2",attr[STYLE_2])
    ||
    fglsvgcanvas.styleDefinition(".style_3",attr[STYLE_3])
) )
CALL root_svg.appendChild( defs )
```

**Related concepts**

The `fglsvgcanvas` web component on page 1988
The `fglsvgcanvas` built-in web component implements a drawing canvas for Scalable Vector Graphics content.

### destroy()
Releases resources allocated for the SVG canvas.

**Syntax**

```python
destroy( cid SMALLINT )
```

1. `cid` is the SVG canvas id, as returned by `fglsvgcanvas.create()`.

**Usage**
This function frees resources allocated for the SVG canvas handler that was created with the `create()` function.

**Example**

```sql
CALL fglsvgcanvas.destroy( cid )
```

**Related concepts**
The `fglsvgcanvas web component` on page 1988

The `fglsvgcanvas` built-in web component implements a drawing canvas for Scalable Vector Graphics content.

### display()
Displays the SVG canvas.

**Syntax**

```python
display( cid SMALLINT )
```

1. `cid` is the SVG canvas id, as returned by `fglsvgcanvas.create()`.

**Usage**
The function sends the SVG content to the front-end for display.

**Example**

```sql
CALL fglsvgcanvas.display( cid )
```

**Related concepts**
The `fglsvgcanvas web component` on page 1988

The `fglsvgcanvas` built-in web component implements a drawing canvas for Scalable Vector Graphics content.

### ellipse()
Produces an SVG "ellipse" element.

**Syntax**

```python
ellipse(
    cx DECIMAL,
    cy DECIMAL,
    rx DECIMAL,
    ry DECIMAL
) RETURNS om.DomNode
```

1. `cx` defines the X coordinate of the center of the ellipse.
2. `cy` defines the Y coordinate of the center of the ellipse.
3. \( rx \) defines the X radius of the ellipse.
4. \( ry \) defines the Y radius of the ellipse.

Usage
This function creates an "ellipse" SVG DOM element from the parameters.

Example

```om
DEFINE n om.DomNode
LET n = fglsvgcanvas.ellipse(100,100,50,60)
CALL n.setAttribute(SVGATT_STYLE,"stroke:#006600;")
```

Related concepts
The `fglsvgcanvas` web component on page 1988
The `fglsvgcanvas` built-in web component implements a drawing canvas for Scalable Vector Graphics content.

filter()
Produces the SVG "filter" element.

Syntax

```om
filter(
  id STRING,
  x STRING,
  y STRING,
  width STRING,
  height STRING )
RETURNS om.DomNode
```

1. \( id \) is the SVG object identifier.
2. \( x \) The X for the filter dimension.
3. \( y \) The Y for the filter dimension.
4. \( width \) The width for the filter dimension.
5. \( height \) The height for the filter dimension.

Usage
This function creates a "filter" SVG DOM element from the parameters.

Filters must be defined in a "defs" element, that is created with the `defs()` function.

Create filter effect sub-elements such as "feOffset" or "feGaussianBlur" with the `createElement()` function.

A filter can then be applied to an SVG element, by setting the "filter" attribute with an "url(#filter-name)" reference.

Example

```om
DEFINE root_svg, defs, f, fe, fse, n om.DomNode
LET defs = fglsvgcanvas.defs( NULL )
CALL root_svg.appendChild( defs )
CALL defs.appendChild( f:=fglsvgcanvas.filter("blur1", -10, -10, 40, 150 ) )
CALL f.appendChild( fe:=fglsvgcanvas.createElement("feOffset",NULL) )
CALL fe.setAttribute("in","SourceAlpha")
CALL fe.setAttribute("dx","3")
CALL fe.setAttribute("dy","3")
```
CALL `fe.setAttribute("result","offset2")`

CALL `f.appendChild( fe:=fglsvgcanvas.createElement("feGaussianBlur",NULL) )`
CALL `fe.setAttribute("in","offset2")`
CALL `fe.setAttribute("stdDeviation","3")`
CALL `fe.setAttribute("result","blur2")`

CALL `f.appendChild( fe:=fglsvgcanvas.createElement("feMerge",NULL) )`
CALL `fe.appendChild( fse:=fglsvgcanvas.createElement("feMergeNode",NULL) )`
CALL `fse.setAttribute("in","blur2")`
CALL `fe.appendChild( fse:=fglsvgcanvas.createElement("feMergeNode",NULL) )`
CALL `fse.setAttribute("in","SourceGraphic")`

... CALL `root_svg.appendChild( n:=fglsvgcanvas.rect(40,50,100,200,5,5) )`
CALL `n.setAttribute("style","stroke:gray; fill:blue; filter:url(#blur1);" )`

**Related concepts**

The fglsvgcanvas web component on page 1988
The fglsvgcanvas built-in web component implements a drawing canvas for Scalable Vector Graphics content.

**finalize()**

Releases the fglsvgcanvas library.

**Syntax**

```javascript
finalize( )
```

**Usage**

This function terminates the fglsvgcanvas module usage.
It must be performed when the fglsvgcanvas library is no longer needed by the caller.

**Related concepts**

The fglsvgcanvas web component on page 1988
The fglsvgcanvas built-in web component implements a drawing canvas for Scalable Vector Graphics content.

**g()**

Produces an SVG "g" element.

**Syntax**

```javascript
g(  
   id STRING  
)
RETURNS om.DomNode
```

1. `id` is the SVG object identifier.

**Usage**

This function creates a "g" SVG DOM element from the parameters.
The `g` element is used to group SVG elements together, apply a transformation, and use common attributes for the group.

For example, you can draw complex shapes inside the "g" element, and rotate the whole object by 180 degrees around point (50,50), by using "rotate(180 50 50)" for the "transform" parameter. For more details about the "transform" SVG attribute, see SVG specification.
Example

```
DEFINE n om DomNode
LET n = fglsvgcanvas.g( "group1" )
CALL n.setAttribute(SVGATT_TRANSFORM, "rotate(45 50 50)")
```

Related concepts

The fglsvgcanvas web component on page 1988
The fglsvgcanvas built-in web component implements a drawing canvas for Scalable Vector Graphics content.

**getBBox()**

Returns the bounding box of an SVG element.

Syntax

```
getBBox( cid SMALLINT, id STRING )
RETURNS t_svg_rect
```

1. `cid` is the SVG canvas id, as returned by fglsvgcanvas.create().
2. `id` identifies the SVG element from which the bounding box is got.

Usage

This function returns the bounding box of the SVG element identified by the `id` parameter.

The values returned are `x`, `y`, `width` and `height` (in the current user space).

The values returned by `getBBox()` can be held in a record variable defined with the `t_svg_rect` type.

If no SVG element is found, the values returned will be NULL.

Example

```
DEFINE rect fglsvgcanvas.t_svg_rect
...
ON ACTION get_bbox
  CALL fglsvgcanvas.getBBox(cid, "label_23") RETURNING rect.*
  DISPLAY rect.x, rect.y, rect.width, rect.height
```

Related concepts

The fglsvgcanvas web component on page 1988
The fglsvgcanvas built-in web component implements a drawing canvas for Scalable Vector Graphics content.

**getItemid()**

Returns SVG element id after a user action.

Syntax

```
getItemId( cid SMALLINT )
RETURNS STRING
```

1. `cid` is the SVG canvas id, as returned by fglsvgcanvas.create().

Usage

This function returns the `id` attribute of the SVG element involved in a user-defined action.

When the fglsvgcanvas web component does not have the focus, it is not possible to return the SVG element id in the field value. If user-defined actions are bound to SVG events that can be fired when the field does not have the focus, the only way to identify the SVG element is the `getItemId()` function.
Example

```plaintext
DEFINE id STRING
... 
ON ACTION mouse_over
    LET id = fglsvgcanvas.getItemId(cid)
```

Related concepts

The fglsvgcanvas web component on page 1988
The fglsvgcanvas built-in web component implements a drawing canvas for Scalable Vector Graphics content.

**image()**

Produces an SVG "image" element.

**Syntax**

```plaintext
image(
    href STRING,
    x DECIMAL,
    y DECIMAL,
    width DECIMAL,
    height DECIMAL,
    preserveAspectRatio STRING )
RETURNS om.DomNode
```

1. *href* defines the xlink:href reference to the image.
2. *x* defines the X coordinate of the image.
3. *y* defines the Y coordinate of the image.
4. *width* defines the width of the image.
5. *height* defines the height of the image.
6. *preserveAspectRatio* defines the aspect ratio to preserve.

**Usage**

This function creates an "image" SVG DOM element from the parameters.

The *href* parameter defines the image resource or URL.

**Important:** Use the `ui.Interface.filenameToURI()` method to produce an URL for application images.

**Example**

```plaintext
DEFINE n om.DomNode
LET n = fglsvgcanvas.image( ui.Interface.filenameToURI("image02.jpg"), 100,100,50,50, "xMidYMid meet" )
```

**Related concepts**

The fglsvgcanvas web component on page 1988
The fglsvgcanvas built-in web component implements a drawing canvas for Scalable Vector Graphics content.

**initialize()**

Prepares the fglsvgcanvas library for use.

**Syntax**

```plaintext
initialize( )
```
Usage

This function initializes the fglsvgcanvas module usage. When the fglsvgcanvas library is no longer needed, call the finalization function `fglsvgcanvas.finalize()`.

Note: Initialization and finalization functions can be called several times by different modules using the fglsvgcanvas library.

Example

```snippet
IMPORT FGL fglsvgcanvas
FUNCTION show_svg_content()
    ...
    CALL fglsvgcanvas.initialize()
    ...
    CALL fglsvgcanvas.finalize()
END FUNCTION
```

Related concepts

The fglsvgcanvas web component on page 1988
The fglsvgcanvas built-in web component implements a drawing canvas for Scalable Vector Graphics content.

line()

Produces an SVG "line" element.

Syntax

```snippet
def line ( x1 DECIMAL, y1 DECIMAL, x2 DECIMAL, y2 DECIMAL )
RETURNS om.DomNode
```

1. `x1` defines the X coordinate of the start point.
2. `y1` defines the Y coordinate of the start point.
3. `x2` defines the X coordinate of the end point.
4. `y2` defines the Y coordinate of the end point.

Usage

This function creates a "line" SVG DOM element from the parameters.

Example

```snippet
DEFINE n om.DomNode
LET n = fglsvgcanvas.line(10,10,50,30)
CALL n.setAttribute(SVGATT_STYLE,"stroke:#006600;")
```

Related concepts

The fglsvgcanvas web component on page 1988
The fglsvgcanvas built-in web component implements a drawing canvas for Scalable Vector Graphics content.

linearGradient()

Produces an SVG "linearGradient" element.

Syntax

```snippet
def linearGradient ( id STRING,
```

The fglsvgcanvas web component on page 1988
The fglsvgcanvas built-in web component implements a drawing canvas for Scalable Vector Graphics content.

linearGradient()

Produces an SVG "linearGradient" element.
x1 STRING,
y1 STRING,
x2 STRING,
y2 STRING,
spreadMethod STRING,
gradientTransform STRING,
gradientUnits STRING)
RETURNS om.DomNode

1. \( id \) is the SVG object identifier.
2. \( x1 \) defines the X coordinate of the linear gradient start point.
3. \( y1 \) defines the Y coordinate of the linear gradient start point.
4. \( x2 \) defines the X coordinate of the linear gradient end point.
5. \( y2 \) defines the Y coordinate of the linear gradient end point.
6. \( spreadMethod \) defines how the gradient is spread out through the shape.
7. \( gradientTransform \) defines the gradient transformation.
8. \( gradientUnits \) defines the coordinate system to be used.

Usage

This function creates a "linearGradient" SVG DOM element from the parameters.

An SVG "linearGradient" element can be used to define fill colors of shapes.

The element must contain "stop" elements that can be created with the \texttt{stop} function.

The resulting gradient definition can be added to a "defs" SVG DOM element.

Example

```om
DEFINE root_svg, defs, lg om.DomNode
...
LET defs = fglsvgcanvas.defs( NULL )
CALL root_svg.appendChild( defs )

LET lg = fglsvgcanvas.linearGradient( "gradient_1",
    "0\%", "0\%", "0\%", "100\%",
    "pad", NULL, NULL )
CALL lg.appendChild( fglsvgcanvas.stop( "0\%", "gray", 0.8 ) )
CALL lg.appendChild( fglsvgcanvas.stop( "100\%", "navy", 0.6 ) )
CALL defs.appendChild( lg )
...
```

Related concepts

\textbf{The fglsvgcanvas web component} on page 1988

The \texttt{fglsvgcanvas} built-in web component implements a drawing canvas for Scalable Vector Graphics content.

\textbf{marker()}

Produces an SVG "marker" element.

Syntax

```om
marker(
    id STRING,
    markerUnits STRING,
    refX DECIMAL,
    refY DECIMAL,
    markerWidth DECIMAL,
    markerHeight DECIMAL,
    orient STRING )
```
RETURNS om.DomNode

1. id is the SVG object identifier.
2. markerUnits defines how the marker scales ("strokeWidth" or "userSpaceOnUse").
3. refX defines the X coordinate of the reference point.
4. refY defines the Y coordinate of the reference point.
5. markerWidth defines the width of the marker.
6. markerHeight defines the height of the marker.
7. orient defines the orientation of the marker (like "auto").

Usage

This function creates a "marker" SVG DOM element from the parameters.

SVG markers define the rendering of start, mid and end of a line or path.

Markers are typically defined in a "defs" SVG element, and then used in a style definition with the marker-start, marker-mid and marker-end attributes, using the styleList() and styleDefinition() functions (see example below).

Example

DEFINE root_svg, defs, m, p, n om.DomNode
DEFINE attr DYNAMIC ARRAY OF om.SaxAttributes
...
LETdefs = fglsvgcanvas.defs( NULL )
CALL root_svg.appendChild( defs )

LET m = fglsvgcanvas.marker("m1", NULL, 5,5,10,10, "auto")
CALL defs.appendChild( m )
CALL m.appendChild( n:=fglsvgcanvas.circle(5,5,3) )
CALL n.setAttribute(SVGATT_STYLE,'stroke:gray; fill:blue;')

LET m = fglsvgcanvas.marker("m2", NULL, 2,6,15,15, "auto")
CALL defs.appendChild( m )
CALL m.appendChild( n:=fglsvgcanvas.path("M2,2 L2,11 L10,6 L2,2") )
CALL n.setAttribute(SVGATT_STYLE,'stroke:gray; fill:blue;')

LET attr[1] = om.SaxAttributes.create()
CALL attr[1].addAttribute(SVGATT_STROKE, "blue")
CALL attr[1].addAttribute(SVGATT_STROKE_WIDTH, 1.5 )
CALL attr[1].addAttribute(SVGATT_FILL, "none" )
CALL attr[1].addAttribute(SVGATT_MARKER_START, fglsvgcanvas.url("m1") )
CALL attr[1].addAttribute(SVGATT_MARKER_END, fglsvgcanvas.url("m2") )
CALL defs.appendChild( fglsvgcanvas.styleList( fglsvgcanvas.styleDefinition(".style_1",attr[1]) )
)
...
LET p = fglsvgcanvas.path("M100,100 L150,150 L200,150")
CALL p.setAttribute(SVGATT_STYLE,".style_1")

Related concepts

The fglsvgcanvas web component on page 1988
The **fglsvgcanvas** built-in web component implements a drawing canvas for Scalable Vector Graphics content.

**mask()**

Produces an SVG "mask" element.

**Syntax**

```javascript
mask(
  id STRING,
  x STRING,
  y STRING,
  width STRING,
  height STRING,
  node om.DomNode,
  name STRING )
```

1. *id* is the SVG object identifier.
2. *x* is the X coordinate of the mask.
3. *y* is the Y coordinate of the mask.
4. *width* is the width of the mask.
5. *height* is the height of the mask.
6. *maskUnits* defines the coordinate system for x,y,width,height.
7. *maskContentUnits* defines the coordinate system for the content of the mask.

**Usage**

This function creates a "mask" SVG DOM element from the parameters.

To use a mask in an SVG DOM element, set the "mask" attribute with the string generate by the `url()` function.

**Related concepts**

The **fglsvgcanvas** web component on page 1988

The **fglsvgcanvas** built-in web component implements a drawing canvas for Scalable Vector Graphics content.

**nl_to_tspan()**

Converts a string to an SVG "text" element with "tspan" sub-elements.

**Syntax**

```javascript
nl_to_tspan(
  text om.DomNode,
  x STRING,
  y STRING,
  dx STRING,
  dy STRING,
  content STRING )
```

1. *text* is the DOM node to update with tspan elements.
2. *x* is used to set the x attribute of each tspan element.
3. *y* is used to set the y attribute of each tspan element.
4. *dx* is used to set the dx attribute of each tspan element.
5. *dy* is used to set the dy attribute of each tspan element.
6. *content* is the source text with new-line characters.

**Usage**

This function creates "tspan" elements for each line of text.
The text DOM element passed as parameter gets "tspan" sub-elements for each new-line character found in the original string.

**Example**

```plaintext
DEFINE root_svg, t, n om.DomNode
...
LET t = fglsvgcanvas.text(NULL,200,NULL)
CALL root_svg.appendChild( t )
CALL fglsvgcanvas.nl_to_tspan(t,120,NULL,NULL,30,"Text using tspan
2\nLine 3")
```

**Related concepts**
The fglsvgcanvas web component on page 1988
The fglsvgcanvas built-in web component implements a drawing canvas for Scalable Vector Graphics content.

### path()

Produces an SVG "path" element.

**Syntax**

```plaintext
path(  
  d STRING  )
RETURNS om.DomNode
```

1. *d* is the SVG path specification.

**Usage**

This function creates a "path" SVG DOM element from the parameters.

The path element draws complex SVG shapes combined from lines, arcs, curves, etc.

For more details about path elements, see SVG specification.

**Example**

Displaying an image from an URL:

```plaintext
DEFINE n om.DomNode
LET n = fglsvgcanvas.path("M50,50 A30,30 0 0,1 35,20 ...")
CALL n.setAttribute(SVGATT_STYLE, "stroke:#006600; stroke-width:0.4")
```

**Related concepts**
The fglsvgcanvas web component on page 1988
The fglsvgcanvas built-in web component implements a drawing canvas for Scalable Vector Graphics content.

### pattern()

Produces an SVG "pattern" element.

**Syntax**

```plaintext
pattern(  
  id STRING,  
  x DECIMAL,  
  y DECIMAL,  
  width DECIMAL,  
  height DECIMAL,  
  patternUnits STRING,  
  patternContentUnits STRING,  
  patternTransform STRING,
```

1. *id* is the SVG pattern id.
2. *x*, *y* are the pattern coordinates.
3. *width*, *height* are the pattern dimensions.
4. *patternUnits* is the units for pattern coordinates.
5. *patternContentUnits* is the units for pattern content.
6. *patternTransform* is the transformation applied to the pattern.

For more details about pattern elements, see SVG specification.
preserveAspectRatio STRING )
RETURNS om.DomNode

1. id is the SVG object identifier.
2. x defines the X coordinate where the pattern starts.
3. y defines the Y coordinate where the pattern starts.
4. width defines the width of the pattern.
5. height defines the height of the pattern.
6. patternUnits defines the patternUnits attribute.
7. patternContentUnits defines the patternContentUnits attribute.
8. patternTransform defines the patternTransform attribute.
9. preserveAspectRatio is the aspect ratio to preserve.

Usage
This function creates a "pattern" SVG DOM element from the parameters.

The resulting DOM can be used in a "defs" element created with the defs() function, to produce pattern definitions.

Example

```
DEFINE pattern, n om.DomNode
...
LET pattern = fglsvgcanvas.pattern( "pattern1", 0,0,20,20, "userSpaceOnUse", "rotate(45)", NULL )
CALL pattern.appendChild( n:=fglsvgcanvas.rect(0,0,1000,10,NULL,NULL) )
CALL n.setAttribute(SVGATT_STYLE, 'stroke:none; fill:blue;' )
CALL pattern.appendChild( n:=fglsvgcanvas.rect(0,10,1000,10,NULL,NULL) )
CALL n.setAttribute(SVGATT_STYLE, 'stroke:none; fill:navy;' )
```

Related concepts
The fglsvgcanvas web component on page 1988
The fglsvgcanvas built-in web component implements a drawing canvas for Scalable Vector Graphics content.

polygon()
Produces an SVG "polygon" element.

Syntax

```
polygon( 
    points STRING )
RETURNS om.DomNode
```

1. points is a STRING defining the points of the polygon.

Usage
This function creates a "polygon" SVG DOM element from the parameters.

The points is a string containing a list of X,Y coordinates to draw the shape, in the form "x1,y1 x2,y2 ...".

Example

```
DEFINE n om.DomNode
LET n = fglsvgcanvas.polygon("10,10 10,20 20,20")
CALL n.setAttribute(SVGATT_STYLE, "stroke:#660000; fill:#cc3333;")
```
Related concepts
The fglsvgcanvas web component on page 1988
The fglsvgcanvas built-in web component implements a drawing canvas for Scalable Vector Graphics content.

polyline()
Produces an SVG "polyline" element.

Syntax

```javascript
polyline(
    points STRING
) RETURNS om.DomNode
```

1. `points` is a STRING defining the points of the polygon.

Usage
This function creates a "polygon" SVG DOM element from the parameters.
The `points` is a string containing a list of X,Y coordinates to draw the shape, in the form "x1,y1 x2,y2 ...".

Example

```javascript
DEFINE n om.DomNode
LET n = fglsvgcanvas.polyline("10,10 10,20 20,20")
CALL n.setAttribute(SVGATT_STYLE, "stroke:#660000;")
```

Related concepts
The fglsvgcanvas web component on page 1988
The fglsvgcanvas built-in web component implements a drawing canvas for Scalable Vector Graphics content.

radialGradient()
Produces an SVG "radialGradient" element.

Syntax

```javascript
radialgradient(
    id STRING,
    cx STRING,
    cy STRING,
    fx STRING,
    fy STRING,
    r STRING,
    spreadMethod STRING,
    gradientTransform STRING,
    gradientUnits STRING
) RETURNS om.DomNode
```

1. `id` is the SVG object identifier.
2. `cx` defines the X coordinate of the radial gradient center.
3. `cy` defines the Y coordinate of the radial gradient center.
4. `fx` defines the X coordinate of the radial gradient focal point.
5. `fy` defines the Y coordinate of the radial gradient focal point.
6. `spreadMethod` defines how the gradient is spread out through the shape.
7. `gradientTransform` defines the gradient transformation.
8. `gradientUnits` defines the coordinate system to be used.
Usage

This function creates a "radialGradient" SVG DOM element from the parameters.

An SVG "radialGradient" element can be used to define fill colors of shapes.

The element must contain "stop" elements that can be created with the stop function.

The resulting gradient definition can be added to a "defs" SVG element.

Example

```plaintext
DEFINE root_svg, defs, rg om.DomNode
...
LET defs = fglsvgcanvas.defs( NULL )
CALL root_svg.appendChild( defs )

LET rg = fglsvgcanvas.radialGradient( "gradient_1",
    NULL, NULL, "5%", "5%", "65%",
    "pad", NULL, NULL )
CALL rg.appendChild( fglsvgcanvas.stop( "0%", "gray", 0.4 ) )
CALL rg.appendChild( fglsvgcanvas.stop( "100%", "navy", 0.7 ) )
CALL defs.appendChild( rg )
```

Related concepts

The fglsvgcanvas web component on page 1988

The fglsvgcanvas built-in web component implements a drawing canvas for Scalable Vector Graphics content.

rect()

Produces an SVG "rect" element.

Syntax

```plaintext
rect( 
    x DECIMAL,
    y DECIMAL,
    width DECIMAL,
    height DECIMAL,
    rx DECIMAL,
    ry DECIMAL )
RETURNS om.DomNode
```

1. x defines the X coordinate of the top/left corner.
2. y defines the Y coordinate of the top/left corner.
3. width defines the width of the rectangle.
4. height defines the height of the rectangle.
5. rx defines the X radius for rounded corners.
6. ry defines the Y radius for rounded corners.

Usage

This function creates a "rect" SVG DOM element from the parameters.

Rounded borders are got by setting the rx, ry parameters.

Example

```plaintext
DEFINE n om.DomNode
LET n = fglsvgcanvas.rect(10,10,50,30,1,1)
```
Related concepts
The fglsvgcanvas web component on page 1988
The fglsvgcanvas built-in web component implements a drawing canvas for Scalable Vector Graphics content.

removeElement()
Deletes an SVG element from the SVG canvas.

Syntax

```plaintext
removeElement(node om.DomNode)
```

1. `node` is the om.DomNode to be removed.

Usage
This function deletes an SVG element from the current SVG canvas.
The DOM node to remove is passed as second parameter.
The SVG DOM node object is de-referenced and destroyed.

Example

```plaintext
DEFINE root_svg, g om.DomNode
...
LET g = fglsvgcanvas.g( "group1" )
CALL root_svg.appendChild( g )
...
CALL fglsvgcanvas.removeElement( g )
```

Related concepts
The fglsvgcanvas web component on page 1988
The fglsvgcanvas built-in web component implements a drawing canvas for Scalable Vector Graphics content.

setAttributes()
Sets the SVG attributes from an attribute set.

Syntax

```plaintext
setAttributes(node om.DomNode, attrs om.SaxAttributes)
```

1. `node` is the DOM node to be updated.
2. `attrs` is the om.SaxAttributes object defining the attributes.

Usage
This function updates the DOM node passed as first parameter with the om.SaxAttributes object.

Note: If a SAX attribute value is NULL, the attribute is removed from the node.
The setAttributes() function should only be used to set explicitly node-level attributes. Consider using CSS style definitions with the styleList() function, or inline styling with the styleAttributeList() function.

Steps to define and use an attribute set with setAttributes():
1. Create and fill an om.SaxAttributes object,
2. Create the SVG DOM element with one of the fglsvgcanvas functions,
3. Call the `setAttributes()` function with the node and the SAX attributes.

**Example**

```plaintext
CONSTANT COLORS_OCEAN = 1
DEFINE attr DYNAMIC ARRAY OF om.SaxAttributes,
   n om.DomNode
...
LET attr[COLORS_OCEAN] = om.SaxAttributes.create()
CALL attr[COLORS_OCEAN].addAttribute(SVGATT_FILL, "cyan")
CALL attr[COLORS_OCEAN].addAttribute(SVGATT_FILL_OPACITY, "0.3")
CALL attr[COLORS_OCEAN].addAttribute(SVGATT_STROKE, "blue")
...
LET n = fglsvgcanvas.polygon("10,10 10,20 20,20")
CALL fglsvgcanvas.setAttributes( n, attr[COLORS_OCEAN] )
```

**Related concepts**

The **fglsvgcanvas web component** on page 1988
The *fglsvgcanvas* built-in web component implements a drawing canvas for Scalable Vector Graphics content.

**setCurrent()**

Selects the SVG canvas handler for subsequent SVG canvas API calls.

**Syntax**

```plaintext
setCurrent( cid SMALLINT )
```

1. `cid` is the SVG canvas id, as returned by `fglsvgcanvas.create()`.

**Usage**

This function selects the SVG canvas identified by the specified `id` handler, for other fglsvgcanvas API calls.

Any subsequent calls to an fglsvgcanvas function that requires the SVG canvas handler will be done with this identifier.

**Example**

```plaintext
DEFINE cid1, cid2 SMALLINT
LET cid1 = fglsvgcanvas.create("formonly.canvas1") -- current canvas is cid1
LET cid2 = fglsvgcanvas.create("formonly.canvas1") -- current canvas is cid2
CALL fglsvgcanvas.setcurrent( cid1 ) -- current canvas is cid1
```

**Related concepts**

The **fglsvgcanvas web component** on page 1988
The *fglsvgcanvas* built-in web component implements a drawing canvas for Scalable Vector Graphics content.

**setRootSVGAttributes()**

Produces the root SVG element.

**Syntax**

```plaintext
setRootSVGAttributes( 
   id STRING, 
   width STRING, 
   height STRING, 
   viewBox STRING, 
   preserveAspectRatio STRING ) 
RETURNS om.DomNode
```
1. *id* is the SVG object identifier.
2. *width* and *height* define the SVG viewport.
3. *viewBox* defines the SVG viewbox (the internal coordinate system).
4. *preserveAspectRatio* is the aspect ratio to preserve.

**Usage**

This function sets the attributes of the root "svg" SVG element from the parameters, and returns the SVG DOM element.

**Note:** This method does not create a new DOM node object (it is created when calling the *create()* function): You can repeat calls to the *setRootSVGAttributes()* function to change/reset the root DOM node SVG attributes.

Create the SVG canvas with the following sequence of calls:

1. Initialize the fglsvgcanvas library with *initialize()*.
2. Create a new SVG canvas handler with *create()* (this creates the DOM node for the root SVG).
3. Set root SVG attributes and get the corresponding DOM node object with *setRootSVGAttributes()*.
4. Use the root DOM node to append child DOM element created from fglsvgcanvas functions.

The *width* and *height* attributes define the SVG viewport, the visible area of the SVG image. You want to leave the viewport attributes to NULL, to let the SVG image adapt to its container.

The viewBox is specified with the *viewBox* parameter, to define the coordinate system to draw SVG objects. For example "0 0 100 100" defines a viewBox where 0,0 are the coordinates of the top/left corner, and 100,100 are the coordinates of the bottom/right corner. If you draw a rectangle with 25,25,50,50 it will be centered and use half of the viewBox.

The *preserveAspectRatio* parameter is used in conjunction with the *viewBox* attribute, to control how the SVG content is stretched, whether or not to force uniform scaling. In most case you want to use the "xMidYMid meet" as aspect ratio.

See SVG reference for more details about viewports and viewboxes.

**Example**

```plaintext
DEFINE cid SMALLINT
DEFINE root_svg om.DomNode
CALL fglsvgcanvas.initialize()
LET cid = fglsvgcanvas.create("formonly.canvas")
LET root_svg = fglsvgcanvas.setRootSVGAttributes(
    "agenda",
    NULL, NULL,
    "0 0 100 1000",
    "xMidYMin slice"
)
```

**Related concepts**

The fglsvgcanvas web component on page 1988
The fglsvgcanvas built-in web component implements a drawing canvas for Scalable Vector Graphics content.

**stop()**

Produces an SVG "stop" element for gradients.

**Syntax**

```
stop(  
    offset STRING,  
    color STRING,  
    opacity STRING )
```
Returns `om.DomNode`

1. `offset` defines the offset SVG attribute.
2. `color` defines the stop-color SVG attribute.
3. `opacity` defines the stop-opacity SVG attribute.

Usage

This function creates a "stop" SVG DOM element from the parameters.

An SVG "stop" element defines the ramp of colors to use on a gradient in a "linearGradient" or "radialGradient" element.

For a usage example, see `linearGradient()` on page 2347.

Related concepts

The `fglsvgcanvas` web component on page 1988

The `fglsvgcanvas` built-in web component implements a drawing canvas for Scalable Vector Graphics content.

`styleAttributeList()`

Builds a string with a list of attributes to be used in a `style` attribute.

Syntax

```plaintext
styleAttributeList(
  attrs om.SaxAttributes)
RETURNS STRING
```

1. `attrs` is the `om.SaxAttributes` object defining the attributes.

Usage

This function builds string containing a list of `name:value;` pairs, from the attribute set defined by an `om.SaxAttributes` object.

The resulting string can then be used to define a CSS style, or it can be defined in an SVG element using `style="attribute-list"`.

Steps to define and use an attribute set with `styleAttributeList()`:

1. Create and fill an `om.SaxAttributes` object,
2. Create the SVG DOM element with one of the `fglsvgcanvas` functions,
3. Set the "style" attribute with value produced from `styleAttributeList()`.

Example

```plaintext
CONSTANT COLORS_FIRE = 1
DEFINE attr DYNAMIC ARRAY OF om.SaxAttributes,
    n om.DomNode
...
LET attr[COLORS_FIRE] = om.SaxAttributes.create()
CALL attr[COLORS_FIRE].setAttribute(SVGATT_STROKE,"red")
...
LET n = fglsvgcanvas.circle(100,100,50)
CALL n.setAttribute(SVGATT_STYLE,
    fglsvgcanvas.styleAttributeList(attr[COLORS_FIRE]))
...
```

Related concepts

The `fglsvgcanvas` web component on page 1988
The fglsvgcanvas built-in web component implements a drawing canvas for Scalable Vector Graphics content.

**styleDefinition()**
Produces a CSS style definition with a selection and list of attributes.

**Syntax**

```
styleDefinition(
    selector STRING,
    attrs om.SaxAttributes )
RETURNS STRING
```

1. `selector` is the style selector.
2. `attrs` is the `om.SaxAttributes` object defining the attributes.

**Usage**
This function creates a CSS style definition line from the attribute set defined by an `om.SaxAttributes` object.

The attribute set must be an `om.SaxAttributes` object.
The resulting string can be used in a style list created by the `styleList()` function.

Steps to define and use an attribute set with `styleDefinition()`:

1. Create and fill an `om.SaxAttributes` object,
2. Use the `styleDefinition()` function using the SAX attributes.

**Related concepts**
The fglsvgcanvas web component on page 1988
The fglsvgcanvas built-in web component implements a drawing canvas for Scalable Vector Graphics content.

**styleList()**
Produces a CSS style list.

**Syntax**

```
styleList(
    content STRING )
RETURNS om.DomNode
```

1. `content` is the string that is a concatenation of style definitions produced by `styleDefinition()` functions.

**Usage**
This function creates a "style" SVG DOM element with a text sub-node containing a list of style definitions, to create a CSS style. The resulting DOM element can then be used in a "defs" element.

A single CSS style definition can be produced with the `styleDefinition()` function.

**Example**

```
CONSTANT STYLE_1 = 1
CONSTANT STYLE_2 = 2
CONSTANT STYLE_3 = 3
DEFINE attr DYNAMIC ARRAY OF om.SaxAttributes,
    root_svg, defs om.DomNode
...
LET attr[STYLE_1] = om.SaxAttributes.create()
CALL attr[STYLE_1].setAttribute(SVGATT_STROKE,"red")
...
LET attr[STYLE_2] = om.SaxAttributes.create()
...
LET attr[STYLE_3] = om.SaxAttributes.create()
...
LET defs = fglsvgcanvas.defs( NULL )
CALL defs.appendChild( fglsvgcanvas.styleList(
  fglsvgcanvas.styleDefinition(".style_1",attr[STYLE_1])
  ||
  fglsvgcanvas.styleDefinition(".style_2",attr[STYLE_2])
  ||
  fglsvgcanvas.styleDefinition(".style_3",attr[STYLE_3])
)
)
CALL root_svg.appendChild( defs )

Related concepts
The fglsvgcanvas web component on page 1988
The fglsvgcanvas built-in web component implements a drawing canvas for Scalable Vector Graphics content.

svg()
Produces an SVG "svg" element.

Syntax

```
svg(
  id STRING,
  x STRING,
  y STRING,
  width STRING,
  height STRING,
  viewBox STRING,
  preserveAspectRatio STRING
RETURNS om.DomNode
```

1.  id is the SVG object identifier.
2.  x, y, width, height define the SVG viewport.
3.  viewBox defines the SVG viewbox.
4.  preserveAspectRatio is the aspect ratio to preserve.

Usage
This function creates a "svg" SVG DOM element definition from the parameters.

SVG allows you to create nested svg sub-elements with their own coordinate system.

Build a tree of svg elements as you need.

Note: The root svg element must be created with the setRootSVGAttributes() function.

The viewport is defined by the x, y, width and height attributes.

The viewbox is defined by the viewBox string for example as "0 0 100 100".

See SVG reference documentation for more details about viewport, viewbox and aspect ratio concepts.

Example

```
DEFINE root_svg, n om.DomNode
...
LET n = fglsvgcanvas.svg("day_1",
                     NULL, NULL, NULL, NULL,
                     "0 0 500 500",
```
"xMidYMid meet"

... CALL root_svg.appendChild( n )

**Related concepts**

*The fglsvgcanvas web component* on page 1988

The fglsvgcanvas built-in web component implements a drawing canvas for Scalable Vector Graphics content.

**text()**

Produces an SVG "text" element.

**Syntax**

```plaintext
text(
    x DECIMAL,
    y DECIMAL,
    content STRING,
    class STRING )
RETURNS om.DomNode
```

1. *x* and *y* define the position of the text.
2. *content* is the actual text.
3. *class* defines a reference to a CSS style.

**Usage**

This function creates a "text" SVG DOM element from the parameters.

To specify the text font attributes, define a CSS style in a `defs()` element with the `styleList()` function, and reference the text style in the *class* parameter of this function.

**Example**

```plaintext
DEFINE n om.DomNode
LET n = fglsvgcanvas.text(10,10,"Hello!","mystyle_1")
```

**Related concepts**

*The fglsvgcanvas web component* on page 1988

The fglsvgcanvas built-in web component implements a drawing canvas for Scalable Vector Graphics content.

**text_path()**

Produces the SVG "text" element with a "textPath" sub-element.

**Syntax**

```plaintext
text_path(
    x DECIMAL,
    y DECIMAL,
    content STRING,
    path STRING
    class STRING )
RETURNS om.DomNode
```

1. *x* and *y* define the position of the text.
2. *content* is the actual text.
3. *path* is the xlink:href reference (without #).
4. *class* defines a reference to a CSS style.
Usage

This function creates a "text" SVG DOM element from the parameters, including a "textPath" sub-element that references a "path" element defined in a "defs" element.

The path parameter is used to build an "xlink:href=#path" reference.

The actual path can be created with the path() function, and included in a "defs" element created with the defs() function.

To specify the text font attributes, define a CSS style in a defs() element with the styleList() function, and reference the text style in the class parameter of this function.

Example

```om
DEFINE root_svg, defs, p, n om.DomNode
...
LET defs = fglsvgcanvas.defs( NULL )
CALL root_svg.appendChild( defs )
...
CALL defs.appendChild( p:=fglsvgcanvas.path("M150,400 C175,380 225,320 450,450") )
CALL p.setAttribute("id", "path_1")
LET n = fglsvgcanvas.text_path( NULL,NULL,
"This text follows a path...","path_1","style_4")
...
```

Related concepts

The fglsvgcanvas web component on page 1988
The fglsvgcanvas built-in web component implements a drawing canvas for Scalable Vector Graphics content.

text_tref()

Produces the SVG "text" element with a "tref" sub-element.

Syntax

```
text_tref(  
  x DECIMAL,  
  y DECIMAL,  
  tref STRING,  
  class STRING )  
RETURNS om.DomNode
```

1. x and y define the position of the text.
2. tref is the xlink reference to the tref.
3. class defines a reference to a CSS style.

Usage

This function creates a "text" SVG DOM element from the parameters, including a "tref" sub-element that references a "text" element defined in a "defs" element.

The tref parameter is used to build an "xlink:href=#path" reference.

The "text" element referenced by the "tref" attribute must be defined in a "defs" element, created with thedefs() function.

To specify the text font attributes, define a CSS style in a defs() element with the styleList() function, and reference the text style in the class parameter of this function.
Example

```oml
DEFINE root_svg, def, t, n om.DomNode
...
LET def = fglsvgcanvas.defs( NULL )
CALL root_svg.appendChild( def )
...
CALL def.appendChild( t:=fglsvgcanvas.text( NULL, NULL, 
"The is the referenced text...") )
CALL t.setAttribute("id","text_1")
LET n = fglsvgcanvas.text_tref(10,20,"text_1","style_4")
...
```

Related concepts

- The `fglsvgcanvas` web component on page 1988
- The `fglsvgcanvas` built-in web component implements a drawing canvas for Scalable Vector Graphics content.

**title()**

Produces an SVG "title" element.

**Syntax**

```oml
title( text STRING )
    RETURNS om.DomNode
```

1. `text` defines text for the title element.

**Usage**

This function creates a "title" SVG DOM element from the text provided as parameter.

In SVG, a title element is usually rendered with a tooltip on desktop platforms.

**Note:** The title element must be the first child of the parent SVG element. See SVG specification for more details.

**Example**

```oml
DEFINE g om.DomNode
LET g = fglsvgcanvas.g( "g1" )
CALL g.appendChild( fglsvgcanvas.title("This is my text") )
```

Related concepts

- The `fglsvgcanvas` web component on page 1988
- The `fglsvgcanvas` built-in web component implements a drawing canvas for Scalable Vector Graphics content.

**tspan()**

Produces an SVG "tspan" element.

**Syntax**

```oml
tspan( 
    x DECIMAL, 
    y DECIMAL, 
    dx STRING, 
    dy STRING, 
    style STRING, 
    content STRING )
    RETURNS om.DomNode
```

1. `x` defines the X position of the tspan element.
2. \( y \) defines the Y position of the \( \text{tspan} \) element.

3. \( dx \) defines the delta-X for horizontal positioning of the \( \text{tspan} \) element.

4. \( dy \) defines the delta-Y for vertical positioning of the \( \text{tspan} \) element.

5. \( style \) defines the in-line style to be applied on the \( \text{tspan} \) element.

6. \( content \) is the actual text.

Usage

This function creates a "\( \text{tspan} \)" SVG DOM element from the parameters.

A \( \text{tspan} \) element can be used to draw multiple lines of text in SVG.

The DOM nodes created by this function must be added to a "\( \text{text} \)" element produced by the \( \text{text()} \) function.

Example

```
DEFINE root_svg, t om.DomNode  
...  
LET t = fglsvgcanvas.text(NULL,200,NULL,"style_2")  
CALL root_svg.appendChild( t )  
CALL t.appendChild( fglsvgcanvas.tspan(120,NULL,NULL,30,NULL,"First line") )  
CALL t.appendChild( fglsvgcanvas.tspan(120,NULL,NULL,30,NULL,"Second line") )
```

Related concepts

- The fglsvgcanvas web component on page 1988
- The fglsvgcanvas built-in web component implements a drawing canvas for Scalable Vector Graphics content.

url()

Produces a "url(#name)" reference for SVG elements.

Syntax

```
url( name STRING )     
RETURNS STRING
```

1. \( name \) is the URL name.

Usage

This function builds a "url(#name)" SVG attribute with the name passed as parameter.

This URL attribute is typically used in SVG elements to reference a common reusable element defined in the "defs" element, such as SVG patterns.

Example

```
CALL n.setAttribute(SVGATT_STYLE,  
    SFMT('stroke:gray; fill:%1;', fglsvgcanvas.url("pattern_1") )
)
```

Related concepts

- The fglsvgcanvas web component on page 1988
The `fglsvgcanvas` built-in web component implements a drawing canvas for Scalable Vector Graphics content.

**use()**

Produces an SVG "use" element.

**Syntax**

```plaintext
use(
  name STRING,
  x DECIMAL,
  y DECIMAL
)
RETURNS om.DomNode
```

1. `name` is the xlink:href reference (without #).
2. `x` defines the X coordinate to place the shape.
3. `y` defines the Y coordinate to place the shape.

**Usage**

This function creates a "use" SVG DOM element from the parameters.

The "use" element can reuse an SVG shape from elsewhere in the SVG document, for example to make several copies of shapes defined in the "defs" element.

**Example**

```plaintext
DEFINE n1, n2 om.DomNode
LET n1 = fglsvgcanvas.use( "myshape1", 100, 100 )
LET n2 = fglsvgcanvas.use( "myshape1", 110, 110 )
```

**Related concepts**

- The `fglsvgcanvas` web component on page 1988
- The `fglsvgcanvas` built-in web component implements a drawing canvas for Scalable Vector Graphics content.

---

**getopt: Command line options module**

**Usage**

See Getopt module usage on page 2367 to understand how the getopt module can be used.
Table 500: GetOpt types (getopt.4gl)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| PUBLIC TYPE GetoptOptions DYNAMIC ARRAY OF RECORD  
  name STRING,  
  description STRING,  
  opt_char CHAR,  
  arg_type INTEGER  
  END RECORD |
| The GetoptOptions structured array type that holds the definition of command line options. |
| PUBLIC TYPE Getopt RECORD  
  ... private members not documented here ...  
  opt_ind INTEGER,  
  opt_char CHAR,  
  opt_arg STRING  
  END RECORD |
| The Getopt structured type is used to process command line options. |

Table 501: GetOpt constants (getopt.4gl)

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetOpt constants</td>
<td>List of predefined constants for the getopt API.</td>
</tr>
</tbody>
</table>

Table 502: GetOpt functions (getopt.4gl)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
</table>
| copyArguments ( ind INTEGER )  
  RETURNS DYNAMIC ARRAY OF STRING |
| Returns a dynamic array of string with all command line arguments starting from the provided index. |
### Table 503: GetOpt methods (getopt.4gl)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Getopt.displayUsage(more_args STRING)</code></td>
<td>Display the usage and command line option description to the standard output stream.</td>
</tr>
<tr>
<td><code>Getopt.getMoreArgumentCount()</code></td>
<td>Returns the number of command line arguments left to be processed after the known options.</td>
</tr>
<tr>
<td><code>Getopt.getMoreArgumentCount()</code></td>
<td>Returns the additional argument at the specified index.</td>
</tr>
<tr>
<td><code>Getopt.getopt()</code></td>
<td>Process the next command line option.</td>
</tr>
<tr>
<td><code>Getopt.initDefault(options GetoptOptions)</code></td>
<td>Initializes a variable defined with the Getopt type.</td>
</tr>
<tr>
<td><code>Getopt.initialize(prog_name STRING, argv DYNAMIC ARRAY OF STRING, options GetoptOptions)</code></td>
<td>Initializes a variable defined with the Getopt type for command line argument processing.</td>
</tr>
<tr>
<td><code>Getopt.invalidOptionSeen()</code></td>
<td>Checks if the command line options are misused.</td>
</tr>
<tr>
<td><code>Getopt.isEof()</code></td>
<td>Extracts an address from a string representing a vCard.</td>
</tr>
<tr>
<td><code>Getopt.isSuccess()</code></td>
<td>Checks if a command line option parsing succeeded.</td>
</tr>
</tbody>
</table>

### Getopt module usage

The `getopt.4gl` module provides command line argument processing.

### Features of getopt.4gl

The `getopt.4gl` module implements types, functions and methods to process command line arguments in a standard way.

Use this library to implement arguments processing, and provide a common command line option syntax for all your programs.

The options can be defined with a short name (like `-c`) and a long name (like `--compare`).
Options can have a mandatory or optional value to be provided with the `--option-name=option-value` form:

$ fglrun myprog --verbose --level=5

If an argument on the command line is in the form `@filename`, options will be read from the file:

$ fglrun myprog @myoptions

Options can start at a given index in the command line arguments, to support for example commands with verbs followed by options:

$ fglrun myprog capture --verbose --filename=file1
$ fglrun myprog duplicate --source=file1 --destination=file2

Command line arguments can end with a set of free arguments, to be processed after the options:

$ fglrun myprog capture --verbose file1 file2 file3

**Steps to implement arguments processing**

To implement command line arguments processing with `getopt`, do the following steps:

1. Define a variable with the `Getopt` type:

   ```
   DEFINE g getopt.Getopt
   ```

2. Define a variable with the `GetoptOptions` type that will contain the definition of the options (fill this array of options with a variable initializer):

   ```
   DEFINE _options getopt.GetoptOptions
   ...]
   ```

3. Initialize the `Getopt` variable with the `initialize()` or `initDefault()` method, passing the `GetoptOptions` array as parameter:

   ```
   CALL g.initialize("myprog", getopt.copyArguments(2), _options)
   ```

   If needed, use the `copyArguments(index)` function, to provide the second parameter of the `initialize()` method, to start command line argument parsing at a given index.

4. Use the `getopt()` method in a `WHILE` loop, to process all command line arguments that correspond to an option definition:

   ```
   WHILE g.getOpt() == getopt.SUCCESS
   CASE g.opt_char
   WHEN 'v'
     DISPLAY "Version 1.50"
     EXIT PROGRAM 0
   WHEN 'h'
     CALL g.displayUsage("file ...")
     EXIT PROGRAM 0
   WHEN 'o'
     LET outfile = g.opt_arg
   END CASE
   END WHILE
   ```

   Additionally, the `isEof()` method can be used to check if all possible options are processed.
5. After the WHILE loop, check for the processing status with the methods invalidOptionSeen() or isSuccess():

```plaintext
IF g.invalidOptionSeen() THEN
    CALL g.displayUsage("file ...")
EXIT PROGRAM 1
END IF
```

6. If additional arguments are possible, use the getMoreArgumentCount() and getMoreArgument() methods, to process these non-option arguments:

```plaintext
LET cnt = g.getMoreArgumentCount()
IF cnt == 0 THEN
    DISPLAY "ERROR: No files were provided..."
    EXIT PROGRAM 1
ELSE
    FOR ind = 1 TO cnt
        DISPLAY SFMT("File to process: %1", g.getMoreArgument(ind))
    END FOR
END IF
```

For a complete example, see getopt() on page 2373.

**GetoptOptions type**
The GetoptOptions structured array type that holds the definition of command line options.

**Syntax**

```plaintext
PUBLIC TYPE GetoptOptions DYNAMIC ARRAY OF RECORD
    name STRING,
    description STRING,
    opt_char CHAR,
    arg_type INTEGER
END RECORD
```

1. name defines the long name of the command line option.
2. description is the text to explain the command line option.
3. opt_char is the single-char command line option name.
4. arg_type can be one of:
   - getopt.NONE: The option has no additional value argument.
   - getopt.OPTIONAL: The option can be used with an optional value argument (--option=value).
   - getopt.REQUIRED: The option needs a mandatory value argument (--option=value).

**Usage**
This type defines a dynamic array of a record structure to hold command line options definitions information.

Define a variable of the getopt.GetoptOptions type and fill it with an initializer.

Once the array is initialized, it can be passed to the initDefault() or initialize() method, to setup a Getopt variable in order to process command line arguments with the getopt() method.

**Example**

```plaintext
IMPORT FGL getopt
MAIN
    DEFINE g getopt.Getopt
    DEFINE _options getopt.GetoptOptions
```
Getopt type
The Getopt structured type is used to process command line options.

Syntax

PUBLIC TYPE Getopt RECORD
  ... private members not documented here ...
  opt_ind INTEGER,
  opt_char CHAR,
  opt_arg STRING
END RECORD

1. opt_ind current command line argument index that is processed.
2. opt_char is the single-character short name of the current processed option.
3. opt_arg if present, holds the value parameter of the current processed option (--option=value). Otherwise, this member is NULL.

Usage
This type defines the Getopt record that is used with getopt methods to parse and validate command line options.

A variable of the type Getopt must be defined and initialized with the initDefault() or initialize() method, before using the getopt() method in a WHILE loop, to process command line options.

Example

IMPORT FGL getopt

MAIN
  DEFINE g getopt.Getopt
  DEFINE _options getopt.GetoptOptions = [ ... ]
  CALL g.initDefault(_options)
  ...
END MAIN

GetOpt constants
List of predefined constants for the getopt API.

Syntax

PUBLIC CONSTANT NONE     = 1
PUBLIC CONSTANT REQUIRED = 2
PUBLIC CONSTANT OPTIONAL = 3
PUBLIC CONSTANT SUCCESS = 0
PUBLIC CONSTANT EOF = 1
PUBLIC CONSTANT BAD_ARGUMENT = 2

Note: Constant values are provided in this syntax diagram for information. You should obviously use the constant names instead of the values.

Usage
The GetOpt predefined constants are used to implement command line option processing with the getopt module.

The getopt.NONE, getopt.REQUIRED, getopt.OPTIONAL constants are used to define the opt_arg member of elements of the GetoptOptions array used to define the command line options.

The getopt.SUCCESS, getopt.EOF, getopt.BAD_ARGUMENT are return codes of the getopt() method.

copyArguments()
Returns a dynamic array of string with all command line arguments starting from the provided index.

Syntax

```plaintext
copyArguments ( ind INTEGER )
RETURNS DYNAMIC ARRAY OF STRING
```

1. `ind` is the command line argument index to start with, to produce the dynamic array of string. Use 1 for the first argument (0 is the program name).

Usage
The copyArguments() function builds a dynamic array of strings with command line arguments, starting from the index passed as parameter.

Use this function as second parameter for the initialize() method, to provide the list of command line arguments to be processed.

For more details, see initialize().

displayUsage()
Display the usage and command line option description to the standard output stream.

Syntax

```plaintext
Getopt.displayUsage(  
    more_args STRING
  )
```

1. `more_args` is a character string to be displayed at the end of the "Usage..." line.

Usage
This is a method for the Getopt type, used to display the description of the command line options of the current program.

The `more_args` passed as parameter to the displayUsage() method will be shown in the first line, after the option list:

```plaintext
Usage: program-name [options] more_args
```
Before using the `displayUsage()` method, a variable of the type `Getopt` must be defined and initialized with the `initDefault()` or `initialize()` method.

The `displayUsage()` method is typically called when an invalid command line option is detected (when `invalidOptionSeen()` returns TRUE) or when the `--help/-h` option is used.

**Example**

```fortran
IMPORT FGL getopt

MAIN
  DEFINE g getopt.Getopt
  ...
  IF g.opt_char=="h" THEN
    CALL g.displayUsage(NULL)
    EXIT PROGRAM 0
  END IF
  ...
END MAIN
```

**getMoreArgumentCount()**

Returns the number of command line arguments left to be processed after the known options.

**Syntax**

```fortran
Getopt.getMoreArgumentCount ( )
RETURNS INTEGER
```

**Usage**

This is a method for the `Getopt` type, that returns the number of command line arguments left to process, after all possible options defined in the `Getopt` object have been detected.

For more details see `getMoreArgument()`.

**getMoreArgument()**

Returns the additional argument at the specified index.

**Syntax**

```fortran
Getopt.getMoreArgument ( ind INTEGER )
RETURNS STRING
```

1. `ind` is the index of the additional argument, from 1 to `getMoreArgumentCount()`.

**Usage**

After processing all command line options, it is possible to scan for additional command line arguments, that are not part of the options defined in the `GetoptOptions` array of the `Getopt` object.

getMoreArgument() is a method for the `Getopt` type, that returns the additional argument at the specified index, after all possible options defined in the `Getopt` object have been detected.

For example, with a command using a set of filenames as argument, you want to allow:

```bash
$ fglrun reduce --verbose --ratio=5 file1 file2 file3
```

In such case, the `--verbose` and `--ratio` options must be defined by the `Getopt` object and processed with the `getopt()` method, while `file1`, `file2` and `file3` are additional arguments to be processed after the `getopt()`
loop, using the `getMoreArgumentCount()` method to get the total number of additional arguments, and the `getMoreArgument()` method to retrieve the additional argument.

**Example**

```plaintext
IMPORT FGL getopt

MAIN
DEFINE g getopt.Getopt
DEFINE cnt, ind INTEGER
...
WHILE g getopt() == getopt.SUCCESS
...
END WHILE

IF g invalidOptionSeen() THEN
    CALL g.displayUsage(NULL)
    EXIT PROGRAM 1
ELSE
    LET cnt = g.getMoreArgumentCount()
    IF cnt == 0 THEN
        DISPLAY "ERROR: No files were provided..."
        EXIT PROGRAM 1
    ELSE
        FOR ind = 1 TO cnt
            DISPLAY SFMT("File to process: %1", g.getMoreArgument(ind))
        END FOR
    END IF
END IF
END MAIN
```

**getopt()**
Process the next command line option.

**Syntax**

```plaintext
getopt.getopt( )
RETURNS INTEGER
```

1. The returned status can be one of:
   - `getopt.SUCCESS`: A command line argument was detected and can be processed.
   - `getopt.EOF`: No more command line arguments to process, exit the `WHILE` loop.
   - `getopt.BAD_ARGUMENT`: An invalid command line argument was detected, exit the `WHILE` loop and inform the user.

**Usage**

This is a method for the `Getopt` type, that processes the next command line option.

Before using the `getopt()` method, a variable of the type `Getopt` must be defined and initialized with the `initDefault()` or `initialize()` method.

The `getopt()` method is typically used with a `WHILE` loop to process all command line arguments, by checking that the method returns `getopt.SUCCESS`:

```plaintext
WHILE g getopt() == getopt.SUCCESS
...
END WHILE
```
Inside the **WHILE** loop, check the properties of the Getopt record to identify the current option that has been found. The `opt_char` member identifies the current option with its single-character short name, and the `opt_arg` member holds the option argument (**--option=value**), if one is available:

```plaintext
CASE g.opt_char
    WHEN 'e'
        CALL extract(g.opt_arg)
    ...
END CASE
```

When the **WHILE** loop ends because `getopt()` has returned a status different from `getopt.SUCCESS`, handle errors by checking the status with the `invalidOptionSeen()` method. If all provided options are correct, process additional command line arguments if needed. For more details, see `getMoreArgument()`.

**Example**

```plaintext
IMPORT FGL getopt

MAIN
    DEFINE ind INTEGER
    DEFINE outfile STRING
    DEFINE g getopt.Getopt
    DEFINE _options getopt.GetoptOptions
        = [(name: "version",
            description: "Version information",
            opt_char: 'v',
            arg_type: getopt.NONE),
           (name: "help",
            description: "This help page",
            opt_char: 'h',
            arg_type: getopt.NONE),
           (name: "outfile",
            description: "Output filename",
            opt_char: 'o',
            arg_type: getopt.REQUIRED)]
    CALL g.initDefault(_options)
    WHILE g.getOpt() == getopt.SUCCESS
        CASE g.opt_char
            WHEN 'v'
                DISPLAY "Version 1.50"
                EXIT PROGRAM 0
            WHEN 'h'
                CALL g.displayUsage("file ...")
                EXIT PROGRAM 0
            WHEN 'o'
                LET outfile = g.opt_arg
        END CASE
    END WHILE
    IF g.invalidOptionSeen() THEN
        CALL g.displayUsage("file ...")
        EXIT PROGRAM 1
    END IF
END MAIN
```
**initDefault()**
Initializes a variable defined with the `Getopt` type.

**Syntax**

```
Getopt.initDefault(
    options GetoptOptions
)
```

1. `options` is a `GetoptOptions` array that holds the definition of the command line options.

**Usage**

This is a method for the `Getopt` type, used to initialize the variable of this type, using the current program name (`arg_val(0)`) and where the command line argument starting index is 1 (`copyArguments(1)`).

A variable of the type `Getopt` must be defined, as well as a `GetoptOptions` dynamic array containing the definitions of the command line options for the current program.

After initializing the `Getopt` record, use the `getopt()` method to parse and validate command line options.

**Example**

```julia
IMPORT FGL getopt
MAIN
    DEFINE g getopt.Getopt
    DEFINE _options getopt.GetoptOptions = [ ... ]
    CALL g.initDefault(_options)
    WHILE g.getopt() == getopt.SUCCESS
        ...
    END WHILE
END MAIN
```

**initialize()**
Initializes a variable defined with the `Getopt` type for command line argument processing.

**Syntax**

```
Getopt.initialize(
    prog_name STRING,
    argv DYNAMIC ARRAY OF STRING,
    options GetoptOptions
)
```

1. `prog_name` is the name of the program (typically, `arg_val(0)`).
2. `argv` is a dynamic array of strings containing all command line arguments to be processed.
3. `options` is a `GetoptOptions` array that holds the definition of the command line options.

**Usage**

This is a method for the `Getopt` type, used to initialize the variable of this type.

**Tip:** Consider using the `initDefault()` method instead of `initialize()`, if the program name must be `arg_val(0)` and the start index to scan command line arguments is 1.

A variable of the type `Getopt` must be defined, as well as a `GetoptOptions` dynamic array containing the definitions of the command line options for the current program.
The first parameter defines the name of the program. This is typically `arg_val(0)`, but it can be customized.

The second argument specifies the list of command line arguments to be processed. This is typically `copyArguments(1)`. For example, start at the second command line argument with `copyArguments(2)`, to implement a command syntax with a verb as first argument:

```plaintext
$ fglrun myprog capture --verbose --filename=file1
$ fglrun myprog duplicate --source=file1 --destination=file2
```

The third argument contains the definition of the command options that are available with this program, defined in a `GetoptOptions` array.

After initializing the `Getopt` record, use the `getopt()` method to parse and validate command line options.

**Example**

```plaintext
IMPORT FGL getopt
MAIN
  DEFINE g getopt.Getopt
  DEFINE _options getopt.GetoptOptions = [ ... ]
  CALL g.initialize("myprog", getopt.copyArguments(2), _options)
  WHILE g.getopt() == getopt.SUCCESS
    ...
  END WHILE
END MAIN
```

**Related concepts**

`initDefault()` on page 2375
Initializes a variable defined with the `Getopt` type.

`invalidOptionSeen()`
Checks if the command line options are misused.

**Syntax**

```plaintext
Getopt.invalidOptionSeen( )
RETURNS BOOLEAN
```

**Usage**

This is a method for the `Getopt` type, that returns TRUE if the command line options are not specified correctly as defined by the `GetoptOptions` definition of the `Getopt` object.

The command line is considered as invalid in the following cases:

- When an unknown option is used (not defined in the `GetoptOptions` array).
- When a required option argument is missing (`arg_type == getopt.REQUIRED`).

The `invalidOptionSeen()` method can be used after calling the `getopt()` method processing command line options in a loop, to detect invalid options usage.

**Note:** When an invalid command line option usage is detected, display the usage with the `displayUsage()` method and exit the program with an error status with `EXIT_PROGRAM 1`.

**Example**

```plaintext
IMPORT FGL getopt
MAIN
```
DEFINE g getopt.Getopt
...
WHILE g.getopt() == getopt.SUCCESS
...
END WHILE

IF g.invalidOptionSeen() THEN
    CALL g.displayUsage()
    EXIT PROGRAM 1
END IF

END MAIN

isEof()
Extracts an address from a string representing a vCard.

Syntax

Getopt.isEof( )
RETURNS BOOLEAN

Usage

This is a method for the Getopt type, used to check if there are no more command line options to process.

The isEof() method can be used after calling the getopt() method processing command line options, to stop the processing when no more known command line options are to be processed.

Example

IMPORT FGL getopt

MAIN
    DEFINE g getopt.Getopt
    DEFINE s INTEGER
    ...
    WHILE TRUE
        LET s = g.getOpt()
        IF g.isEof() THEN EXIT WHILE END IF
        DISPLAY g.opt_char
    END WHILE

END MAIN

isSuccess()
Checks if a command line option parsing succeeded.

Syntax

Getopt.isSuccess( )
RETURNS BOOLEAN

Usage

This is a method for the Getopt type, that returns TRUE if last command line argument was processed successfully.

The isSuccess() method can be used after calling the getopt() method processing command line options in a loop, to detect correct options usage.
Note: The processing status of the current option is typically checked with the return code of the `getopt()` method.

Example

```lisp
IMPORT FGL getopt

MAIN
  DEFINE g getopt.Getopt
  ...
  WHILE g.getopt() == getopt.SUCCESS
    ...
  END WHILE
  IF g.isSuccess() THEN
    ...
  END IF
END MAIN
```

Built-in packages

These topics cover the built-in classes provided by the Genero Business Development Language.

Note: Unlike extension packages, built-in packages do not need to be imported with the `IMPORT` statement.

BDL data types package

These topics cover the built-in classes of BDL data types.

BYTE data type as class

The BYTE primitive data type provides a set of utility methods to manipulate BYTE data.

BYTE methods can be invoked with the variable, for example:

```lisp
DEFINE b BYTE
  ...
  CALL b.writeFile("mydata")
```

Related concepts

BYTE on page 290

The BYTE data type stores any type of binary data, such as images or sounds.

BYTE data type methods

Table 504: Object methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>readFile(path STRING)</code></td>
<td>Reads a file into a BYTE locator.</td>
</tr>
<tr>
<td><code>writeFile(path STRING)</code></td>
<td>Writes the content of a BYTE to a file.</td>
</tr>
</tbody>
</table>
BYTE.readFile
Reads a file into a BYTE locator.

Syntax
```plaintext
readFile(
    path STRING
)
```

1. `path` is the path the file to be loaded.

Usage
This method reads content from the specified file into the BYTE locator.
The bytes are loaded as is, without any conversion.
If the file is not found or if it cannot be read, the error -8087 is raised.

Example
```plaintext
MAIN
    DEFINE b BYTE
    LOCATE b IN MEMORY
    CALL b.readFile("mydata")
END MAIN
```

BYTE.writeFile
Writes the content of a BYTE to a file.

Syntax
```plaintext
writeFile(
    path STRING
)
```

1. `path` is the file to be written to.

Usage
This method writes the content of the current BYTE locator to the specified file.
The bytes are written as is, without any conversion.
If the file cannot be written, the error -8087 is raised.

Example
```plaintext
MAIN
    DEFINE b BYTE
    DATABASE mydb
    LOCATE b IN MEMORY
    SELECT col_byte INTO b FROM tab1
    CALL b.writeFile("mydata")
END MAIN
```

STRING data type as class
The STRING primitive data type provides a set of utility methods to manipulate character strings.
STRING methods can be invoked with a variable, for example:
```plaintext
DEFINE s STRING
    IF s.equalsIgnoreCase("pink") THEN
```
### Related concepts

**STRING** on page 305
The **STRING** data type is a variable-length, dynamically allocated character string data type, without limitation.
## STRING data type methods

### Table 505: Object methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>append(str STRING)</code></td>
<td>Concatenates a string.</td>
</tr>
<tr>
<td><code>equals(str STRING)</code></td>
<td>Compares a string to the content of a string variable.</td>
</tr>
<tr>
<td><code>equalsIgnoreCase(str STRING)</code></td>
<td>Makes a case-insensitive string comparison.</td>
</tr>
<tr>
<td><code>getCharAt(index INTEGER)</code></td>
<td>Returns the character at the specified position.</td>
</tr>
<tr>
<td><code>getIndexOf(str STRING, startIndex INTEGER)</code></td>
<td>Returns the position of a substring.</td>
</tr>
<tr>
<td><code>getLength()</code></td>
<td>Returns the length of the current string.</td>
</tr>
<tr>
<td><code>subString(startIndex INTEGER, endIndex INTEGER)</code></td>
<td>Returns a substring from start and end positions in a given string.</td>
</tr>
<tr>
<td><code>toLowerCase()</code></td>
<td>Returns the string converted to lower case.</td>
</tr>
<tr>
<td><code>toUpperCase()</code></td>
<td>Returns the string converted to upper case.</td>
</tr>
<tr>
<td><code>trim()</code></td>
<td>Removes leading and trailing blank space (ASCII 32) characters.</td>
</tr>
<tr>
<td><code>trimWhiteSpace()</code></td>
<td>Removes leading and trailing whitespace characters.</td>
</tr>
<tr>
<td><code>trimLeft()</code></td>
<td>Removes leading blank space (ASCII 32) characters.</td>
</tr>
<tr>
<td><code>trimLeftWhiteSpace()</code></td>
<td>Removes leading whitespace characters.</td>
</tr>
<tr>
<td><code>trimRight()</code></td>
<td>Removes trailing blank space (ASCII 32) characters.</td>
</tr>
</tbody>
</table>
**STRING.append**
Concatenates a string.

**Syntax**

```plaintext
append ( str STRING )
RETURNS STRING
```

1. *str* is the string to be concatenated.

**Usage**
This method concatenates a string to the current STRING variable and returns the resulting string.

The original STRING variable is not modified.

Appending a NULL will have no effect: the original string is returned.

**Example**

```plaintext
MAIN
  DEFINE s STRING
  LET s = "Some text"
  DISPLAY s.append("... more text")
END MAIN
```

Output:

```
Some text... more text
```

**STRING.equals**
Compares a string to the content of a string variable.

**Syntax**

```plaintext
equals ( str STRING )
RETURNS BOOLEAN
```

1. *str* is the string to compare with.

**Usage**
This method compares a string to the current STRING variable and returns TRUE if both strings match.

If the original STRING variable or the string passed as parameter is NULL, the result is FALSE.

**Example**

```plaintext
MAIN
  DEFINE s STRING
  LET s = "white"
  IF s.equals("white") THEN
    DISPLAY "Matches"
  END IF
END MAIN
```
**STRING.equalsIgnoreCase**

Makes a case-insensitive string comparison.

**Syntax**

```
equalsIgnoreCase( str STRING )
RETURNS BOOLEAN
```

1. `str` is the string to compare with.

**Usage**

This method compares a string to the current `STRING` variable by ignoring the character case, and returns `TRUE` if both strings match.

If the original `STRING` variable or the string passed as parameter is `NULL`, the result will be `FALSE`.

**Example**

```
MAIN
  DEFINE s STRING
  LET s = "white"
  IF s.equalsIgnoreCase("WHITE") THEN
    DISPLAY "Matches"
  END IF
END MAIN
```

**STRING.getCharAt**

Returns the character at the specified position.

**Syntax**

```
getCharAt( index INTEGER )
RETURNS CHAR(1)
```

1. `index` is the position of the character int the string.

**Usage**

This method extracts the character at the specified position from the `STRING` variable.

If the `STRING` variable is `NULL`, or if the `position` is out of the bounds of the string, the result will be `NULL`.

**Important**: When using byte length semantics, the position is expressed in bytes, and when using char length semantics, position is specified in characters. In byte length semantics, the method returns `NULL` if the position does not match a valid character-byte index in the current string.

**Example**

```
MAIN
  DEFINE s STRING
  LET s = "Some text"
  DISPLAY s.getCharAt(4)
END MAIN
```

Output:

```
e
```
Related concepts
Length semantics settings on page 521

STRING.getIndexOf
Returns the position of a substring.

Syntax

```java
getIndexOf(
    str STRING,
    startIndex INTEGER
) RETURNS INTEGER
```

1. `str` is the substring to be searched.
2. `startIndex` is the starting position for the search.

Usage
This method scans a STRING variable to find the substring passed as parameter, and returns the position of the substring.

The method starts the search for the substring at the starting position specified by the second parameter.

The method returns zero if:
• The STRING variable is NULL.
• The `str` substring was not found.
• The `str` substring is NULL.
• The start position is out of bounds.

Important: When using byte length semantics, the position is expressed in bytes, and when using char length semantics, it is specified in characters.

Example

```java
MAIN
    DEFINE s STRING
    LET s = "Some text"
    DISPLAY s.indexOf("text",1)
END MAIN
```

Output:

6

Related concepts
Length semantics settings on page 521

STRING.getLength
Returns the length of the current string.

Syntax

```java
getLength() RETURNS INTEGER
```

Usage
This method counts the number of bytes or characters in a STRING variable.
Note: Unlike the LENGTH() function, the getLength() method counts the trailing blanks. If the STRING variable is NULL, the method returns zero.

Important: When using byte length semantics, the length is expressed in bytes, and when using char length semantics, it is expressed in characters.

Example

```plaintext
MAIN
  DEFINE s STRING
  LET s = "Some text"
  DISPLAY s.getLength()
END MAIN
```

Output:

```
9
```

Related concepts

Length semantics settings on page 521

**STRING.subString**

Returns a substring from start and end positions in a given string.

Syntax

```
subString(
  startIndex INTEGER,
  endIndex INTEGER )
RETURNS STRING
```

1. *startIndex* is the starting position of the substring.
2. *endIndex* is the ending position of the substring.

Usage

This method returns a substring of the current STRING variable based on start and end positions in the original string. If the STRING variable is NULL, or when the positions are out of bounds, the method returns NULL.

Important: When using byte length semantics, the positions are expressed in bytes, and when using char length semantics, positions are expressed in characters.

Example

```plaintext
MAIN
  DEFINE s STRING
  LET s = "1234567890"
  DISPLAY s.subString(5,7)
END MAIN
```

Output:

```
567
```

Related concepts

Length semantics settings on page 521
STRING.toLowerCase
Returns the string converted to lower case.

Syntax

```
toLowerCase( )
    RETURNS STRING
```

Usage
This method converts the current STRING variable to lower case and returns the resulting string.
If the original STRING variable is NULL, the result is NULL.

Example

```
MAIN
    DEFINE s STRING
    LET s = "SOME TEXT"
    DISPLAY s.toLowerCase()
END MAIN
```

Output:

```
some text
```

STRING.toUpperCase
Returns the string converted to upper case.

Syntax

```
toUpperCase( )
    RETURNS STRING
```

Usage
This method converts the current STRING variable to upper case and returns the resulting string.
If the original STRING variable is NULL, the result is NULL.

Example

```
MAIN
    DEFINE s STRING
    LET s = "Some text"
    DISPLAY s.toUpperCase()
END MAIN
```

Output:

```
SOME TEXT
```

STRING.trim
Removes leading and trailing blank space (ASCII 32) characters.

Syntax

```
trim( )
```
Returns String

Usage
The `trim()` method removes the leading and trailing blank space characters of the current `STRING` variable and returns a new string.

**Note:** The method removes only blank space (ASCII(32)) characters. Characters tab (\t), newline (\n), carriage-return (\r) and form-feed (\f) are not removed.

If the original `STRING` variable is NULL, the result will be NULL.

Example
```main
DEFINE s STRING
LET s = "   Some text   "
DISPLAY "["||s.trim()||"]"
END MAIN
```

Output:

[Some text]

Related concepts
- `STRING.trimWhiteSpace` on page 2388
  Removes leading and trailing whitespace characters.

`STRING.trimWhiteSpace`
Removes leading and trailing whitespace characters.

Syntax
```
trimWhiteSpace()
RETURNS STRING
```

Usage
The `trimWhiteSpace()` method removes the leading and trailing whitespace characters of the current `STRING` variable and returns a new string.

**Note:** The method considers as whitespace characters all characters less than or equal to blank space (ASCII(32)). This includes tab (\t), newline (\n), carriage-return (\r) and form-feed (\f).

If the original `STRING` variable is NULL, the result will be NULL.

Example
```main
DEFINE s STRING
LET s = "\n\t Some text \n\t"
DISPLAY "["||s.trimWhiteSpace()||"]"
END MAIN
```

Output:

[Some text]

Related concepts
- `STRING.trim` on page 2387
Removes leading and trailing blank space (ASCII 32) characters.

**STRING.trimLeft**
Removes leading blank space (ASCII 32) characters.

**Syntax**

```
trimLeft( )
RETURNS STRING
```

**Usage**

The `trimLeft()` method removes the leading blank space characters of the current STRING variable and returns a new string.

**Note:** The method removes only blank space (ASCII (32)) characters. Characters tab (\t), newline (\n), carriage-return (\r) and form-feed (\f) are not removed.

If the original STRING variable is NULL, the result will be NULL.

**Example**

```
MAIN
  DEFINE s STRING
  LET s = "     Some text"
  DISPLAY "[\"||s.trimLeft()||\"]"
END MAIN
```

Output:

```
[Some text]
```

**Related concepts**

**STRING.trimLeftWhiteSpace** on page 2389
Removes leading whitespace characters.

**STRING.trimLeftWhiteSpace**
Removes leading whitespace characters.

**Syntax**

```
trimLeftWhiteSpace( )
RETURNS STRING
```

**Usage**

The `trimLeftWhiteSpace()` method removes the leading whitespace characters of the current STRING variable and returns a new string.

**Note:** The method considers as whitespace characters all characters less than or equal to blank space (ASCII (32)). This includes tab (\t), newline (\n), carriage-return (\r) and form-feed (\f).

If the original STRING variable is NULL, the result will be NULL.

**Example**

```
MAIN
  DEFINE s STRING
  LET s = "\n\t     Some text"
  DISPLAY "[\"||s.trimLeftWhiteSpace()||\"]"
```

Related concepts

**STRING.trimLeft** on page 2389
Removes leading blank space (ASCII 32) characters.

**STRING.trimRight**
Removes trailing blank space (ASCII 32) characters.

**Syntax**

```plaintext
trimRight() 
RETURNS STRING
```

**Usage**

The `trimRight()` method removes the trailing blank space characters of the current `STRING` variable and returns a new string.

**Note:** The method removes only blank space (ASCII 32) characters. Characters tab (\t), newline (\n), carriage-return (\r) and form-feed (\f) are not removed.

If the original `STRING` variable is `NULL`, the result will be `NULL`.

**Example**

**MAIN**

```plaintext
DEFINE s STRING 
LET s = "Some text     " 
DISPLAY "["||s.trimRight()||"]"
END MAIN
```

**Output:**

```plaintext
[Some text]
```

**Related concepts**

**STRING.trimRightWhiteSpace** on page 2390
Removes trailing whitespace characters.

**STRING.trimRightWhiteSpace**
Removes trailing whitespace characters.

**Syntax**

```plaintext
trimRightWhiteSpace() 
RETURNS STRING
```

**Usage**

The `trimRightWhiteSpace()` method removes the trailing whitespace characters of the current `STRING` variable and returns a new string.
Note: The method considers as whitespace characters all characters less than or equal to blank space (ASCII (32)). This includes tab (\t), newline (\n), carriage-return (\r) and form-feed (\f).

If the original STRING variable is NULL, the result will be NULL.

Example

```plaintext
MAIN
  DEFINE s STRING
  LET s = "Some text   
          \n          \t"
  DISPLAY "["||s.trimRightWhiteSpace()||"]"
END MAIN
```

Output:

```
[Some text]
```

Related concepts

STRING.trimRight on page 2390
Removes trailing blank space (ASCII 32) characters.

TEXT data type as class

The TEXT primitive data type provides a set of utility methods to manipulate TEXT data.

TEXT methods can be invoked with a variable, for example:

```plaintext
DEFINE t TEXT
  ...
  CALL t.writeFile("mydata")

Related concepts

TEXT on page 307
The TEXT data type stores large text data.

TEXT data type methods

Table 506: Object methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getTextLength()</td>
<td>Returns the length of TEXT content.</td>
</tr>
<tr>
<td></td>
<td>RETURNS INTEGER</td>
</tr>
<tr>
<td>readFile(path STRING)</td>
<td>Reads a file into a TEXT locator.</td>
</tr>
<tr>
<td>writeFile(path STRING)</td>
<td>Writes the content of TEXT type to a file.</td>
</tr>
</tbody>
</table>

TEXT.getTextLength
Returns the length of TEXT content.

Syntax

```plaintext
getTextLength()
  RETURNS INTEGER
```
Usage
This method returns the number of bytes in TEXT data.

**Important:** This method always returns a number of bytes, even when using character length semantics.

Example

```plaintext
MAIN
  DEFINE t TEXT
  LOCATE t IN MEMORY
  LET t = "aaaaaaaaaaaaa"
  DISPLAY t.getLength() -- Shows 13
END MAIN
```

Related concepts

- **Length semantics settings** on page 521

**TEXT.readFile**

Reads a file into a TEXT locator.

Syntax

```plaintext
readFile(
  path STRING )
```

1. `path` is the path the file to be loaded.

Usage

This method reads content from the specified file into a TEXT locator.

If the file is not found or if it cannot be read, the error -8087 is raised.

**Important:** The character set used in the file must match the current application locale.

**Note:** Files encoded in UTF-8 can start with the UTF-8 Byte Order Mark (BOM), a sequence of 0xEF 0xBB 0xBF bytes, also known as UNICODE U+FEFF. When reading files, Genero BDL will ignore the UTF-8 BOM, if it is present at the beginning of the file. This applies to instructions such as LOAD, as well as I/O APIs such as base.Channel.read() and readLine().

Example

```plaintext
MAIN
  DEFINE t TEXT
  LOCATE t IN MEMORY
  CALL t.readFile("mydata")
END MAIN
```

**TEXT.writeFile**

Writes the content of TEXT type to a file.

Syntax

```plaintext
writeFile(
  path STRING )
```

1. `path` is the file to be written to.
**Usage**

This method writes the content of the current TEXT locator to the specified file.

If the file cannot be written, the error -8087 is raised.

**Important:** The character set used in the file must match the current application locale.

**Example**

```plaintext
MAIN
   DEFINE t TEXT
   LOCATE t IN MEMORY
   SELECT col_text INTO t FROM ...
   CALL t.writeFile("mydata")
END MAIN
```

**DYNAMIC ARRAY as class**

The DYNAMIC ARRAY (or static ARRAY) type provides a set of utility methods to manipulate array elements.

DYNAMIC ARRAY methods can be invoked with a variable, for example:

```plaintext
DEFINE a DYNAMIC ARRAY OF STRING
CALL a.appendElement()
DISPLAY a.getLength()
```

**Related concepts**

Arrays on page 418
Arrays (static or dynamic) allow you to handle an ordered collection of elements.

**DYNAMIC ARRAY methods**

**Table 507: Object methods**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>appendElement ( )</strong></td>
<td>Adds a new element to the end of the array.</td>
</tr>
<tr>
<td><strong>clear ( )</strong></td>
<td>Removes all elements of the array.</td>
</tr>
<tr>
<td><strong>copyTo ( dst dynamic-array-type )</strong></td>
<td>Copies a complete array to the destination array passed as parameter.</td>
</tr>
<tr>
<td><strong>deleteElement ( index INTEGER )</strong></td>
<td>Removes an element from the array.</td>
</tr>
<tr>
<td><strong>getLength ( )</strong></td>
<td>Returns the length of the array.</td>
</tr>
<tr>
<td><strong>insertElement ( index INTEGER )</strong></td>
<td>Inserts a new element at the given index.</td>
</tr>
<tr>
<td><strong>search ( key STRING, value STRING )</strong></td>
<td>Scans the array to find an element that matches the search parameter.</td>
</tr>
<tr>
<td><strong>searchRange ( key STRING, value STRING, from INTEGER, to INTEGER )</strong></td>
<td>Scans the array to find an element that matches the search parameter.</td>
</tr>
<tr>
<td><strong>sort ( key STRING, reverse BOOLEAN )</strong></td>
<td>Sorts the rows in the array.</td>
</tr>
</tbody>
</table>

**DYNAMIC ARRAY.appendElement**

Adds a new element to the end of the array.

**Syntax**

```
appendElement ( )
```

**Usage**

This method creates a new element at the end of the array.

The element is initialized to **NULL**.

**Example**

```
MAIN
  DEFINE a DYNAMIC ARRAY OF INTEGER
  DEFINE x INT
```
FOR x=1 TO 5
    CALL a.appendElement()
    LET a[x] = 100+x
END FOR
DISPLAY a.getLength() -- shows 5
DISPLAY a[3] -- shows 103
END MAIN

Since element allocation occurs automatically for dynamic arrays, you can omit the call to the `appendElement()` method and directly assign the new element:

MAIN
    DEFINE a DYNAMIC ARRAY OF INTEGER
    LET a[100] = 87234 -- Array gets a length of 100 automatically
    LET a[101] = 98562 -- New element at position 101
END MAIN

However, for better code readability, you might want to use the `appendElement()` method in some cases.

**DYNAMIC ARRAY.clear**
Removes all elements of the array.

### Syntax

```
clear( )
```

### Usage

This method clears the array, by removing all its elements.

For example, if the array is not empty, use the `clear()` method just before filling the array with a new set of elements.

### Example

```
MAIN
    DEFINE a1 DYNAMIC ARRAY OF STRING
    CALL fill_array(a1)
    DISPLAY a1.getLength()
    DISPLAY a1[3]
END MAIN

FUNCTION fill_array(arr)
    DEFINE arr DYNAMIC ARRAY OF STRING
    DEFINE i INTEGER
    CALL arr.clear()
    FOR i=1 TO 10
        LET arr[i] = "Item #"||i
    END FOR
END FUNCTION
```

**DYNAMIC ARRAY.copyTo**
Copies a complete array to the destination array passed as parameter.

### Syntax

```
copyTo( dst dynamic-array-type )
```

1. `dst` is the destination dynamic array, with the same type definition as the source array.
2. `dynamic-array-type` is a `DYNAMIC ARRAY` of `...` type that corresponds to the source array.
**Usage**

The `copyTo()` method copies all elements of the source array into the destination array. The method truncates the length of the destination array to the length of the source array. The `copyTo()` method is equivalent to the following code:

```plaintext
CALL destin.clear()
FOR x=1 TO source.getLength()
    LET destin[x].* = source[x].*
END FOR
```

If the destination array is not of the same type as the source array, error -8112 is thrown.

**Important:** Avoid making a copy of huge arrays, consider using a database temporary table instead.

**Example**

```plaintext
MAIN
    TYPE t_names RECORD
        num INT,
        name STRING
    END RECORD
    DEFINE aa, ab DYNAMIC ARRAY OF t_names
    LET aa[1].num = 101
    LET aa[1].name = "Mike Tormme"
    CALL aa.copyTo(ab)
    DISPLAY ab[1].*
END MAIN
```

**DYNAMIC ARRAY.deleteElement**

Removes an element from the array.

**Syntax**

```plaintext
deleteElement( index INTEGER )
```

1. `index` is the position of the element to be removed.

**Usage**

This method removes the array element at the specified index.

**Note:** No error is raised if the index is out of bounds.

**Example**

```plaintext
MAIN
    DEFINE a DYNAMIC ARRAY OF INTEGER
    LET a[10] = 9
    CALL a.deleteElement(5)
    DISPLAY a.getLength() -- shows 9
    DISPLAY a[9] -- shows 9
END MAIN
```

**DYNAMIC ARRAY.getLength**

Returns the length of the array.

**Syntax**

```plaintext
getLength( )
```
**Usage**

This method returns the number of elements in the array.

**Tip:** To improve performances in FOR ... END FOR loops, assign a variable with the array length and use this variable as argument for the TO clause.

**Example**

```plaintext
MAIN
    DEFINE arr DYNAMIC ARRAY OF STRING,
    i, m INTEGER
    LET arr[1] = "aaa"
    LET arr[2] = "bbbb"
    LET arr[3] = "cccc"
    LET m = arr.getLength()
    FOR i=1 TO m
        DISPLAY arr[i]
    END FOR
END MAIN
```

**DYNAMIC ARRAY.insertElement**

Inserts a new element at the given index.

**Syntax**

```plaintext
insertElement ( index INTEGER )
```

1. `index` is the position where a new element must be inserted.

**Usage**

This method inserts a new element in the array, before the specified index.

**Note:** No error is raised if the index is out of bounds.

**Example**

```plaintext
MAIN
    DEFINE a DYNAMIC ARRAY OF INTEGER
    LET a[10] = 11
    CALL a.insertElement(10)
    LET a[10] = 10
    DISPLAY a.getLength() -- shows 11
    DISPLAY a[10] -- shows 10
END MAIN
```

**DYNAMIC ARRAY.search**

Scans the array to find an element that matches the search parameter.

**Syntax**

```plaintext
search ( key STRING, value STRING )
```

1. `key` is the name of the RECORD member when the array is structured.
2. `value` is the value to look for.
Usage

The `search()` method will scan the whole array, to find a specific element that matches the value passed as second parameter.

The method returns the index of the first occurrence found.

The method returns zero, if no matching element is found.

The `search()` method is equivalent to the following function (same as `arr.search("member", val)`):

```plaintext
FUNCTION search(arr, val)
    DEFINE arr DYNAMIC ARRAY OF...
    val STRING
    DEFINE x INT
    FOR x=1 TO arr.getLength()
        IF arr[x].member = value THEN RETURN x END IF
    END FOR
    RETURN 0
END FUNCTION
```

If the array is structured (DYNAMIC ARRAY OF RECORD), the method returns the index of the first occurrence of an element where the record member specified by `key` is equal to the `value` parameter.

If the array is a flat array (DYNAMIC ARRAY OF base-type), the method returns the index of the first occurrence of the element that matches the `value` parameter. The `key` parameter is ignored.

Example

```plaintext
MAIN
    DEFINE a DYNAMIC ARRAY OF RECORD
        name STRING
    END RECORD
    LET a[1].name = "Mike"
    LET a[2].name = "Phil"
    LET a[3].name = "John"
    DISPLAY a.search("name", "Marc")  -- Shows 0
    DISPLAY a.search("name", "John")  -- Shows 3
END MAIN
```

Related concepts

`DYNAMIC ARRAY.searchRange` on page 2398
Scans the array to find an element that matches the search parameter.

`DYNAMIC ARRAY.searchRange`  
Scans the array to find an element that matches the search parameter.

Syntax

```plaintext
searchRange( key STRING, value STRING, from INTEGER, to INTEGER )
RETURNS INTEGER
```

1. `key` is the name of the RECORD member when the array is structured.
2. `value` is the value to look for.
3. `from` is the start index (inclusive).
4. `to` is the end index (inclusive).
Usage

The `searchRange()` method is similar to the `search()` method, except that array rows are scanned from a given index to an ending index.

The method returns the index of the first occurrence found in the specified range.

The method returns zero, if no matching element is found in the specified range.

If the start index is lower than 1, the search starts at index 1.

If the end index is lower than the start index, the method returns zero.

If the end index is greater than the array length, it is ignored and the search is done until the last element of the array.

Example

```main
DEFINE a DYNAMIC ARRAY OF RECORD
  name STRING
END RECORD

LET a[1].name = "Mike"
LET a[2].name = "Phil"
LET a[3].name = "John"
LET a[4].name = "Phil"

DISPLAY a.searchRange("name", "John", 1, 2) -- Shows 0
DISPLAY a.searchRange("name", "Phil", 1, 2) -- Shows 2
DISPLAY a.searchRange("name", "Phil", 3, 4) -- Shows 4
DISPLAY a.searchRange("name", "Phil", 0, 10) -- Shows 2
END MAIN
```

Related concepts

`DYNAMIC ARRAY.search` on page 2397
Scans the array to find an element that matches the search parameter.

`DYNAMIC ARRAY.sort`
Sorts the rows in the array.

Syntax

```
sort ( key STRING, reverse BOOLEAN )
```

1. `key` is the name of a member of a structured array (`DYNAMIC ARRAY OF RECORD`), or NULL if the array is not structured.
2. `reverse` is FALSE for ascending order, TRUE for descending order.

Usage

A dynamic array can be sorted with the `sort()` method.

- For non-structured dynamic arrays (`DYNAMIC ARRAY OF simple-type`), the first argument of `sort()` must be NULL. The array will be sorted by the single-typed elements.
- With structured arrays (`DYNAMIC ARRAY OF RECORD`), this method sorts the array by the member passed as first parameter.

Use the second parameter to define the sort order as ascending (FALSE) or descending (TRUE).

Note: Character string data is sorted depending on the current application locale. The `sort()` method uses the operating system collation functions. Note that collation rules can slightly differ from platform to platform and therefore produce different ordering.
When making subsequent calls to the `sort()` method using different record members of the array, the rows will be ordered by all of the record members specified for the cumulative sorts, with the most recent call defining the main sort field.

Another way to think of this is in terms of the `ORDER BY` clause of a SQL statement: If your dynamic array contained the variables A, B and C, and you included the following calls to the `sort()` method:

```plaintext
CALL a.sort("C",false)
CALL a.sort("B",false)
CALL a.sort("A",false)
```

This would be equivalent to writing an `ORDER BY` clause that states:

```plaintext
ORDER BY A, B, C
```

**Example**

In this example, the first call to the `sort()` method sorts the rows by name in ascending order, and the second call will sort the rows by key in descendant order, then by name within each key. The last sort becomes the main sort field.

```plaintext
MAIN
  DEFINE a DYNAMIC ARRAY OF RECORD
    key INTEGER,
    name VARCHAR(30)
  END RECORD
  LET a[1].key = 776236    LET a[1].name = "aaaaa"
  LET a[2].key = 273434    LET a[2].name = "cccccccc"
  LET a[3].key = 934092    LET a[3].name = "bbbb"
  CALL a.sort("name",FALSE) -- Sorted by name (asc order)
  CALL a.sort("key",TRUE) -- Sorted by key (desc), then by name
END MAIN
```

**DICTIONARY as class**

The `DICTIONARY` type provides a set of utility methods to manipulate the dictionary elements.

`DICTIONARY` methods can be invoked with the variable, for example:

```plaintext
DEFINE d DICTIONARY OF STRING
CALL d.clear()
DISPLAY d.getLength()
```

**Related concepts**

[Dictionaries](#) on page 426
A dictionary holds an unordered collection of elements accessed by a key.

**DICTIONARY methods**

**Table 508: Object methods**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear ( )</td>
<td>Removes all elements of the dictionary.</td>
</tr>
<tr>
<td>contains ( key STRING )</td>
<td>Checks if an element with the given key exists in the dictionary.</td>
</tr>
<tr>
<td>copyTo ( dst dictionary-type )</td>
<td>Copies all elements of the dictionary into another dictionary.</td>
</tr>
<tr>
<td>getKeys ( )</td>
<td>Returns a dynamic array of all keys of the dictionary.</td>
</tr>
<tr>
<td>getLength ( )</td>
<td>Returns the number of elements in the dictionary.</td>
</tr>
<tr>
<td>remove ( key STRING )</td>
<td>Removes an element of the dictionary identified by the key.</td>
</tr>
</tbody>
</table>

**DICTIONARY.clear**

Removes all elements of the dictionary.

**Syntax**

```plaintext
clear ( )
```

**Usage**

The `clear()` method cleans up the dictionary, by removing all its elements.

For example, if the dictionary is not empty, use the `clear()` method just before filling the dictionary with a new set of elements.

**Example**

```plaintext
FUNCTION fill_dictionary(dict)
   DEFINE dict DICTIONARY OF STRING
   CALL dict.clear()
   LET dict["abc"] = 111
   LET dict["def"] = 222
   LET dict["xyz"] = 999
END FUNCTION
```
**DICTIONARY.contains**
Checks if an element with the given key exists in the dictionary.

**Syntax**

```plaintext
contains( key STRING )
RETURNS BOOLEAN
```

1. `key` is the dictionary key to check.

**Usage**
The `contains()` method scans the dictionary to find if there is an element with the key passed as parameter. The method returns TRUE if the element exists, or FALSE if the element is not found.

**Example**

```
MAIN
  DEFINE dict DICTIONARY OF STRING
  LET dict["abc"] = 111
  LET dict["def"] = 222
  LET dict["xyz"] = 999
  DISPLAY dict.contains("def") -- shows 1 (TRUE)
  DISPLAY dict.contains("zzz") -- shows 0 (FALSE)
END MAIN
```

**DICTIONARY.copyTo**
Copies all elements of the dictionary into another dictionary.

**Syntax**

```plaintext
copyTo( dst dictionary-type )
```

1. `dst` is the destination dictionary, with the same type definition as the source dictionary.
2. `dictionary-type` is a DICTIONARY OF ... type that corresponds to the source dictionary.

**Usage**
The `copyTo()` method clones the complete dictionary into the destination dictionary passed as parameter. The destination dictionary will be cleared before the copy operation starts.

If the destination dictionary is not of the same type as the source dictionary, error -8112 is thrown.

**Important:** Avoid making a copy of huge dictionaries, consider using a database temporary table instead.

**Example**

```
MAIN
  TYPE t_contact RECORD
      name STRING,
      address STRING,
      birth DATE
  END RECORD
  DEFINE d1, d2 DICTIONARY OF t_contact
  LET d1["Kirk"].name = "James T. Kirk"
  LET d1["Kirk"].address = "Riverside, Iowa"
  LET d1["Spock"].name = "Spock"
  LET d1["Spock"].address = "Shi'Kahr"
```

CALL d1.copyTo(d2)
DISPLAY d2.getLength()
DISPLAY d2["Spock"].*
END MAIN

**DICTIONARY.getLength**

Returns the number of elements in the dictionary.

**Syntax**

```
getLength()  
RETURNS INTEGER
```

**Usage**

The `getLength()` method returns the total number of elements in the dictionary.

**Example**

```
MAIN
  DEFINE dict DICTIONARY OF STRING
  LET dict["abc"] = 111
  LET dict["def"] = 222
  LET dict["xyz"] = 999
  DISPLAY dict.getLength() -- shows 3
END MAIN
```

**DICTIONARY.getKeys**

Returns a dynamic array of all keys of the dictionary.

**Syntax**

```
getKeys()  
RETURNS DYNAMIC ARRAY OF STRING
```

**Usage**

The `getKeys()` method builds and returns a `DYNAMIC ARRAY OF STRING` with all keys contained in the dictionary.

It is then possible to scan the dictionary with the keys returned by this method.

**Note:** A dictionary is an unordered list of elements. The keys might not be returned in the same order as they have been added to the dictionary.

**Example**

```
MAIN
  DEFINE dict DICTIONARY OF STRING,
      keys DYNAMIC ARRAY OF STRING,
      i INTEGER
  LET dict["first"] = "abc"
  LET dict["second"] = "def"
  LET dict["last"] = "xyz"
  LET keys = dict.getKeys()
  FOR i = 1 to keys.getLength()
    DISPLAY i, dict[keys[i]]
```
**DICTIONARY.remove**

Removes an element of the dictionary identified by the key.

**Syntax**

```java
remove(key STRING )
```

1. `key` is the dictionary key of the element to remove.

**Usage**

The `remove()` method deletes and element of the dictionary, identified by the key passed as parameter.

**Example**

```java
MAIN
DEFINE dict DICTIONARY OF STRING
LET dict["abc"] = 111
CALL dict.remove("abc")
DISPLAY dict.contains("abc") -- shows 0 (FALSE)
END MAIN
```

**Java Array type as class**

The Java Array type provides a set of utility methods to manipulate array elements.

Java array methods can be invoked with a type reference or the array variable, for example:

```java
IMPORT JAVA java.lang.String
MAIN
TYPE string_array_type ARRAY[] OF java.lang.String
DEFINE names string_array_type
LET names = string_array_type.create(100)
LET names[1] = "aaaaaaa"
DISPLAY names.getLength()  
END MAIN
```

**Related concepts**

- The **Java interface** on page 2192
  - The **Java interface** allows you to import Java classes and instantiate Java objects in your programs.
- **ARRAY** on page 419
  - An array defines a vector variable with a list of elements.

**Java Array type methods**

**Table 509: Class methods**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>java-array-type.create(size INTEGER )</code></td>
<td>Creates a new Java array of the given type.</td>
</tr>
</tbody>
</table>
Table 510: Object methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getLength()</code></td>
<td>Returns the length of the Java array.</td>
</tr>
<tr>
<td></td>
<td>RETURNS INTEGER</td>
</tr>
</tbody>
</table>

`java-array-type.create`
Creates a new Java array of the given type.

**Syntax**

```
java-array-type.create( size INTEGER )
RETURNS java-array-type
```

1. `size` defines the actual number of elements of the array.

**Usage**

This class method creates a new instance of the Java array specified by the type used, with the size provided as parameter.

The type must be declared as a user defined type define with the `ARRAY [] OF` notation reserved for Java arrays.

**Example**

```java
IMPORT JAVA java.lang.String
MAIN
    TYPE string_array_type ARRAY[] OF java.lang.String
    DEFINE names string_array_type
    LET names = string_array_type.create(100)
    LET names[1] = "aaaaaaa"
    DISPLAY names[1]
END MAIN
```

**Related concepts**

*Types* on page 431
Types can be defined by the programmer to centralize the definition of complex/structured variables.

`java-array.getLength`
Returns the length of the Java array.

**Syntax**

```
getLength( )
RETURNS INTEGER
```

**Usage**

This method returns the number of elements in the Java array.

**Example**

```java
IMPORT JAVA java.lang.String
MAIN
    TYPE string_array_type ARRAY[] OF java.lang.String
    DEFINE names string_array_type
    LET names = string_array_type.create(100)
```
The base package

These topics cover the built-in classes of the base package

The Application class

The `base.Application` class provides a set of utility functions related to the program environment.

Command line arguments, execution directory and FGLPROFILE resource entries are some of the elements you can query with this class.

This class is built-in and can be used directly in the source code.

This class does not have to be instantiated. It provides class methods for the current program.
### base.Application methods

#### Table 511: Class methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>base.Application.getArgument(index INTEGER)</code></td>
<td>Returns the command line argument by position.</td>
</tr>
<tr>
<td><code>base.Application.getArgumentCount()</code></td>
<td>Returns the total number of command line arguments.</td>
</tr>
<tr>
<td><code>base.Application.getProgramDir()</code></td>
<td>Returns the directory path of the current program.</td>
</tr>
<tr>
<td><code>base.Application.getProgramName()</code></td>
<td>Returns the name of the current program.</td>
</tr>
<tr>
<td><code>base.Application.getFglDir()</code></td>
<td>Returns the path to the FGLDIR installation directory.</td>
</tr>
<tr>
<td><code>base.Application.getResourceEntry(name STRING)</code></td>
<td>Returns the value of a FGLPROFILE entry.</td>
</tr>
<tr>
<td><code>base.Application.getStackTrace()</code></td>
<td>Returns the function call stack trace.</td>
</tr>
<tr>
<td><code>base.Application.isMobile()</code></td>
<td>Indicates if the application runs on a mobile device.</td>
</tr>
<tr>
<td><code>base.Application.reloadResources(newResourcePath STRING)</code></td>
<td>Resets FGLRESOURCEPATH and reloads localized string resources.</td>
</tr>
</tbody>
</table>

#### `base.Application.getArgument`

Returns the command line argument by position.

#### Syntax

```java
base.Application.getArgument(index INTEGER)
RETURNS STRING
```

1. `index` is the index of the program argument.
**Usage**

The index is the program argument position. The first program argument is identified by the position 1. Argument number zero is the program name.

Returns NULL if there is no argument provided at the position.

**Example**

```plaintext
MAIN
    DEFINE i INTEGER
    FOR i=1 TO base.Application.getArgumentCount()
        DISPLAY base.Application.getArgument(i)
    END FOR
END MAIN
```

*base.Application.getArgumentCount*

Returns the total number of command line arguments.

**Syntax**

```plaintext
base.Application.getArgumentCount ()
    RETURNS INTEGER
```

**Usage**

Returns the total number of command line arguments, can be used to scan the argument values with *base.Application.getArgument().*

*base.Application.getFglDir*

Returns the path to the FGLDIR installation directory.

**Syntax**

```plaintext
base.Application.getFglDir ()
    RETURNS STRING
```

**Usage**

The `getFglDir()` method returns the installation directory path defined by the FGLDIR environment variable. The directory path is system-dependent.

*base.Application.getProgramDir*

Returns the directory path of the current program.

**Syntax**

```plaintext
base.Application.getProgramDir ()
    RETURNS STRING
```

**Usage**

This method returns the directory path where the program file (42r) is located.

The directory path is system-dependent.

**Related concepts**

FGLAPPDIR on page 273
Contains the path to the application directory when executing on a mobile device.

**base.Application.getProgramName**

Returns the name of the current program.

### Syntax

```java
base.Application.getProgramName()
RETURNS STRING
```

### Usage

This method returns the name of the current program. This is the name of the 42m or 42r module without the file extension passed to fglrun.

**base.Application.getResourceEntry**

Returns the value of a FGLPROFILE entry.

### Syntax

```java
base.Application.getResourceEntry(name STRING )
RETURNS STRING
```

1. `name` is the name of a FGLPROFILE entry.

### Usage

The `base.Application.getResourceEntry()` method reads the FGLPROFILE file(s) and returns the value defined for the entry passed as parameter.

If the entry does not exist in the configuration file, the method returns NULL.

**Important:** FGLPROFILE entry names are not case sensitive.

If multiple entries are defined with the same name (this can happen especially when several profile files are defined in the FGLPROFILE environment variable), the last entry found wins.

The `fgl_getresource()` built-in function is equivalent to `base.Application.getResourceEntry()`.

### Example

```java
MAIN
   DISPLAY base.Application.getResourceEntry("mycompany.params.logmode")
END MAIN
```

### Related concepts

- **fgl_getresource()** on page 2287
  Returns the value of an FGLPROFILE entry.

**base.Application.getStackTrace**

Returns the function call stack trace.

### Syntax

```java
base.Application.getStackTrace()
RETURNS STRING
```
Usage

Use the `getStackTrace()` method, to print the stack trace to a log file.

**Important:** Sensitive and personal data may be written to the output. Make sure that the log output is written to files that can only be read by application administrators.

This method returns a string containing a formatted list of the current function stack.

**Note:** The output format of `getStackTrace()` method is for debug purpose only and can change in future product releases.

You typically use this function in a `WHENEVER ERROR CALL` handler.

```plaintext
MAIN
    WHENEVER ANY ERROR CALL my_handler
    CALL func1("abcdef",999)
END MAIN

FUNCTION func1(name,id)
    DEFINE name STRING, id INTEGER
    DEFINE r INTEGER
    LET r = div(5,0)
END FUNCTION

FUNCTION div(x,y)
    DEFINE x,y DECIMAL
    RETURN ( x / y )
END FUNCTION

FUNCTION my_handler()
    DISPLAY base.Application.getStackTrace()
END FUNCTION
```

Example of stack trace output:

```
#0 my_handler() at x.4gl:17
#1 div() at x.4gl:14
#2 func1() at x.4gl:9
#3 main() at x.4gl:3
```

`base.Application.isMobile`

Indicates if the application runs on a mobile device.

**Syntax**

```plaintext
base.Application.isMobile()
RETURNS BOOLEAN
```

**Usage**

This class method can be called to check if the program code is running on a smartphone or tablet device. The method will return `TRUE` if the program executes in standalone mode (meaning the runtime system is on a mobile device).

**Example**

```plaintext
MAIN
    MENU "test"
        COMMAND "check"
            MESSAGE SFMT("isMobile = %1", base.Application.isMobile())
    END MENU
```
**base.Application.reloadResources**
Resets FGLRESOURCEPATH and reloads localized string resources.

**Syntax**

```
base.Application.reloadResources(
    newResourcePath STRING)
```

1. `newResourcePath` is a list of directories to search for string resource files.

**Usage**

The `reloadResources()` method overwrites the search path defined by the FGLRESOURCEPATH environment variable, to find program resource files in directories that are only known at runtime.

**Warning:** The `reloadResources()` method, to reset the FGLRESOURCEPATH environment variable in order to find program resource files in a different directory, must only be used at the beginning of the program execution.

This method is typically used to define a search path for localized string files when a program starts, to let the end user change the current application language. It avoids starting a new application (via `RUN`), after the end user selects a language in a parent program.

The runtime behaves as if FGLRESOURCEPATH had been set to this value from the start.

**Note:** Pay attention to the path separator, which is specific to the operating system. See FGLRESOURCEPATH reference for more details.

The method does the following:

1. Resets the environment variable FGLRESOURCEPATH with the specified value.
2. Reloads already loaded string localization files (.42s)
3. Reloads the default action defaults file (default.4ad)

**Notes:**

- Reloading resources has no effect on displayed forms: only forms displayed after reloading resources will use the new reloaded strings.
- Presentation Styles (.4st) are not reloaded.

**Note:** The writing direction of a language/script is defined with the UserInterface.reverse presentation style attribute:

```
<Style name="UserInterface">
    <StyleAttribute name="reverse" value="yes" />
</Style>
```

Since presentation styles are not reloaded, it is not possible to switch between scripts having different writing directions. To change the writing direction, the program must be restarted.

- Reloading resources has no effect on .42m modules that are already loaded (any %"string" will not be localized again). For this reason, the `reloadResources()` method must be called at a very early stage of the program.

**Note:** The debugger (fglrun -d) loads all program modules immediately when starting. Therefore, reloading resources has no effect on localized strings in .42m modules when debugging.

Related concepts
Loading localized strings at runtime on page 544
Understand the rules for using localized strings at runtime.

The Channel class
The base.Channel class is a built-in class providing basic input/output functions.

base.Channel methods

Table 512: Class methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>base.Channel.create()</code></td>
<td>Returns <code>base.Channel</code>  Create a new channel object.</td>
</tr>
</tbody>
</table>
### Table 513: Object methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dataAvailable()</code></td>
<td>Tests if some data can be read from the channel.</td>
</tr>
<tr>
<td>close()</td>
<td>Closes the channel.</td>
</tr>
<tr>
<td><code>isEof()</code></td>
<td>Detect the end of a file.</td>
</tr>
<tr>
<td>openFile({</td>
<td>Opens a file channel.</td>
</tr>
<tr>
<td><code>path</code> STRING,</td>
<td></td>
</tr>
<tr>
<td><code>mode</code> STRING })</td>
<td></td>
</tr>
<tr>
<td>openPipe({</td>
<td>Opens a pipe channel to a subprocess.</td>
</tr>
<tr>
<td><code>command</code> STRING,</td>
<td></td>
</tr>
<tr>
<td><code>mode</code> STRING })</td>
<td></td>
</tr>
<tr>
<td>openClientSocket({</td>
<td>Open a TCP client socket channel.</td>
</tr>
<tr>
<td><code>host</code> STRING,</td>
<td></td>
</tr>
<tr>
<td><code>port</code> INTEGER,</td>
<td></td>
</tr>
<tr>
<td><code>mode</code> STRING,</td>
<td></td>
</tr>
<tr>
<td><code>timeout</code> INTEGER }</td>
<td></td>
</tr>
<tr>
<td>openServerSocket({</td>
<td>Open a TCP server socket channel.</td>
</tr>
<tr>
<td><code>interface</code> STRING,</td>
<td></td>
</tr>
<tr>
<td><code>port</code> INTEGER,</td>
<td></td>
</tr>
<tr>
<td><code>mode</code> STRING })</td>
<td></td>
</tr>
<tr>
<td>read({</td>
<td>Reads a list of data delimited by a separator from the channel.</td>
</tr>
<tr>
<td><code>[ variable primitive-type [, ..] ]</code></td>
<td></td>
</tr>
<tr>
<td><code>valueList</code> }</td>
<td></td>
</tr>
<tr>
<td>readLine()</td>
<td>Read a complete line from the channel.</td>
</tr>
<tr>
<td>readOctets({</td>
<td>Read a given number of bytes and return as a character string.</td>
</tr>
<tr>
<td><code>length</code> INTEGER }</td>
<td></td>
</tr>
<tr>
<td>setDelimiter({</td>
<td>Define the value delimiter for a channel.</td>
</tr>
<tr>
<td><code>delimiter</code> STRING })</td>
<td></td>
</tr>
<tr>
<td>write({</td>
<td>Writes a list of data delimited by a separator to the channel.</td>
</tr>
<tr>
<td><code>[ valueList ]</code> }</td>
<td></td>
</tr>
<tr>
<td>writeLine({</td>
<td>Write a complete line to the channel.</td>
</tr>
<tr>
<td><code>value</code> STRING }</td>
<td></td>
</tr>
<tr>
<td>writeNoNL({</td>
<td>Writes a string to the channel (without newline character).</td>
</tr>
<tr>
<td><code>value</code> STRING }</td>
<td></td>
</tr>
</tbody>
</table>
**base.Channel.create**
Create a new channel object.

**Syntax**
```
base.Channel.create()
RETURNS base.Channel
```

**Usage**
Use the `base.Channel.create()` class method to create a channel object. The new created object must be assigned to a program variable defined with the `base.Channel` type.

**Example**
```
DEFINE ch base.Channel
LET ch = base.Channel.create()
```

For a complete example, see Example 1: Using record-formatted data file on page 2430.

**base.Channel.close**
Closes the channel.

**Syntax**
```
close()
```

**Usage**
Call the `close()` method when you are finished using the channel. The channel can be re-opened after it has been closed.

**Note:** A channel is automatically closed, when the channel object is destroyed.

**Example**
```
CALL ch.close()
```

For a complete example, see Example 1: Using record-formatted data file on page 2430.

**base.Channel.dataAvailable**
Tests if some data can be read from the channel.

**Syntax**
```
dataAvailable()
RETURNS BOOLEAN
```

**Usage**
The `dataAvailable()` method returns `TRUE` if some data can be read from the channel.

This method is only to be used in some rare cases. Use `dataAvailable()` if the protocol allows asynchronous messages from the peer. An example is an asynchronous error message from the peer, to stop sending more data. `dataAvailable()` checks if at least one byte is available on the stream. A subsequent read will block, if the read operation can not be completed. This should not happen: the methods `read()` and `readLine()` and their
counterparts write() and writeLine() read and write complete lines (a line is a sequence of characters terminated by the line separator).

The dataAvailable() method opens up the possibility to read data asynchronously. One possible use for this method is to stop a data transfer from a local site, after receiving an error message from the remote site.

**Example**

The local site (parent.4gl) sends a huge amount of data to the remote site (child.4gl) using base.Channel.writeLine(). If an error occurs on the remote side during the processing of data, the remote site writes an error message to the channel, causing the local site to stop the data transmission.

On the local site, the file is parent.4gl.

```ga
MAIN
DEFINE n INT
DEFINE c base.Channel

LET c = base.Channel.create()
CALL c.openPipe("fglrun child", "u")
LET n = 0
WHILE TRUE
  IF c.dataAvailable() THEN
    DISPLAY "message from child: ", c.readLine()
    EXIT WHILE
  END IF
  LET n = n + 1
  DISPLAY "parent: write line ", n
  CALL c.writeLine("line " || n)
END WHILE
END MAIN
```

On the remote site, the file is child.4gl:

**Note:** Do not add DISPLAY instructions in the child program, as this would corrupt the pipe communication: The parent program opens a pipe channel read from child's stdout and write to child's stdin.

```ga
MAIN
DEFINE c base.Channel
DEFINE s STRING
DEFINE n INT

LET c = base.Channel.create()
CALL c.openFile("", "u")
LET n = 0
WHILE NOT c.isEof()
  LET s = c.readLine()
  LET n = n + 1
  IF n == 3 THEN
    CALL c.writeLine("error: something happens")
    CALL readRemainingData(c)
    EXIT WHILE
  END IF
END WHILE
END MAIN

FUNCTION readRemainingData(c)
DEFINE c base.Channel
DEFINE s STRING
WHILE NOT c.isEof()
  LET s = c.readLine()
END WHILE
END FUNCTION
```
**Related concepts**

*base.Channel.read* on page 2419
Reads a list of data delimited by a separator from the channel.

*base.Channel.readLine* on page 2420
Read a complete line from the channel.

*base.Channel.isEof*
Detect the end of a file.

**Syntax**

```plaintext
isEof()  
RETURNS BOOLEAN
```

**Usage**

Use the `isEof()` method to detect the end of a file while reading from a channel.

The end of file is only detected after the last read. In other words, you first read, then check for the end of file and process if not end of file.

**Example**

```plaintext
DEFINE s STRING
WHILE TRUE
  LET s = ch.readLine()
  IF ch.isEof() THEN
    EXIT WHILE
  END IF
  DISPLAY s
END WHILE
```

For a complete example, see [Example 3: Reading lines from a text file](#) on page 2430.

*base.Channel.openClientSocket*
Open a TCP client socket channel.

**Syntax**

```plaintext
openClientSocket(
  host STRING,
  port INTEGER,
  mode STRING,
  timeout INTEGER )
```

1. `host` is the name of the host machine you want to connect to.
2. `port` is the port number of the service.
3. `mode` is the open mode. Can be "r", "w" or "u" (combined with "b" if needed).
4. `timeout` is the timeout in seconds. -1 indicates no timeout (wait forever).

**Usage**

Use the `openClientSocket()` method to establish a TCP connection to a server.

Pay attention to the character set used by the network protocol you want to use by opening a channel with this method. The protocol must be based on ASCII, or must use the same character set as the application.

The `host` parameter defines the host name of the server.

The `port` parameter defines the TCP port to connect to.
The opening *mode* can be one of the following:

- **r**: For read mode: only to read from the socket
- **w**: For write mode: only to write to the socket
- **u**: For read and write mode: To read and write from/to the socket

Any of these modes can be followed by **b**, to use binary mode and avoid CR/LF translation on Windows® platforms.

**Note**: The binary mode is only required in specific cases, and will only take effect when writing data.

If the opening *mode* is not one of the above letters, the method will raise error -8085.

When the *timeout* parameter is -1, the connection waits forever.

The method raises error -8084 if the channel cannot be opened.

**Example**

```java
CALL ch.openClientSocket( "localhost", 80, "u", 5 )
```

For a complete example, see Example 4: Communicating with an HTTP server on page 2431.

**Related concepts**

- Setup a TCP socket channel on page 2429
- `base.Channel.openServerSocket` on page 2418
  - Open a TCP server socket channel.

- `base.Channel.openFile`
  - Opens a file channel.

**Syntax**

```java
openFile(
    path STRING,
    mode STRING)
```

1. *path* is the path to the file to open, can be NULL for stdin/stdout.
2. *mode* is the open mode. Can be "r", "w", "a" or "u" (combined with "b" if needed).

**Usage**

The `openFile()` method can be used to open a file for reading, writing, or both.

When passing NULL as file name, the channel can be used to read and/or write to stdout or stdin, depending on the *mode* value.

When passing "<stderr>" as file name the standard error stream will be used. The application can then print messages to stderr (typically for batch programs). See Example 5: Writing to STDERR on page 2431

The opening *mode* can be one of the following:

- **r**: For read mode: reads from a file (standard input if path is NULL).
- **w**: For write mode: starts with an empty file (standard output if the path is NULL).
- **a**: For append mode: writes at the end of a file (standard output if the path is NULL).
- **u**: For read from standard input and write to standard output (path must be NULL).

Any of these modes can be followed by **b**, to use binary mode and avoid CR/LF translation on Windows® platforms.

**Note**: The binary mode is only required in specific cases, and will only take effect when writing data.

If the opening *mode* is not one of the above letters, the method will raise error -8085

When you use the **w** or **a** modes, the file is created if it does not exist.
The method raises error -6340 if the file cannot be opened.

Example

```plaintext
CALL ch.openFile( "file.txt", "w" )
```

For a complete example, see Example 1: Using record-formatted data file on page 2430.

`base.Channel.openPipe`

Opens a pipe channel to a subprocess.

Syntax

```plaintext
openPipe (
    command STRING,
    mode STRING
)
```

1. `command` is the system command to be executed.
2. `mode` is the open mode. Can be "r", "w", "a" or "u" (combined with "b" if needed).

Usage

With the `openPipe()` method, you can read from the standard output of a subprocess, write to the standard input, or both.

**Important:** This feature is not supported on mobile platforms.

The opening `mode` can be one of the following:

- r: For read only from standard output of the command.
- w: For write only to standard input of the command.
- a: For write only to standard input of the command.
- u: For read from standard output and write to standard input of the command.

Any of these modes can be followed by b, to use binary mode and avoid CR/LF translation on Windows® platforms.

**Note:** The binary mode is only required in specific cases, and will only take effect when writing data.

If the opening `mode` is not one of the above letters, the method will raise error -8085.

Example

```plaintext
CALL ch.openPipe( "ls", "r" )
```

For a complete example, see Example 2: Executing the ls UNIX command on page 2430.

`base.Channel.openServerSocket`

Open a TCP server socket channel.

Syntax

```plaintext
openServerSocket (
    interface STRING,
    port INTEGER,
    mode STRING
)
```

1. `interface` is the name of the network interface to be used.
2. `port` is the port number of the service.
3. `mode` is the open mode. Only "u" is allowed (combined with "b" if needed).
Usage

The `openServerSocket()` method initializes the channel object to listen to a given TCP interface and port. The server socket accepts multiple client connections. After calling the `openServerSocket()` method, a call to `readLine()` waits until the first client connects and returns after reading a complete line. Only one client connection can be serviced at time: it's not possible to select a specific client connection. A client connection must be closed by writing the EOF character to the channel. The EOF character is ASCII 26. Do not call `base.Channel.close()` to close a client/server connection; this would close the server socket and reject any pending client connection. The next call to `readLine()` after writing EOF will wait until the next client connects or select the next pending client.

Pay attention to the character set used by the network protocol you want to use by opening a channel with this method. The protocol must be based on ASCII, or must use the same character set as the application.

The `interface` parameter defines the network interface to be used, in case the server uses different network adapters. Use NULL to listen to all network interfaces, or when the server has only one network interface.

The `port` parameter defines the TCP port to listen to.

The opening `mode` must be "u", to read and write from/to the socket. The method will raise error -8085 if the mode is different from "u".

The "u" mode can be combined with the "b" binary mode, to avoid CR/LF translation on Windows® platforms.

**Note:** The binary mode is only required in specific cases, and will only take effect when writing data.

The method raises error -8084 if the socket cannot be opened.

Example

```main
DEFINE io base.Channel
DEFINE s STRING
LET io = base.Channel.create()
CALL io.openServerSocket("127.0.0.1", 4711, "u")
WHILE TRUE
    LET s = io.readLine()
    CALL io.writeLine(s)
    -- next line closes the current connection
    CALL io.writeLine(ASCII 26) -- EOF
END WHILE
END MAIN
```

Related concepts

- Setup a TCP socket channel on page 2429
- `base.Channel.openClientSocket` on page 2416
- Open a TCP client socket channel.

`base.Channel.read`

Reads a list of data delimited by a separator from the channel.

Syntax

```plaintext
read(
    [ variable primitive-type [, . . . ] ]
) RETURNS INTEGER
```

1. `variable` is a program variable.
2. `primitive-type` is a primitive data type such as `INTEGER`, `VARCHAR(50)`, etc.
3. A comma-separated list of simple variables can be provided. These are provided as a variable parameter list.
4. The variable(s) must be specified between `[ ]` square brackets.
5. The `record.*` notation can be used, to read all values into the members of the record.

**Usage**

After opening the channel object, use the `read()` method to read a record of data from the channel.

The `read()` method uses the field delimiter defined by `setDelimiter()`.

The `read()` method takes a modifiable list of variables as parameter, by using the `[ ]` square brace notation.

A call to `read()` is blocking until the read operation is complete.

If the `read()` method returns less data than expected, then the remaining variables will be initialized to NULL. If the `read()` method returns more data than expected, the data is silently ignored.

Any target variable must have a **primitive type**, or be a `RECORD` that contains only members defined with a primitive type.

If data is read, the `read()` method returns **TRUE**. Otherwise, it returns **FALSE**, indicating the end of the file or stream.

**Note:** Files encoded in UTF-8 can start with the UTF-8 Byte Order Mark (BOM), a sequence of 0xEF 0xBB 0xBF bytes, also known as UNICODE U+FEFF. When reading files, Genero BDL will ignore the UTF-8 BOM, if it is present at the beginning of the file. This applies to instructions such as LOAD, as well as I/O APIs such as `base.Channel.read()` and `readLine()`.

**Example**

```plaintext
WHILE ch.read([cust_rec.*])
    ...
END WHILE
```

For a complete example, see Example 1: Using record-formatted data file on page 2430.

**Related concepts**

- Read and write record data on page 2425
- `base.Channel.dataAvailable` on page 2414
  Tests if some data can be read from the channel.

- `base.Channel.readLine`
  Read a complete line from the channel.

**Syntax**

```plaintext
readLine()
RETURNS STRING
```

**Usage**

After opening the channel object, use the `readLine()` method to read a complete line from the channel.

The `readLine()` method returns an empty string if the line is empty.

A call to `readLine()` is blocking until the read operation is complete.

The `readLine()` function returns **NULL** if end of file is reached. To distinguish empty lines from **NULL**, you must use the `STRING` data type. If you use a `CHAR` or `VARCHAR`, you will get **NULL** for empty lines. To detect the end of file, use the `isEof()` method.

**Note:** Files encoded in UTF-8 can start with the UTF-8 Byte Order Mark (BOM), a sequence of 0xEF 0xBB 0xBF bytes, also known as UNICODE U+FEFF. When reading files, Genero BDL will ignore the UTF-8 BOM, if it is present at the beginning of the file. This applies to instructions such as LOAD, as well as I/O APIs such as `base.Channel.read()` and `readLine()`.
Example

```
WHILE TRUE
   LET s = ch.readLine()
   IF ch.isEof() THEN EXIT WHILE END IF
   ... 
END WHILE
```

For a complete example, see Example 3: Reading lines from a text file on page 2430.

Related concepts

Read and write text lines on page 2426
base.Channel.dataAvailable on page 2414
Tests if some data can be read from the channel.

```
base.Channel.readOctets
```

Read a given number of bytes and return as a character string.

Syntax

```
readOctets(
   length INTEGER)
RETURNS STRING
```

1. `length` is the number of bytes to read, not the number of characters.

Usage

After opening the channel object, call the `readOctets()` method to read a given number of bytes from the channel. The bytes are returned as a character string.

Important: The bytes read with `readOctets()` must match the application locale. In multibyte encoding (UTF-8), if a sequence of bytes does represent a valid character, the resulting string will get ? question mark for invalid characters and data will be lost.

A valid use case of the method is the HTTP protocol. Reading HTML content with `readLine()` is not possible. The body consists of multiple lines, and the last line might not be terminated by a line-terminator, and the stream gets no EOF:

```
HTTP/1.0 200 OK
Date: Wed, 16 Apr 2014 18:50:51 GMT
Content-Type: text/html
Content-Length: 1354

<html>
<body>
<h1>My title</h1>
:
</body>
</html>
```

The `readOctets()` function returns NULL if end of file is reached.

Note: To distinguish empty lines from NULL, you must use the STRING data type. If you use a CHAR or VARCHAR, you will get NULL for empty lines. To properly detect end of file, use the `isEof()` method.

If `readOctets()` cannot read as many bytes as specified by the parameter, the method returns NULL. Before reading the actual bytes with a `readOctets()` call, find the number of data bytes to read from stream source, as shown in the example below.

Example

Note: This example uses UTF-8 encoding.
The main program:

```
MAIN
DEFINE len INTEGER
DEFINE chunk STRING
DEFINE ch base.Channel
LET ch = base.Channel.create()
CALL ch.openFile("file.txt","r")
WHILE TRUE
  LET len = ch.readOctets(3) -- 3 digits for length
  IF ch.isEof() THEN EXIT WHILE END IF
  LET chunk = ch.readOctets(len)
  DISPLAY len USING "##&"," ", NVL(chunk,"(NULL)")
END WHILE
CALL ch.close()
END MAIN
```

The data file (contains 3 digits for byte length of following string):

```
003abc006forêt000001x
```

Program output (note that forêt needs 3+2+1=6 bytes in UTF-8):

```
$ fgllcomp -M readOctets.4gl && fgllrun readOctets.42m
 3 abc
 6 forêt
 0
 1 x
```

Related concepts

*Setup a TCP socket channel* on page 2429

*base.Channel.setDelimiter*

Define the value delimiter for a channel.

**Syntax**

```
setDelimiter ( delimiter STRING )
```

1. `delimiter` is the value delimiter to be used.

**Usage**

After creating the channel object, define the field value delimiter with the `setDelimiter()` method.

```
CALL ch.setDelimiter("^")
```

The default delimiter is defined by the `DBDELIMITER` environment variable, or a pipe (`|`) if `DBDELIMITER` is not defined.

Specify `CSV` as the delimiter to read/write in Comma Separated Value format.

```
CALL ch.setDelimiter("CSV")
```

**Important:** Setting a NULL delimiter is allowed for backward compatibility, but must be avoided. This was a workaround to read/write complete lines. If the delimiter is set to NULL, the `read()` and `write()` methods do not use the backslash (\) escape character. As a result, data with special characters like backslash, delimiter or line-feed will be written as is, and reading data will ignore escaped characters in the source stream. If you need to read or
write non-formatted data, it is recommended that you use the `readLine() / writeLine()` methods instead. These methods do not use a delimiter, nor do they use the backslash escape character.

**Related concepts**
Read and write record data on page 2425

**base.Channel.write**
Writes a list of data delimited by a separator to the channel.

**Syntax**

```
write(  
    [ valueList ]  
)
```

1. `valueList` is a list of expressions separated by a comma.

**Usage**
After opening a channel, use the `write()` method to write a record of data to the channel.

The `write()` method uses the field delimiter defined by `setDelimiter()`.

The `write()` method takes a modifiable list of variables as the parameter, using the `[ ]` square brace notation.

The method raises error `-6345` if the channel fails to write data.

**Example**

```
CALL ch.write([cust_rec.*])
```

**Related concepts**
Read and write record data on page 2425

**base.Channel.writeLine**
Write a complete line to the channel.

**Syntax**

```
writeLine(  
    value STRING  
)
```

1. `value` is the string expression to be written to the channel.

**Usage**
After opening a channel, use the `writeLine()` method to write a line of text to the channel.

The `writeLine()` method does not use the field delimiter, it writes the text data to the stream, with an ending newline character.

To write a string with no ending newline character, use the `writeNoNL()` method.

The method raises error `-6345` if the channel fails to write data.

**Example**

```
CALL ch.writeLine("Customer number: " || custno)
```

For a complete example, see Example 5: Writing to STDERR on page 2431.

**Related concepts**
Read and write text lines on page 2426
base.Channel.writeNoNL
Writes a string to the channel (without newline character).

Syntax

writeNoNL(
  value STRING
)

1. value is the character string to be written to the channel.

Usage

After opening a channel, use the writeNoNL() method to write a string to the channel, without a trailing newline character.

**Important:** Do not confuse the writeNoNL() method with the write() method. The first is provided to write raw character strings to the stream, while the second is designed to write records with formatted data and field delimiters. Note also that the Channel class provides the writeLine() method to write a string with an ending newline character.

The method raises error -6345 if the channel fails to write data.

Example

CALL ch.writeNoNL("Some text ...")

Related concepts

base.Channel.readOctets on page 2421
Read a given number of bytes and return as a character string.

base.Channel.writeLine on page 2423
Write a complete line to the channel.

base.Channel.write on page 2423
Writes a list of data delimited by a separator to the channel.

Usage

The base.Channel class is a built-in class providing basic input/output functionality for:

- text file reading/writing
- subprocess communication (through pipes)
- basic network communication (through TCP sockets)

**Important:** No character set conversion is done when reading or writing data with channel objects. The character set used by the other end of the channel must correspond to the locale of the runtime system, for both input and output. For more details, see Character string encoding on page 2428.

Steps to use a channel object:

- Define a variable with the base.Channel type.
- Create a channel object with base.Channel.create() and assign it to the variable.
- Open the channel for a file, piped process or socket (as a client).
- Read and/or write data on the channel.
- Close the channel.

**Note:** When reading or writing character strings, the escape character is the backslash (\).

The are three modes to read and write data with Channels:

1. Reading/writing formatted data as a set of fields in a line (records), with the read() and write() methods.
2. Reading/writing complete lines with the readLine() and writeLine() methods.
3. Handling raw character string data by reading/writing pieces of strings, with the `readOctets()` and `writeNoNL()` methods.

Channels may raise exceptions that can be trapped with `WHENEVER ERROR` or `TRY/CATCH` blocks.

*Read and write record data*

When the channel is open, use the `read()/write()` methods to read and write data records where field values are separated by a delimiter defined by `setDelimiter()`.

**Note:** The `LOAD/UNLOAD` SQL instructions follow the same formatting rules as the `read()/write()` channel methods.

The input or output stream is text data where each line contains the string representation of a record. Field values are separated by the delimiter character defined.

For example, a formatted text file looks like this, when using a default pipe (|) delimiter:

| 8712 | David | Cosneski | 24-12-1978 |
| 3422 | Frank | Zapinetti | 13-04-1968 |
| 323  | Mark  | Kelson    | 03-10-1988 |

In the serialized data, empty fields (|) have a length of zero and are considered as **NULL**.

The code in the example reads the above field-formatted data:

```
MAIN
  DEFINE ch base.Channel
  DEFINE custinfo RECORD
      cust_num INTEGER,
      cust_fname VARCHAR(40),
      cust_lname VARCHAR(40),
      cust_bdate DATE
  END RECORD
  LET ch = base.Channel.create()
  CALL ch.setDelimiter("|")
  CALL ch.openFile("custinfo.txt","r")
  WHILE ch.read([custinfo.*])
    DISPLAY custinfo.*
  END WHILE
  CALL ch.close()
END MAIN
```

The backslash \ is the escape character: When writing data with `write()`, special characters like the backslash, line-feed or the delimiter character are escaped. When reading data with `read()`, any escaped `\char` character is converted to `char`.

The following code example writes a single field value where the character string contains a backslash, the pipe delimiter and a line-feed character. The backslash is also the escape character for string literals, therefore we need to double the backslash to get a backslash in the string, while the line-feed character (<lf>) is represented by backslash-n (\n) in string literals:

```
CALL ch.setDelimiter("|")
CALL ch.write("aaa\bbb|ccc\n
ddd")   -- [aaa<bs>bbb|ccc<lf>ddd]
```

This code will produce the following text file:

```
aaa\bbb\ccc\n
ddd
```

When reading such a line back into memory with the `read()` method, all escaped characters are converted back to the single character. In this example, `\` becomes `\`, `\|` becomes `|` and `<lf>` becomes `<lf>`.
When using the `read()`/`write()` methods, the escaped line-feed (LF, \n) characters are written as BS + LF to the output, and when reading with `read()`, BS + LF are detected and interpreted, to be restored as if the value was assigned by a `LET` instruction, with the same string used in the `write()` function.

If you want to write a LF as part of a value, the string must contain the backslash and line-feed as two independent characters. You need to escape the backslash when you write the string constant in the .4gl source file.

```plaintext
CALL ch.setDelimiter("|")
CALL ch.write("aaa\nbbb")  -- [aaa<bs><lf>bbb]
CALL ch.write("ccc\nddd")  -- [aaa<lf>bbb]
```

The code above would generate the following output:

```
aaa
bbb|
ccc
ddd|
```

where the first two lines contain data for the same line, in the meaning of a Channel record.

When you read these lines back with a `read()` call, you get the following strings in memory:

```
Read 1: aaa<bs><lf>bbb
Read 2: ccc
Read 3: ddd
```

These reads would correspond to the following assignments when using string constants:

```
LET s = "aaa\\nbbb"
LET s = "ccc"
LET s = "ddd"
```

Data of a record can also be serialized as CSV (Comma Separated Values), when defining "CSV" as delimiter value:

```
CALL ch.setDelimiter("CSV")
```

This CSV format is similar to the standard channel format, with the following differences:

- Values in the file might be surrounded with double quotes (").
- If a value contains a comma or a NEWLINE, it is not escaped; the value must be quoted in the file.
- Double-quote characters in values are doubled in the output file and the output value must be quoted.
- Backslash characters are not escaped and are read as is; the value must be quoted.
- Leading and trailing blanks are kept (no truncation).
- No ending delimiter is expected at the end of the record line.

**Read and write text lines**

When the channel is open, use the `readLine()`/`writeLine()` methods to read and write simple text lines of data terminated by a line terminator.

When using the `readLine()` and `writeLine()` functions, a LF character represents the end of a line.

For example, a simple text file can look like this:

```
first line
second line
third line
```

Sample code to read the above text file is as follows:

```
MAIN
```
DEFINE i INTEGER
DEFINE s STRING
DEFINE ch base.Channel
LET ch = base.Channel.create()
CALL ch.openFile("file.txt","r")
LET i = 1
WHILE TRUE
  LET s = ch.readLine()
  IF ch.isEof() THEN EXIT WHILE END IF
  DISPLAY i, " ", s
  LET i = i + 1
END WHILE
CALL ch.close()
END MAIN

LF characters escaped by a backslash are **not** interpreted as part of the line during a `readLine()` call.

When a line is written, any LF characters in the string will be written as is to the output. When a line is read, the LF escaped by a backslash is **not** interpreted as part of the line.

For example, this code:

```plaintext
CALL ch.writeLine("aaa\\nbbb")  -- [aaa<bs><lf>bbb]
CALL ch.writeLine("ccc\nddd")    -- [aaa<lf>bbb]
```

would generate this output:

```plaintext
aaa\nbbb
ccc
ddd
```

and the subsequent `readLine()` will read four different lines, where the first line is ended by a backslash:

```plaintext
Read 1 aaa<bs>
Read 2 bbb
Read 3 ccc
Read 4 ddd
```

**Line terminators on Windows and UNIX**

On Windows® platforms, DOS formatted text files use CR/LF as line terminators. You can manage these type of files with the `base.Channel` class.

By default, on both Windows® and UNIX™ platforms, when records are read from a DOS file with the `base.Channel` class, the CR/LF line terminator is removed. When a record is written to a file on Windows®, the lines are terminated with CR/LF in the file; on UNIX™, the lines are terminated with LF only.

To avoid the automatic translation of CR/LF on Windows®, you can use the `b` option of the `openFile()` and `openPipe()` methods. You can combine the `b` option with `r` or `w`, based on the read or write operations that you want to perform.

```plaintext
CALL ch.openFile( "mytext.txt", "rb" )
```

On Windows®, when lines are read with the `b` option, only LF is removed from CR/LF line terminators; CR will be copied as a character part of the last field. In contrast, when lines are written with the `b` option, LF characters will not be converted to CR/LF.

On UNIX™, writing lines with or without the binary mode option does not matter.
| Library reference | 2428

Character string encoding
The runtime system stores and handles character strings in a given application locale, defined by environment
settings.
When reading or writing character strings from/to a channel, no character set conversion is done. The other end of the
channel must use the same encoding as the runtime system.
$ cat ch.4gl
-- This source is using UTF-8 encoding
MAIN
DEFINE ch base.Channel
LET ch = base.Channel.create()
CALL ch.openFile("myfile.txt","w")
CALL ch.writeLine("abcéúíôû")
CALL ch.close()
END MAIN
$ fglcomp ch.4gl && fglrun ch.42m
$ file myfile.txt
myfile.txt: UTF-8 Unicode text
BYTE data serialization
When using channels, binary data stored in BYTE variables is serialized in hexadecimal.
For example, with the following program:
MAIN
DEFINE ch base.Channel
DEFINE rec RECORD
id INT,
image BYTE
END RECORD
LET ch = base.Channel.create()
CALL ch.openFile("myfile.txt","w")
LET rec.id = 99999
LOCATE rec.image IN FILE "mydata.bin"
CALL ch.write(rec)
CALL ch.close()
END MAIN
On Unix, create the binary file like this:
$ echo -n -e '\x8f\x9f\xff' > mydata.bin
Then compile and run the program:
$ fglcomp ch.4gl && fglrun ch.42m
The resulting text file will contain the following serialized data:
99999|8f9fff|
Handle channel exceptions
Channel errors can be trapped with the WHENEVER ERROR exception handler:
WHENEVER ERROR CONTINUE
CALL ch.write([num,label])
IF STATUS THEN
ERROR "An error occurred while reading from Channel"


CALL ch.close()
RETURN -1
END IF
WHENEVER ERROR STOP

Or with a TRY/CATCH block:

TRY
  CALL ch.write([num,label])
CATCH
  ERROR "An error occurred while reading from Channel"
  CALL ch.close()
  RETURN -1
END TRY

Setup a TCP socket channel

The base.Channel class provides methods to implement basic TCP client and server programs.

Important: Consider character set encodings when designing such programs. No implicit character set conversion is done by the runtime system. Both client and server must use the same character set and length semantics.

The following code example implements a client program connecting to a TCP port, using the openClientSocket() method:

MAIN
  DEFINE ch base.Channel,
  time DATETIME HOUR TO SECOND,
  data STRING
  LET ch = base.Channel.create()
  CALL ch.openClientSocket("localhost",99999,"u",3)
  CALL ch.readLine()
  LET time = ch.readLine()
  DISPLAY "client 1: ", time
  CALL ch.readLine()
  LET data = ch.readLine()
  DISPLAY "client 2: ", data
  CALL ch.writeLine("disconnect")
  CALL ch.close()
END MAIN

The following code example implements the server program that can be used with the above client program. The server program uses the openServerSocket() and readLine() methods to listen to a given TCP interface/port. Note that the connection with a client must be ended by sending an EOF character (ASCII 26) to the client, the next readLine() call will wait for a new client connection, or select a pending client connection:

MAIN
  DEFINE ch base.Channel,
  cmd, data STRING
  LET ch = base.Channel.create()
  DISPLAY "starting server...
  CALL ch.openServerSocket(null, 99999, "u")
  WHILE TRUE
    LET cmd = ch.readLine()
    IF ch.isEof() THEN
      DISPLAY "Connection ended by client..."
      EXIT WHILE
    END IF
    DISPLAY "cmd: ", cmd
    IF cmd == "get_time" THEN
      CALL ch.writeLine(CURRENT HOUR TO SECOND)
    END IF
    IF cmd == "get_string" THEN

LET data = "This is a string..."
CALL ch.writeLine(data)
END IF
IF cmd == "disconnect" THEN
CALL ch.writeLine(ASCII 26) -- EOF
END IF
END WHILE
DISPLAY "end of server..."
END MAIN

Related concepts
Application locale on page 512
The application locale defines the language and codeset for your application.

Examples
base.Channel usage examples.

Example 1: Using record-formatted data file
This program reads data from file.txt, which contains two columns separated by a pipe (|) character. It writes this data to the end of fileout.txt, using a percent sign (%) as the delimiter.

MAIN
DEFINE custinfo RECORD
    cust_num INTEGER,
    cust_name VARCHAR(40)
END RECORD
DEFINE ch_in, ch_out base.Channel
LET ch_in = base.Channel.create()
CALL ch_in.setDelimiter("|")
LET ch_out = base.Channel.create()
CALL ch_out.setDelimiter("%")
CALL ch_in.openFile("file.txt","r")
CALL ch_out.openFile("fileout.txt","w")
WHILE ch_in.read([custinfo.*])
    CALL ch_out.write([custinfo.*])
END WHILE
CALL ch_in.close()
CALL ch_out.close()
END MAIN

Example 2: Executing the ls UNIX™ command
This program executes the ls command and displays the file names and extensions separately.

MAIN
DEFINE fn CHAR(40)
DEFINE ex CHAR(10)
DEFINE ch base.Channel
LET ch = base.Channel.create()
CALL ch.setDelimiter(".")
CALL ch.openPipe("ls -l","r")
WHILE ch.read([fn,ex])
    DISPLAY fn, "   ", ex
END WHILE
CALL ch.close()
END MAIN

Example 3: Reading lines from a text file

MAIN
DEFINE i INTEGER
DEFINE s STRING
DEFINE ch base.Channel
LET ch = base.Channel.create()
CALL ch.openFile("file.txt","r")
LET i = 1
WHILE TRUE
  LET s = ch.readLine()
  IF ch.isEof() THEN EXIT WHILE END IF
  DISPLAY i, " ", s
  LET i = i + 1
END WHILE
CALL ch.close()
END MAIN

Example 4: Communicating with an HTTP server

MAIN
DEFINE ch base.Channel, eof INTEGER
LET ch = base.Channel.create()
-- HTTP protocol forces every line to be terminate by \r\n
-- So we use channel binary mode to avoid CR+LF translation on Windows.
-- In text mode, each line would be terminated by \r\r\n on Windows.
WHENEVER ERROR CONTINUE
  CALL ch.openClientSocket("localhost", 80, "ub", 30)
  IF STATUS != 0 THEN
    DISPLAY "Could not open socket: error ", STATUS
    EXIT PROGRAM 1
  END IF
WHENEVER ERROR STOP
-- HTTP expects CR+LF: Note that LF is added by writeLine()!
CALL ch.writeLine("GET / HTTP/1.0\r")
-- No HTTP headers...
-- Empty line = end of headers
CALL ch.writeLine("\r")
WHILE NOT eof
  DISPLAY ch.readLine()
  LET eof = ch.isEof()
END WHILE
CALL ch.close()
END MAIN

Example 5: Writing to STDERR

The following code implements a simple user function to write messages to the stderr stream:

FUNCTION to_stderr(s)
  DEFINE s STRING
  DEFINE c base.Channel
  LET c = base.Channel.create()
  CALL c.openFile("<stderr>", "w")
  CALL c.writeLine(s)
END FUNCTION
The SqlHandle class
The `base.SqlHandle` class is a built-in class providing an API to execute parameterized SQL statements, with or without result sets.

base.SqlHandle methods

Table 514: Class methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>base.SqlHandle.create()</code></td>
<td>Create a new <code>base.SqlHandle</code> object.</td>
</tr>
<tr>
<td>RETURNS <code>base.SqlHandle</code></td>
<td></td>
</tr>
</tbody>
</table>

### Table 515: Object methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>close()</td>
<td>Closes the SQL handle (cursor).</td>
</tr>
<tr>
<td>execute()</td>
<td>Executes a simple SQL statement (without result set).</td>
</tr>
<tr>
<td>fetch()</td>
<td>Fetches a new row from the SQL result set.</td>
</tr>
<tr>
<td>fetchAbsolute(position INTEGER)</td>
<td>Fetches a specified row in a scrollable SQL result set.</td>
</tr>
<tr>
<td>fetchFirst()</td>
<td>Fetches the first row in a scrollable SQL result set.</td>
</tr>
<tr>
<td>fetchLast()</td>
<td>Fetches the last row in a scrollable SQL result set.</td>
</tr>
<tr>
<td>fetchPrevious()</td>
<td>Fetches the previous row in a scrollable SQL result set.</td>
</tr>
<tr>
<td>fetchRelative(position INTEGER)</td>
<td>Fetches a row relative to the current row in a scrollable SQL result set.</td>
</tr>
<tr>
<td>flush()</td>
<td>Flushes the rows from the insert cursor buffer.</td>
</tr>
<tr>
<td>resultSetCount()</td>
<td>Returns the number of result set columns produced by the SQL statement.</td>
</tr>
<tr>
<td>resultSetName(index INTEGER)</td>
<td>Returns the name of a column in the result set produced by the SQL statement.</td>
</tr>
<tr>
<td>resultSetType(index INTEGER)</td>
<td>Returns the Genero type name of a column in the result set produced by the SQL statement.</td>
</tr>
<tr>
<td>resultSetValue(index INTEGER)</td>
<td>Returns the value of a column in the result set produced by the SQL statement.</td>
</tr>
<tr>
<td>open()</td>
<td>Opens the SQL handle (SELECT or INSERT cursor).</td>
</tr>
<tr>
<td>openScrollCursor()</td>
<td>Opens the SQL handle (with scrollable option).</td>
</tr>
<tr>
<td>prepare(sql STRING)</td>
<td>Prepares a SQL statement for the SQL handle.</td>
</tr>
<tr>
<td>put()</td>
<td>Put a new row in the insert cursor buffer.</td>
</tr>
<tr>
<td>setParameter(index INTEGER, value fgl-type)</td>
<td>Sets the value of a SQL parameter for this SQL handle.</td>
</tr>
</tbody>
</table>
**base.SqlHandle.create**
Create a new base.SqlHandle object.

**Syntax**

```c
base.SqlHandle.create()
RETURNS base.SqlHandle
```

**Usage**

Use the `create()` method to create a `base.SqlHandle` object to execute SQL statements.

The value returned by this method must be assigned to a variable defined with the `base.SqlHandle` type.

As with other built-in classes, the SqlHandle object will be automatically destroyed if no longer referenced.

**Example**

```c
DEFINE sh base.SqlHandle
LET sh = base.SqlHandle.create()
...
```

For a complete example, see Example 2: SqlHandle with result set SQL on page 2446.

**base.SqlHandle.close**

Closes the SQL handle (cursor).

**Syntax**

```c
close()
```

**Usage**

Call the `close()` method when you are finished using the SQL handle.

The statement can be re-opened after it has been closed.

**Note:** A SqlHandle object is automatically closed when the object is destroyed.

As with standard Genero SQL instructions, SQL errors can be trapped with `WHENEVER ERROR` or `TRY / CATCH` blocks and by testing `SQLCA.SQLCODE`.

**Example**

```c
DEFINE sh base.SqlHandle
...
CALL sh.close()
```

For a complete example, see Example 2: SqlHandle with result set SQL on page 2446.

**base.SqlHandle.execute**

Executes a simple SQL statement (without result set).

**Syntax**

```c
execute()
```
**Usage**

Call the `execute()` method to execute the prepared SQL statement, without producing a result set (INSERT, UPDATE, DELETE, CREATE TABLE, ...).

The SQL statement must have been prepared with a `prepare()` call.

If the SQL statement contains ? parameter place holders, issue a `setParameter()` call for each parameter before executing the SQL statement.

As with standard Genero SQL instructions, SQL errors can be trapped with `WHENEVER ERROR` or `TRY / CATCH` blocks and by testing `SQLCA.SQLCODE`.

**Example**

```groovy
DEFINE sh base.SqlHandle
...
CALL sh.execute()
```

For a complete example, see Example 1: SqlHandle with simple SQL on page 2445.

**Syntax**

```groovy
fetch()
```

**Usage**

Call the `fetch()` method to fetch a new row from the SQL result set.

**Note:** When using a dynamic scroll cursor, the `fetch()` method can be used to fetch to the next row.

The SQL statement must have been opened with an `open()` call.

After performing the fetch call, you can query for column information with the `getResultCount()`, `getResultName(index)`, `getResultType(index)` and `getResultValue(index)` methods.

If no row is found (end of result set), `SQLCA.SQLCODE` is set to 100 (NOTFOUND).

As with standard Genero SQL instructions, SQL errors can be trapped with `WHENEVER ERROR` or `TRY / CATCH` blocks and by testing `SQLCA.SQLCODE`.

**Example**

```groovy
DEFINE sh base.SqlHandle
...
CALL sh.fetch()
```

For a complete example, see Example 2: SqlHandle with result set SQL on page 2446.

**Syntax**

```groovy
fetchAbsolute(position INTEGER)
```

1. `position` is the absolute row position in the result set (starts at 1).
**Usage**

Call the `fetchAbsolute()` method to fetch the specified row in a scrollable SQL result set.

The SQL statement must have been opened with an `openScrollCursor()` call.

After performing the fetch call, you can query for column information with the `getResultCount()`, `getResultName(index)`, `getResultType(index)` and `getResultValue(index)` methods.

If no row is found (end of result set), SQLCA.SQLCODE is set to 100 (NOTFOUND).

If the specified position does not correspond to a row position in the result set, SQLCA.SQLCODE is set to 100 (NOTFOUND).

As with standard Genero SQL instructions, SQL errors can be trapped with `WHENEVER ERROR` or `TRY / CATCH` blocks and by testing SQLCA.SQLCODE.

**Example**

```plaintext
DEFINE sh base.SqlHandle
...
CALL sh.fetchAbsolute(10)
```

For a complete example, see Example 4: SqlHandle with scroll cursor on page 2447.

**base.SqlHandle.fetchFirst**

Fetches the first row in a scrollable SQL result set.

**Syntax**

```plaintext
fetchFirst()
```

**Usage**

Call the `fetchFirst()` method to fetch the first row in a scrollable SQL result set.

The SQL statement must have been opened with an `openScrollCursor()` call.

After performing the fetch call, you can query for column information with the `getResultCount()`, `getResultName(index)`, `getResultType(index)` and `getResultValue(index)` methods.

If no row is found (end of result set), SQLCA.SQLCODE is set to 100 (NOTFOUND).

If the result set is empty, SQLCA.SQLCODE is set to 100 (NOTFOUND).

As with standard Genero SQL instructions, SQL errors can be trapped with `WHENEVER ERROR` or `TRY / CATCH` blocks and by testing SQLCA.SQLCODE.

**Example**

```plaintext
DEFINE sh base.SqlHandle
...
CALL sh.fetchFirst()
```

For a complete example, see Example 4: SqlHandle with scroll cursor on page 2447.

**base.SqlHandle.fetchLast**

Fetches the last row in a scrollable SQL result set.

**Syntax**

```plaintext
fetchLast()
```
Usage

Call the `fetchLast()` method to fetch the last row in a scrollable SQL result set.
The SQL statement must have been opened with an `openScrollCursor()` call.

After performing the fetch call, you can query for column information with the `getResultCount()`, `getResultName(index)`, `getResultType(index)` and `getResultValue(index)` methods.

If no row is found (end of result set), SQLCA.SQLCODE is set to 100 (NOTFOUND).

If the result set is empty, SQLCA.SQLCODE is set to 100 (NOTFOUND).

As with standard Genero SQL instructions, SQL errors can be trapped with `WHENEVER ERROR` or `TRY / CATCH` blocks and by testing `SQLCA.SQLCODE`.

Example

```
DEFINE sh base.SqlHandle
...
CALL sh.fetchLast()
```

For a complete example, see Example 4: SqlHandle with scroll cursor on page 2447.

**base.SqlHandle.fetchPrevious**
Fetched the previous row in a scrollable SQL result set.

Syntax

```
fetchPrevious()
```

Usage

Call the `fetchPrevious()` method to fetch the previous row in a scrollable SQL result set.
The SQL statement must have been opened with an `openScrollCursor()` call.

After performing the fetch call, you can query for column information with the `getResultCount()`, `getResultName(index)`, `getResultType(index)` and `getResultValue(index)` methods.

If no row is found (end of result set), SQLCA.SQLCODE is set to 100 (NOTFOUND).

If the result set is empty, or if the current row is already the first row, SQLCA.SQLCODE is set to 100 (NOTFOUND).

As with standard Genero SQL instructions, SQL errors can be trapped with `WHENEVER ERROR` or `TRY / CATCH` blocks and by testing `SQLCA.SQLCODE`.

Example

```
DEFINE sh base.SqlHandle
...
CALL sh.fetchPrevious()
```

For a complete example, see Example 4: SqlHandle with scroll cursor on page 2447.

**base.SqlHandle.fetchRelative**
Fetched a row relative to the current row in a scrollable SQL result set.

Syntax

```
fetchRelative(
    position INTEGER)
```
1. *position* is the row offset in the result set. The offset can be negative, to fetch backwards.

**Usage**

Call the `fetchRelative()` method to fetch the row at the specified offset, relative to the current row in a scrollable SQL result set.

The SQL statement must have been opened with an `openScrollCursor()` call.

After performing the fetch call, you can query for column information with the `getResultCount()`, `getResultName(index)`, `getResultType(index)` and `getResultValue(index)` methods.

If no row is found (end of result set), `SQLCA.SQLCODE` is set to 100 (NOTFOUND).

If the result set is empty, or if no row exists at the specified offset relative to the current row position, `SQLCA.SQLCODE` is set to 100 (NOTFOUND).

As with standard Genero SQL instructions, SQL errors can be trapped with `WHenever ERROR` or `TRY / CATCH` blocks and by testing `SQLCA.SQLCODE`.

**Example**

```plaintext
DEFINE sh base.SqlHandle
...  
CALL sh.fetchRelative(-3)
```

For a complete example, see [Example 4: SqlHandle with scroll cursor](#) on page 2447.

**base.SqlHandle.flush**

Flushes the rows from the insert cursor buffer.

**Syntax**

```plaintext
flush()
```

**Usage**

With an insert cursor, call the `flush()` method to force the buffered rows to the database server.

The SQL statement must have been opened with an `open()` call.

As with standard Genero SQL instructions, SQL errors can be trapped with `WHenever ERROR` or `TRY / CATCH` blocks and by testing `SQLCA.SQLCODE`.

**Example**

```plaintext
DEFINE sh base.SqlHandle
...  
CALL sh.flush()
```

For a complete example, see [Example 3: SqlHandle with insert cursor](#) on page 2446.

**base.SqlHandle.getResultCount**

Returns the number of result set columns produced by the SQL statement.

**Syntax**

```plaintext
getResultCount()  
RETURNS INTEGER
```
Usage
Call the `getResultCount()` method to query the number of columns in the result set, after executing the SQL statement with the `open()` method and fetching a row with `fetch()`.

Example

```
DEFINE sh base.SqlHandle, i INT
...
FOR i=1 TO sh.getResultCount()
    DISPLAY sh.getResultName(i)
END FOR
```

For a complete example, see Example 2: SqlHandle with result set SQL on page 2446.

`base.SqlHandle.getResultName`
Returns the name of a column in the result set produced by the SQL statement.

Syntax

```
getResultName( index INTEGER )
RETURNS STRING
```

1. `index` is the ordinal position of the result set column (starts at 1).

Usage
Call the `getResultName()` method to query the name of a column in the result set, after executing the SQL statement with the `open()` method and fetching a row with `fetch()`.

The method takes the position of the column as the parameter.

Example

```
DEFINE sh base.SqlHandle, i INT
...
FOR i=1 TO sh.getResultCount()
    DISPLAY sh.getResultName(i)
END FOR
```

For a complete example, see Example 2: SqlHandle with result set SQL on page 2446.

`base.SqlHandle.getResultType`
Returns the Genero type name of a column in the result set produced by the SQL statement.

Syntax

```
getResultType( index INTEGER )
RETURNS STRING
```

1. `index` is the ordinal position of the result set column (starts at 1).

Usage
Call the `getResultType()` method to query the type of a column in the result set, after executing the SQL statement with the `open()` method and fetching a row with `fetch()`.

The method takes the position of the column as the parameter.

The type name is a string that represents a Genero type. For example, "INTEGER", "DECIMAL(10,2)", "DATE", "DATETIME YEAR TO SECOND".
Important:
The column type returned by the `getResultType()` method can differ, depending on the type of database server, the character set locale, and the length semantics used.

The native column type is provided by the database client software. This native type is then converted to a Genero type. For example, if you define a column with Oracle's native DATE type, the resulting type returned by `getResultType()` will be a Genero DATETIME YEAR TO SECOND, which corresponds to Oracle's DATE (YYYY-MM-DD hh:mm:ss).

Furthermore, depending on the character set and the length semantics used, the size of CHAR/VARCHAR types can differ, in order to get a Genero type that is large enough to hold the maximum character string the query may return. The resulting type can for example be different when using UTF-8 with CHAR or with BYTE length semantics. When using a single byte character set and BYTE length semantics, or when using UTF-8 and CHAR length semantics, the CHAR size returned by `getResultType()` will always match the size of the database column in character units. For more details, see Length semantics settings on page 521.

Example

```genero
DEFINE sh base.SqlHandle, i INT
...
FOR i=1 TO sh.getResultCount()
   DISPLAY sh.getResultType(i)
END FOR
```

For a complete example, see Example 2: SqlHandle with result set SQL on page 2446.

Related concepts

Defining the application locale on page 519
This section describes the settings defining the application locale, changing the behavior of the compilers and runtime system.

`base.SqlHandle.getResultValue`

Returns the value of a column in the result set produced by the SQL statement.

Syntax

```
getResultValue( index INTEGER )
RETURNS fgl-type
```

1. `index` is the ordinal position of the result set column (starts at 1).

Usage

Call the `getResultValue()` method to get the value of a column in the result set, after executing the SQL statement with the `open()` method and fetching a row with `fetch()`.

The method takes the position of the column as the parameter.

The value returned can be assigned to a program variable of the type corresponding to the type name returned by `getResultType()`.

**Important:** TEXT and BYTE values are returned by reference. In order to get the value of a TEXT or BYTE column, define a variable of this type and assign the `getResultValue()` return. The returned TEXT or BYTE variable is already located in memory, there is no need to LOCATE the variable before calling `getResultValue()`.

```genero
DEFINE p_text TEXT
...
LET p_text = h.getResultValue(3)
...
**Example**

```plaintext
DEFINE sh base.SqlHandle, i INT
...
FOR i=1 TO sh.getResultCount()
    DISPLAY sh.getResultValue(i)
END FOR
```

For a complete example, see Example 2: SqlHandle with result set SQL on page 2446.

`base.SqlHandle.open`
Opens the SQL handle (SELECT or INSERT cursor).

**Syntax**

`open()`

**Usage**

Call the `open()` method to execute the prepared SQL statement, and open the result set cursor or insert cursor.

The SQL statement must have been prepared with a `prepare()` call.

If the SQL statement contains `?` parameters:

- For a statement with a result set (SELECT), values must be provided for each parameter before the `open()` call.
- For an insert cursor, values must be provided after the `open()` call, before each `put()` call.

As with standard Genero SQL instructions, SQL errors can be trapped with `WHENEVER ERROR` or `TRY / CATCH` blocks and by testing `SQLCA.SQLCODE`.

**Example**

```plaintext
DEFINE sh base.SqlHandle
...
CALL sh.open()
```

For a complete example, see Example 2: SqlHandle with result set SQL on page 2446.

`base.SqlHandle.openScrollCursor`
Opens the SQL handle (with scrollable option).

**Syntax**

`openScrollCursor()`

**Usage**

Call the `openScrollCursor()` method to execute a prepared SQL statement, and open the result set for use with a scrollable SQL cursor.

The SQL statement must have been prepared with a `prepare()` call.

If the SQL statement contains `?` parameters, values must be provided for each parameter before the `openScrollCursor()` call.

After opening the scrollable cursor, use methods such as `fetchFirst()`, `fetchPrevious()` and `fetchAbsolute(n)` to move forwards and backwards in the SQL result set.

As with standard Genero SQL instructions, SQL errors can be trapped with `WHENEVER ERROR` or `TRY / CATCH` blocks and by testing `SQLCA.SQLCODE`.
**Example**

```sql
DEFINE sh base.SqlHandle
...
CALL sh.openScrollCursor()
```

*base.SqlHandle.prepare*

Prepares a SQL statement for the SQL handle.

**Syntax**

```sql
prepare(
   sql STRING )
```

1. *sql* is the SQL statement to be prepared.

**Usage**

Call the `prepare()` method to prepare the SQL statement that will be executed with either `execute()` or `open()`.

The SQL statement can contain `?` parameter place holders, to be filled with the `setParameter()` method before executing the statement.

As with standard Genero SQL instructions, SQL errors can be trapped with `WHENEVER ERROR` or `TRY / CATCH` blocks and by testing `SQLCA.SQLCODE`.

**Example**

```sql
DEFINE sh base.SqlHandle
...
CALL sh.prepare("INSERT INTO mytable VALUES (?,?)")
```

For a complete example, see Example 2: SqlHandle with result set SQL on page 2446.

*base.SqlHandle.put*

Put a new row in the insert cursor buffer.

**Syntax**

```sql
put ()
```

**Usage**

Call the `put()` method to create a new row for the insert cursor.

The SQL statement must have been prepared with a `prepare()` call.

All SQL parameter values must be provided before performing the `put()` call.

As with standard Genero SQL instructions, SQL errors can be trapped with `WHENEVER ERROR` or `TRY / CATCH` blocks and by testing `SQLCA.SQLCODE`.

**Example**

```sql
DEFINE sh base.SqlHandle
...
CALL sh.put()
```

For a complete example, see Example 3: SqlHandle with insert cursor on page 2446.
**base.SqlHandle.setParameter**
Sets the value of a SQL parameter for this SQL handle.

**Syntax**

```
setParameter(
    index INTEGER,
    value fgl-type )
```

1. `index` is the ordinal position of the ? SQL parameter (starts at 1).
2. `value` is the variable containing the parameter value.

**Usage**

Call the `setParameter()` method to define the value of a SQL parameter specified with a ? place holder in the string passed to the `prepare()` method.

The SQL statement must have been prepared with a `prepare()` call.

The method will raise error -8131, if the index passed as parameter is lower as 1 or greater as the number of ? parameter placeholders in the prepared SQL statement.

It is possible to pass numeric and string constants directly to the method, but type conversion cannot be done without a program variable.

**Example**

```
DEFINE sh base.SqlHandle
DEFINE v_pk INT, v_crea DATETIME YEAR TO SECOND
...
CALL sh.setParameter(1,v_pk)
CALL sh.setParameter(2,v_crea)
```

For a complete example, see Example 2: SqlHandle with result set SQL on page 2446.

**Usage**

The `base.SqlHandle` class is a built-in class providing dynamic SQL support with a 3GL API.

Compared to regular SQL cursor instructions, the main purpose of the `base.SqlHandle` class is to provide column name and SQL data type information with the `getResultName()` and `getResultType()` methods. It is also possible to write generic code for parameterized queries with the `setParameter()` method.

**Important:** A database connection must exist in order to use SqlHandle objects.

Unlike regular Genero cursors, SQL handle objects are created dynamically, and can be passed as parameter or returned from functions:

```
MAIN
    DEFINE h base.SqlHandle
    CONNECT TO "myDb"
    LET h = base.SqlHandle.create()
    CALL my_prepare(h)
    CALL my_execute(h)
END MAIN

FUNCTION my_prepare(h)
    DEFINE h base.SqlHandle
    CALL h.prepare("INSERT INTO cust VALUES ( ...
END FUNCTION

FUNCTION my_execute(h)
    DEFINE h base.SqlHandle
```
CALL h.execute()
END FUNCTION

Executing a simple SQL statement without a result set
Perform the following steps, to execute a SQL statement without a result set:

1. Define the SQL handle variable as `base.SqlHandle`
2. Create a SQL handle object `base.SqlHandle.create()`
3. `prepare(sql-text)`
4. For each SQL parameter:
   a. `setParameter(index, value)`
5. `execute()` -- test for `SQLCA.SQLCODE`
6. Repeat from (5), (4), or (3)

Executing a SQL statement returning a result set
Perform the following steps, to execute a SQL statement with a result set:

1. Define the SQL handle variable as `base.SqlHandle`
2. Create a SQL handle object `base.SqlHandle.create()`
3. `prepare(sql-text)`
4. For each SQL parameter:
   a. `setParameter(index, value)`
5. `open()`
6. `fetch()` -- test for `SQLCA.SQLCODE == 100`
7. `getResultCount()` -- for each column index:
   a. `getResultName(index)`
   b. `getResultType(index)`
   c. `getResultValue(index)`
8. `close()`
9. Repeat from (6), (4), (5), or (3)

Executing a SQL statement returning a result set, as scrollable cursor
Perform the following steps, to execute a SQL statement with a result set and scroll forwards and backwards in the rows:

1. Define the SQL handle variable as `base.SqlHandle`
2. Create a SQL handle object `base.SqlHandle.create()`
3. `prepare(sql-text)`
4. For each SQL parameter:
   a. `setParameter(index, value)`
5. `openScrollCursor()`
6. `fetch()` (next row), `fetchLast()`, `fetchFirst()`, `fetchPrevious()`, `fetchRelative(n)` or `fetchAbsolute(n)` -- test for `SQLCA.SQLCODE == 100`
7. `getResultCount()` -- for each column index:
   a. `getResultName(index)`
   b. `getResultType(index)`
   c. `getResultValue(index)`
8. `close()`
9. Repeat from (6), (4), (5), or (3)
Creating rows with an insert cursor

Perform the following steps, to insert many rows with a SQL handle insert cursor:

1. Define the SQL handle variable as `base.SqlHandle`
2. Create a SQL handle object `base.SqlHandle.create()`
3. `prepare(insert-stmt-with-params)`
4. `BEGIN WORK`
5. `open()`
6. For each row to insert:
   a. For each SQL parameter:
      1. `setParameter(index, value)`
   b. `put()`
7. `close()`
8. `COMMIT WORK`
9. Repeat from (4) or (3)

SQL error handling with SqlHandle

Handling SQL error and status information (such as NOTFOUND) can be done with SqlHandle objects as with regular SQL instruction, by testing the `SQLCA.SQLCODE` register, and by using TRY/CATCH blocks or WHENEVER ERROR.

```plaintext
MAIN
DEFINE h base.SqlHandle
CONNECT TO "mydb"
LET h = base.SqlHandle.create()
TRY
   CALL h.prepare("SELECT * FROM mytab")
   CALL h.open()
   CALL h.fetch()
   DISPLAY h.getResultValue(1)
   CALL h.close()
CATCH
   DISPLAY "SQL ERROR:", SQLCA.SQLCODE
END TRY
END MAIN
```

Related concepts

- **Result set processing** on page 715
  Shows how to fetch rows from a database query.
- **Dynamic SQL management** on page 709
  Explains how to execute and manage SQL statements at runtime.
- **SQL execution diagnostics** on page 591
  If an SQL statement execution fails, error description can be found in the `SQLCA.SQLCODE, SQLSTATE, STATUS` and `SQLERRMESSAGE` predefined registers.

Examples

- `base.SqlHandle` usage examples.

**Example 1: SqlHandle with simple SQL**

The following code executes a simple UPDATE statement with the `base.SqlHandle` API:

```plaintext
MAIN
DEFINE h base.SqlHandle
CONNECT TO "mydb"
LET h = base.SqlHandle.create()
```
CALL h.prepare("UPDATE t1 SET name = ? WHERE pk = ?")
CALL h.setParameter(1, "Scott")
CALL h.setParameter(2, "8723")
TRY
  CALL h.execute()
CATCH
  DISPLAY "Error detected: ", SQLCA.SQLCODE
END TRY
END MAIN

Example 2: SqlHandle with result set SQL
The following code executes a simple SELECT statement with the base.SqlHandle API:

MAIN
  DEFINE h base.SqlHandle,
      v VARCHAR(50),
      i INTEGER
  CONNECT TO ":memory:+driver='dbmsqt'"
  CALL create_table()
  LET h = base.SqlHandle.create()
  CALL h.prepare("SELECT * FROM t1 WHERE name>? ORDER BY pkey")

  LET v = "a"
  CALL h.setParameter(1, v)
  CALL h.open()
  WHILE TRUE
    CALL h.fetch()
    IF SQLCA.SQLCODE==NOTFOUND THEN
      EXIT WHILE
    END IF
    DISPLAY "-----------------
    FOR i=1 TO h.getResultCount()
      DISPLAY i, ":", h.getResultName(i),
            "/", h.getResultType(i),
            ":", h.getResultValue(i)
    END FOR
  END WHILE
  CALL h.close()
END MAIN

FUNCTION create_table()
  CREATE TABLE t1 ( pkey INTEGER PRIMARY KEY,
       name VARCHAR(50) )
  INSERT INTO t1 VALUES ( 101, 'aaaaa' )
  INSERT INTO t1 VALUES ( 102, 'bbbbbbbb' )
  INSERT INTO t1 VALUES ( 103, 'cccccc' )
  INSERT INTO t1 VALUES ( 104, 'ddddddd' )
END FUNCTION

Example 3: SqlHandle with insert cursor
The following code implements an insert cursor with the base.SqlHandle API:

MAIN
  DEFINE h base.SqlHandle
  DEFINE i INTEGER
  CONNECT TO ":memory:+driver='dbmsqt'"
  CREATE TABLE t1 ( pkey INTEGER, name VARCHAR(50))
  LET h = base.SqlHandle.create()
  CALL h.prepare("INSERT INTO t1 VALUES (?, ?)"
  BEGIN WORK
    CALL h.open()
FOR i=1 TO 10
    CALL h.setParameter(1, i)
    CALL h.setParameter(2, SFMT("item_%1",i))
    CALL h.put()
    IF i MOD 100 == 0 THEN
        CALL h.flush()
    END IF
END FOR
CALL h.close()
COMMIT WORK
SELECT COUNT(*) INTO i FROM t1
DISPLAY i
END MAIN

Example 4: SqlHandle with scroll cursor

The following code implements an insert cursor with the base.SqlHandle API:

MAIN
    DEFINE h base.SqlHandle
    CONNECT TO ":memory:+driver='dbmsqt'"
    CALL create_table()
    LET h = base.SqlHandle.create()
    CALL h.prepare("SELECT * FROM t1 ORDER BY pkey")
    CALL h.openScrollCursor()
    CALL h.fetchFirst()
    DISPLAY h.getResultValue(1), " / ", h.getResultValue(2)
    CALL h.fetchLast()
    DISPLAY h.getResultValue(1), " / ", h.getResultValue(2)
    CALL h.fetchPrevious()
    DISPLAY h.getResultValue(1), " / ", h.getResultValue(2)
    CALL h.fetch()
    DISPLAY h.getResultValue(1), " / ", h.getResultValue(2)
    CALL h.fetchAbsolute(2)
    DISPLAY h.getResultValue(1), " / ", h.getResultValue(2)
    CALL h.fetchRelative(2)
    DISPLAY h.getResultValue(1), " / ", h.getResultValue(2)
    CALL h.close()
END MAIN

FUNCTION create_table()
    CREATE TABLE t1 ( pkey INTEGER PRIMARY KEY,
                        name VARCHAR(50) )
    INSERT INTO t1 VALUES ( 101, 'aaaaa' )
    INSERT INTO t1 VALUES ( 102, 'bbbbbbbb' )
    INSERT INTO t1 VALUES ( 103, 'cccccc' )
    INSERT INTO t1 VALUES ( 104, 'ddddddd' )
END FUNCTION

The StringBuffer class

The base.StringBuffer class is a built-in class designed to manipulate character strings.

This class is optimized for string operations such as scanning, replacements, concatenation.

Use the base.StringBuffer class instead of STRING variables to implement heavy-duty string manipulations. When you use a base.StringBuffer object, you work directly on the internal string buffer. When you use the STRING data type and modify a string, the runtime system creates a new buffer. While this does not impact the performance of programs with a user interface or even batch programs doing SQL, it can impact performance when you need to rapidly process large character strings. For example, if you need to process 500 KB of text (such as when you are performing a global search-and-replace of specific words), you get much better performance with a base.StringBuffer object than you would with a STRING variable.
When you pass a `base.StringBuffer` object as a function parameter, the function receives a variable that references the object. Passing the object by reference is much more efficient than using a `STRING` that is passed by value, because `STRING` data is copied on the stack. The function manipulates the original string, not a copy of the string.

**Important:** The methods of this class use character positions and string length. When using byte length semantics, the length is expressed in bytes. When using char length semantics, the unit is characters. This matters when using a multibyte locale such as UTF-8.

**`base.StringBuffer` methods**

**Table 516: Class methods**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>base.StringBuffer.create()</code></td>
<td>Create a string buffer object.</td>
</tr>
<tr>
<td>RETURNS <code>base.StringBuffer</code></td>
<td></td>
</tr>
</tbody>
</table>
### Table 517: Object methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>append (str STRING)</td>
<td>Append a string at the end of the current string.</td>
</tr>
<tr>
<td>clear ()</td>
<td>Clear the string buffer.</td>
</tr>
<tr>
<td>equals (str STRING)</td>
<td>Compare strings (case sensitive).</td>
</tr>
<tr>
<td>equalsIgnoreCase (str STRING)</td>
<td>Returns BOOLEAN</td>
</tr>
<tr>
<td>getCharAt (index INTEGER)</td>
<td>Return the character at a specified position.</td>
</tr>
<tr>
<td>getIndexOf (part STRING, start INTEGER)</td>
<td>Return the position of a substring.</td>
</tr>
<tr>
<td>getLength ()</td>
<td>Return the length of a string.</td>
</tr>
<tr>
<td>insertAt (index INTEGER, str STRING)</td>
<td>Insert a string at a given position.</td>
</tr>
<tr>
<td>replace (oldStr STRING, newStr STRING, occurrences INTEGER)</td>
<td>Replace one string with another.</td>
</tr>
<tr>
<td>replaceAt (index INTEGER, length INTEGER, str STRING)</td>
<td>Replace part of a string with another string.</td>
</tr>
<tr>
<td>subString (startIndex INTEGER, endIndex INTEGER)</td>
<td>Return the substring at the specified position.</td>
</tr>
<tr>
<td>toLowerCase ()</td>
<td>Converts the string in the buffer to lower case.</td>
</tr>
<tr>
<td>toUpperCase ()</td>
<td>Converts the string in the buffer to upper case.</td>
</tr>
<tr>
<td>toString ()</td>
<td>Create a STRING from the string buffer.</td>
</tr>
<tr>
<td>trim ()</td>
<td>Remove leading and trailing blank space (ASCII 32) characters.</td>
</tr>
<tr>
<td>trimWhiteSpace ()</td>
<td>Remove leading and trailing whitespace characters.</td>
</tr>
<tr>
<td>trimLeft ()</td>
<td>Removes leading blank space (ASCII 32) characters.</td>
</tr>
<tr>
<td>trimLeftWhiteSpace ()</td>
<td>Removes leading whitespace characters.</td>
</tr>
<tr>
<td>trimRight ()</td>
<td>Removes trailing blank space (ASCII 32) characters.</td>
</tr>
<tr>
<td>trimRightWhiteSpace ()</td>
<td>Removes trailing whitespace characters.</td>
</tr>
</tbody>
</table>
**base.StringBuffer.create**
Create a string buffer object.

**Syntax**

```plaintext
base.StringBuffer.create()
RETURNS base.StringBuffer
```

**Usage**

Use the `base.StringBuffer.create()` class method to create a string buffer object. The created object must be assigned to a program variable defined with the `base.StringBuffer` type.

**Example**

```plaintext
DEFINE buf base.StringBuffer
LET buf = base.StringBuffer.create()
...
```

For a complete example, see Example 2: Modify a StringBuffer with a function on page 2462.

**base.StringBuffer.append**
Append a string at the end of the current string.

**Syntax**

```plaintext
append(  
  str STRING  
)
```

1. *str* is the string to append to the string buffer.

**Usage**

The `append()` method appends a string to the internal string buffer.

**Example**

```plaintext
DEFINE buf base.StringBuffer
LET buf = base.StringBuffer.create()
CALL buf.append("abc")
...
```

For a complete example, see Example 1: Add strings to a StringBuffer on page 2461.

**base.StringBuffer.clear**
Clear the string buffer.

**Syntax**

```plaintext
clear()
```

**Usage**

Use the `clear()` method to clear the string buffer.

After clearing, the string buffer is empty and the length is zero.
Example

```plaintext
DEFINE buf base.StringBuffer
CALL buf.clear()
...
```

`base.StringBuffer.equals`
Compare strings (case sensitive).

Syntax

```plaintext
equals(
  str STRING
) RETURNS BOOLEAN
```

1. `str` is the string to compare with.

Usage

Use the `equals()` method to determine whether the value of a `base.StringBuffer` object is identical to a specified string.

This method is case-sensitive.

Since the parameter for the method must be a string, you can use the `toString()` method to convert a `base.StringBuffer` object in order to compare it.

The method returns `TRUE` if the strings are identical, otherwise it returns `FALSE`.

Example

```plaintext
MAIN
  DEFINE buf, buf2 base.StringBuffer,
       mystring STRING

  LET buf = base.StringBuffer.create()
  CALL buf.append("there")

  -- compare to a STRING
  IF buf.equals("there") THEN
    DISPLAY "buf matches there"
  END IF

  -- compare to a STRING variable
  LET mystring = "there"
  IF buf.equals(mystring) THEN
    DISPLAY "buf matches mystring"
  END IF

  -- compare to another StringBuffer object
  LET buf2 = base.StringBuffer.create()
  CALL buf2.append("there")
  IF buf.equals(buf2.toString()) THEN
    DISPLAY "buf matches buf2"
  END IF
END MAIN
```

Output:

```
buf matches there
buf matches mystring
buf matches buf2
```
base.StringBuffer.equalsIgnoreCase
Compare strings (case insensitive)

Syntax

```plaintext
equalsIgnoreCase(
    str STRING
) 
RETURNS BOOLEAN
```

1. `str` is the string to compare with.

Usage

The `equalsIgnoreCase()` method compares the current string buffer with the passed string, ignoring the character case.

Since the parameter for the method must be a string, you can use the `toString()` method to convert a `base.StringBuffer` object in order to compare it.

The method returns `TRUE` if the strings are identical, otherwise it returns `FALSE`.

Example

```plaintext
MAIN
    DEFINE buf3 base.StringBuffer
    LET buf3 = base.StringBuffer.create()
    CALL buf3.append("there")
    IF buf3.equalsIgnoreCase("There") THEN
       DISPLAY "buf matches There ignoring case"
    END IF
END MAIN
```

Output:

```
buf matches There ignoring case
```

base.StringBuffer.getCharAt
Return the character at a specified position.

Syntax

```plaintext
getCharAt(
    index INTEGER
) 
RETURNS STRING
```

1. `index` is the character position in the string.

Usage

The `getCharAt()` method returns the character from the string buffer at the position that you specify.

The first character position is 1.

The method returns `NULL` if the position is lower as 1 or greater as the length of the string.

**Important:** When using byte length semantics, the position is expressed in bytes. When using char length semantics, the unit is characters. This matters when using a multibyte locale such as UTF-8.

Example

```plaintext
MAIN
    DEFINE buf base.StringBuffer
```
LET buf = base.StringBuffer.create()
CALL buf.append("abcdef")
DISPLAY buf.getCharAt(3)
END MAIN

Output:
c

base.StringBuffer.indexOf
Return the position of a substring.

Syntax

indexOf(
    part STRING,
    start INTEGER )
RETURNS INTEGER

1. part is the substring to be found.
2. start is the starting position.

Usage

The indexOf() method returns the position of a substring in the string buffer. Specify the substring and an integer specifying the position at which the search should begin. Use 1 if you want to start at the beginning of the string buffer.

The method returns zero if the substring is not found.

CALL buf.append("abcdef")
DISPLAY buf.indexOf("def",1) -- Shows 4

Important: When using byte length semantics, the position is expressed in bytes. When using char length semantics, the unit is characters. This matters when using a multibyte locale such as UTF-8.

Example

This example iterates through the complete string to display the position of multiple occurrences of the same substring.

MAIN
DEFINE buf base.StringBuffer
DEFINE pos INTEGER
DEFINE s STRING
LET buf = base.StringBuffer.create()
CALL buf.append("---abc-----abc--abc----")
LET pos = 1
LET s = "abc"
WHILE TRUE
    LET pos = buf.indexOf(s,pos)
    IF pos == 0 THEN
        EXIT WHILE
    END IF
    DISPLAY "Pos: ", pos
    LET pos = pos + length(s)
END WHILE
END MAIN
base.StringBuffer.getLength
Return the length of a string.

Syntax

```java
getLength()
RETURNS INTEGER
```

Usage

Use the `getLength()` method to return the number of characters in the current string buffer, including trailing spaces.

The length of an empty string buffer is 0.

**Important:** When using byte length semantics, the string length is expressed in bytes. When using char length semantics, the unit is characters. This matters when using a multibyte locale such as UTF-8.

Example

```java
MAIN
    DEFINE buf base.StringBuffer
    LET buf = base.StringBuffer.create()
    CALL buf.append("abc")
    DISPLAY buf.getLength()
    -- append three spaces to the end of the string
    CALL buf.append("   ")
    DISPLAY buf.getLength()
END MAIN
```

Output:

```
3
6
```

base.StringBuffer.insertAt
Insert a string at a given position.

Syntax

```java
insertAt(
    index INTEGER,
    str STRING )
```

1. `index` is the position where the string must be inserted.
2. `str` is the string to be inserted.

Usage

The `insertAt()` method inserts a string before the specified position in the string buffer.

**Important:** When using byte length semantics, the position is expressed in bytes. When using char length semantics, the unit is characters. This matters when using a multibyte locale such as UTF-8.
Example

```main
DEFINE buf base.StringBuffer
LET buf = base.StringBuffer.create()
CALL buf.append("abcdef")
CALL buf.insertAt(3, "xx")
DISPLAY buf.toString()
END MAIN
```

Output:

```
abxxcdef
```

`base.StringBuffer.replace`

Replace one string with another.

Syntax

```replace(
    oldStr STRING,
    newStr STRING,
    occurrences INTEGER )
```

1. `oldStr` is the string to be replaced.
2. `newStr` is the new string replacing the old string.
3. `occurrences` is the number of replacements to do (zero for all).

Usage

The `replace()` method replaces a string within the current string buffer with a different string. Specify the original string, replacement string, and the number of occurrences to replace. Use 0 to replace all occurrences.

Example

```main
DEFINE buf base.StringBuffer
LET buf = base.StringBuffer.create()
CALL buf.append("aaxxbbxxcc")
CALL buf.replace("xx", "zz", 1)
DISPLAY buf.toString()
END MAIN
```

Output:

```
aazzbbxxcc
```

Related concepts

`base.StringBuffer.replaceAt` on page 2455

Replace part of a string with another string.

`base.StringBuffer.replaceAt`

Replace part of a string with another string.

Syntax

```replaceAt(
    index INTEGER, 
    length INTEGER,
)```
1. `index` is position where the replacement starts.
2. `length` is the number of characters to be replaced.
3. `str` is the replacement string.

**Usage**

The `replaceAt()` method replaces part of the current string with another string. The parameters are integers indicating the position at which the replacement should start, the number of characters to be replaced, and the replacement string.

The first position in the string is 1.

**Important:** When using byte length semantics, the position and length are expressed in bytes. When using char length semantics, the unit is characters. This matters when using a multibyte locale such as UTF-8.

**Example**

```plaintext
MAIN
    DEFINE buf base.StringBuffer
    LET buf = base.StringBuffer.create()
    CALL buf.append("abxxxxef")
    CALL buf.replaceAt(3,4,"cd")
    DISPLAY buf.toString()
END MAIN
```

Output:

```
abcdef
```

**Related concepts**

- `base.StringBuffer.replace` on page 2455
  Replace one string with another.

- `base.StringBuffer.subString`
  Return the substring at the specified position.

**Syntax**

```plaintext
subString(
    startIndex INTEGER,
    endIndex INTEGER )
RETURNS STRING
```

1. `startIndex` is the substring to be found.
2. `endIndex` is the ending position.

**Usage**

The `subString()` method returns the substring defined by the start and end positions passed as parameter.

The first character is at position 1.

**Important:** When using byte length semantics, the positions are expressed in bytes. When using char length semantics, the unit is characters. This matters when using a multibyte locale such as UTF-8.

**Example**

```plaintext
MAIN
```

```plaintext
```
DEFINE buf base.StringBuffer
LET buf = base.StringBuffer.create()
CALL buf.append("abcdefg")
DISPLAY buf.subString(2,5)
END MAIN

Output:

bcde

base.StringBuffer.toLowerCase
Converts the string in the buffer to lower case.

Syntax

toLowerCase()

Usage
The toLowerCase() method converts the current string to lower case.

Example

MAIN
  DEFINE buf base.StringBuffer
  LET buf = base.StringBuffer.create()
  CALL buf.append("AbC")
  CALL buf.toLowerCase()
  DISPLAY buf.toString()
END MAIN

Output:

abc

base.StringBuffer.toString
Create a STRING from the string buffer.

Syntax

toString()
  RETURNS STRING

Usage
The toString() method creates a STRING value from the current string buffer.
Use this method if you need to pass the string to another method or instruction that expects a STRING as parameter.

Example

MAIN
  DEFINE buf base.StringBuffer
  LET buf = base.StringBuffer.create()
  CALL buf.append("abc")
  DISPLAY buf.toString()
END MAIN
Output:

```
abc
```

`base.StringBuffer.toUpperCase`
Converts the string in the buffer to upper case.

**Syntax**

```
toUpperCase()
```

**Usage**
The `toUpperCase()` method converts the current string to upper case.

**Example**

```
MAIN
  DEFINE buf base.StringBuffer
  LET buf = base.StringBuffer.create()
  CALL buf.append("AbC")
  CALL buf.toUpperCase()
  DISPLAY buf.toString()
END MAIN
```

Output:

```
ABC
```

`base.StringBuffer.trim`
Remove leading and trailing blank space (ASCII 32) characters.

**Syntax**

```
trim()
```

**Usage**
The `trim()` method removes the leading and trailing blank space characters in the string buffer.

**Note:** The method removes only blank space (ASCII 32) characters. Characters tab (\t), newline (\n), carriage-return (\r) and form-feed (\f) are not removed.

**Example**

```
MAIN
  DEFINE buf base.StringBuffer
  LET buf = base.StringBuffer.create()
  CALL buf.append(" abc ")
  CALL buf.trim()
  DISPLAY "["||buf.toString()||"]" -- Shows [abc]
END MAIN
```

Output:

```
[abc]
```

**Related concepts**

`base.StringBuffer.trimWhiteSpace` on page 2459
Remove leading and trailing whitespace characters.

`base.StringBuffer.trimWhiteSpace`

Remove leading and trailing whitespace characters.

**Syntax**

```
trimWhiteSpace()
```

**Usage**

The `trimWhiteSpace()` method removes the leading and trailing whitespace characters in the string buffer.

**Note:** The method considers as whitespace characters all characters less than or equal to blank space (ASCII 32). This includes tab (\t), newline (\n), carriage-return (\r) and form-feed (\f).

**Example**

```
MAIN
   DEFINE buf base.StringBuffer
   LET buf = base.StringBuffer.create()
   CALL buf.append("\n\t  abc  \n\t")
   CALL buf.trimWhiteSpace()
   DISPLAY "["||buf.toString()||"]" -- Shows [abc]
END MAIN
```

**Output:**

```
[abc]
```

**Related concepts**

- `base.StringBuffer.trim` on page 2458
  Remove leading and trailing blank space (ASCII 32) characters.

- `base.StringBuffer.trimLeft`
  Removes leading blank space (ASCII 32) characters.

**Syntax**

```
trimLeft()
```

**Usage**

The `trimLeft()` method removes the leading blank space characters in the string buffer.

**Note:** The method removes only blank space (ASCII 32) characters. Characters tab (\t), newline (\n), carriage-return (\r) and form-feed (\f) are not removed.

**Example**

```
MAIN
   DEFINE buf base.StringBuffer
   LET buf = base.StringBuffer.create()
   CALL buf.append(" abc ")
   CALL buf.trimLeft()
   DISPLAY "["||buf.toString()||"]"
END MAIN
```
Output:

```
[abc ]
```

**Related concepts**

- `base.StringBuffer.trimLeftWhiteSpace` on page 2460
  Removes leading whitespace characters.

- `base.StringBuffer.trimLeftWhiteSpace`
  Removes leading whitespace characters.

**Syntax**

```java
trimLeftWhiteSpace()
```

**Usage**

The `trimLeftWhiteSpace()` method removes the leading whitespace characters in the string buffer.

**Note:** The method considers as whitespace characters all characters less than or equal to blank space (ASCII 32). This includes tab (\t), newline (\n), carriage-return (\r) and form-feed (\f).

**Example**

```java
MAIN
  DEFINE buf base.StringBuffer
  LET buf = base.StringBuffer.create()
  CALL buf.append("\n\t abc ")
  CALL buf.trimLeftWhiteSpace()
  DISPLAY ": |buf.toString()| ]" END MAIN
```

Output:

```
[abc ]
```

**Related concepts**

- `base.StringBuffer.trimLeft` on page 2459
  Removes leading blank space (ASCII 32) characters.

- `base.StringBuffer.trimRight`
  Removes trailing blank space (ASCII 32) characters.

**Syntax**

```java
trimRight()
```

**Usage**

The `trimRight()` method removes the trailing blank space characters in the string buffer.

**Note:** The method removes only blank space (ASCII 32) characters. Characters tab (\t), newline (\n), carriage-return (\r) and form-feed (\f) are not removed.

**Example**

```java
MAIN
  DEFINE buf base.StringBuffer
  LET buf = base.StringBuffer.create()
```
CALL buf.append("  abc  ")
CALL buf.trimRight()
DISPLAY "}"||buf.toString()||""
END MAIN

Output:

[  abc]

Related concepts
base.StringBuffer.trimRightWhiteSpace on page 2461
Removes trailing whitespace characters.

base.StringBuffer.trimRightWhiteSpace
Removes trailing whitespace characters.

Syntax

trimRightWhiteSpace()

Usage

The trimRightWhiteSpace() method removes the trailing whitespace characters in the string buffer.

Note: The method considers as whitespace characters all characters less than or equal to blank space (ASCII 32). This includes tab (\t), newline (\n), carriage-return (\r) and form-feed (\f).

Example

MAIN
  DEFINE buf base.StringBuffer
  LET buf = base.StringBuffer.create()
  CALL buf.append("  abc  \n\t")
  CALL buf.trimRightWhiteSpace()
  DISPLAY "}"||buf.toString()||""
END MAIN

Output:

[  abc]

Related concepts
base.StringBuffer.trimRight on page 2460
Removes trailing blank space (ASCII 32) characters.

Examples
base.StringBuffer usage examples.
Example 1: Add strings to a StringBuffer

MAIN
  DEFINE buf base.StringBuffer
  LET buf = base.StringBuffer.create()
  CALL buf.append("abc")
  DISPLAY buf.toString()
  CALL buf.append("def")
  DISPLAY buf.toString()
  CALL buf.append(123456)
  DISPLAY buf.toString()
END MAIN
Example 2: Modify a StringBuffer with a function

```plaintext
MAIN
  DEFINE buf base.StringBuffer
  LET buf = base.StringBuffer.create()
  CALL modify(buf)
  DISPLAY "buf is ", buf.toString()
END MAIN

FUNCTION modify(sb)
  DEFINE sb base.StringBuffer
  CALL sb.append("more")
  DISPLAY "sb is ", sb.toString()
END FUNCTION
```

Output:

```
sb is more
buf is more
```

The `StringTokenizer` class

The `base.StringTokenizer` class is designed to parse a string to extract tokens based on delimiters.

The steps to use a string tokenizer are:

1. Define a variable of the `base.StringTokenizer` type.
2. Create the string tokenizer object with one of the create methods, passing the string to be parsed as parameter.
3. Optionally, count the number of tokens with `countTokens()` before processing.
4. Use a `WHILE` loop to process the different tokens, by using `hasMoreTokens()` as loop condition and `nextToken()` inside the loop body to get the next token.

`base.StringTokenizer` methods

Table 518: Class methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>base.StringTokenizer.create</code></td>
<td>Create a string tokenizer object.</td>
</tr>
<tr>
<td>(<code>str STRING, delimiters STRING</code>)</td>
<td></td>
</tr>
<tr>
<td>RETURNS <code>base.StringTokenizer</code></td>
<td></td>
</tr>
<tr>
<td><code>base.StringTokenizer.createExt</code></td>
<td>Create a string tokenizer object with escape char and null handling.</td>
</tr>
<tr>
<td>(<code>str STRING, delimiters STRING, escapeChar STRING, withNulls BOOLEAN</code>)</td>
<td></td>
</tr>
<tr>
<td>RETURNS <code>base.StringTokenizer</code></td>
<td></td>
</tr>
</tbody>
</table>
Table 519: Object methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>countTokens()</td>
<td>RETURNS INTEGER</td>
</tr>
<tr>
<td></td>
<td>Returns the number of tokens left to be returned.</td>
</tr>
<tr>
<td>hasMoreTokens()</td>
<td>RETURNS BOOLEAN</td>
</tr>
<tr>
<td></td>
<td>Returns TRUE if there are more tokens to return.</td>
</tr>
<tr>
<td>nextToken()</td>
<td>RETURNS STRING</td>
</tr>
<tr>
<td></td>
<td>Returns the next token found in the source string.</td>
</tr>
</tbody>
</table>

**base.StringTokenizer.create**

Create a string tokenizer object.

**Syntax**

```plaintext
base.StringTokenizer.create (str STRING, delimiters STRING) RETURNS base.StringTokenizer
```

1. *str* is the character string to be parsed.
2. *delimiters* defines the delimiters to be used.

**Usage**

Use the `base.StringTokenizer.create()` class method to create a string tokenizer object. The created object must be assigned to a program variable defined with the `base.StringTokenizer` type. The method can take a unique or multiple delimiters into account. A delimiter is always one character long. The empty tokens are not taken into account, and no escape character is defined for the delimiters. The `nextToken()` method will never return NULL strings.

**Note:** To specify a backslash as a delimiter, you must use double backslashes in both the source string and as the delimiter, as shown in Example 3: Specify a backslash as a delimiter on page 2465

**Example**

```plaintext
DEFINE tok base.StringTokenizer
   -- Using a single pipe delimiter
LET tok = base.StringTokenizer.create("aaa|bbb|ccc","|")
   -- Using several delimiters
LET tok = base.StringTokenizer.create("aaa|bbb;ccc+ddd","|;")
```

For a complete example, see Example 1: Split a UNIX directory path on page 2465.

**base.StringTokenizer.createExt**

Create a string tokenizer object with escape char and null handling.

**Syntax**

```plaintext
base.StringTokenizer.createExt (str STRING, delimiters STRING, escapeChar STRING)
```
**delimiters** STRING,
**escapeChar** STRING,
**withNulls** BOOLEAN
) RETURN base.StringTokenizer

1. *str* is the character string to be parsed.
2. *delimiters* defines the delimiters to be used.
3. *escapeChar* defines the character to escape a delimiter.
4. *withNulls* indicates if empty tokens must be returned when there is nothing between delimiters.

**Usage**

Use the `base.StringTokenizer.createExt()` class method to create a string tokenizer object, with escape character and null token handling.

The created object must be assigned to a program variable defined with the `base.StringTokenizer` type.

The method can take a unique or multiple delimiters into account. A delimiter is always one character long.

When defining an escape character with the third parameter, the delimiters can be escaped in the source string.

When passing `TRUE` for the `withNulls` parameter, empty tokens are taken into account. The `nextToken()` method might return NULL strings. In the source string, leading and trailing delimiters or the amount of delimiters between two tokens affects the number of tokens.

**Note:** To specify a backslash as a delimiter, you must use double backslashes in both the source string and as the delimiter, as shown in Example 3: Specify a backslash as a delimiter on page 2465

**Example**

```define
tok base.StringTokenizer
let tok = base.StringTokenizer.createExt("aaa | b \bb \ccc", " |", " ", TRUE)
```

For a complete example, see Example 2: Escaped delimiters and NULL tokens on page 2465.

**base.StringTokenizer.countTokens**

Returns the number of tokens left to be returned.

**Syntax**

```base
countTokens ()
RETURNS INTEGER
```

**Usage**

Use the `countTokens()` method to count the number of tokens left to be returned by the string tokenizer.

This method can be used to get the number of tokens before processing the source string with the `hasMoreTokens()` and `nextToken()` methods.

**base.StringTokenizer.hasMoreTokens**

Returns `TRUE` if there are more tokens to return.

**Syntax**

```base
hasMoreTokens ()
RETURNS BOOLEAN
```

**Usage**

The `hasMoreTokens()` method indicates if there are other tokens in the source string that are not yet processed.
Use the `hasMoreTokens()` method typically as the expression of a `WHILE` block.

`base.StringTokenizer.nextToken`

Returns the next token found in the source string.

**Syntax**

```plaintext
nextToken()  
RETURNS STRING
```

**Usage**

The `nextToken()` method parses the source string for tokens, following the creation method used, and returns the next token if found.

The method returns `NULL` if no token is found, or if an empty token was found and the `nulls` parameter of the `createExt()` method was set to `TRUE`.

Use the `hasMoreTokens()` method to check if more tokens are to be read.

**Examples**

`base.StringTokenizer` usage examples.

**Example 1: Split a UNIX™ directory path**

```
MAIN
    DEFINE tok base.StringTokenizer
    LET tok = base.StringTokenizer.create("/home/tomy","/")
    WHILE tok.hasMoreTokens()
        DISPLAY tok.nextToken()
    END WHILE
END MAIN
```

**Example 2: Escaped delimiters and NULL tokens**

```
MAIN
    DEFINE tok base.StringTokenizer
    LET tok = base.StringTokenizer.createExt("||aaa||bbc|","","\",TRUE)
    WHILE tok.hasMoreTokens()
        DISPLAY tok.nextToken()
    END WHILE
END MAIN
```

**Example 3: Specify a backslash as a delimiter**

```
MAIN
    DEFINE tok base.StringTokenizer
    LET tok = base.StringTokenizer.create("C:\My Documents\My Pictures","\")
    WHILE tok.hasMoreTokens()
        DISPLAY tok.nextToken()
    END WHILE
END MAIN
```

**The TypeInfo class**

The `base.TypeInfo` class creates a DOM node from a structured program variable.

You can use this class to do program variables introspection, in order to get the data type names and values in a DOM node, that can be traversed with the XML utility classes, or to be serialized in a file for export purpose.

The `base.TypeInfo` class does not have to be instantiated: you can directly use the `create()` method.
A serialized DOM node created by `base.TypeInfo` class looks for example as follows:

```xml
<?xml version="1.0" encoding="ISO-8859-1">
<Record>
  <Field type="INTEGER" value="234" name="key"/>
  <Field type="CHAR(20)" value="Johnson" name="lastname"/>
  <Field type="DATE" value="12/24/1962" name="birthdate"/>
</Record>
```

Steps to use the class:

- Define a variable with the `om.DomNode` type.
- Create a DOM node with `base.TypeInfo.create(var)` and assign it to the DOM node variable.
- Use the created DOM node.

For example, to convert a list of database records to XML, fetch rows from a database table in a structured array, specify the array as the input parameter for the `base.TypeInfo.create()` method to create a new `base.DomNode` object, and serialize the resulting DOM node to a file by using the `node.writeXml()` method. You can then pass the resulting file to any application that is able to read XML for input.

**Note:** Consider using the JSON interface to serialize and deserialize program variables.

### base.TypeInfo methods

**Table 520: Class methods**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| **base.TypeInfo.create** (  
  `field` {  
    primitive-type  
    record-type  
    array-type  
    dictionary-type  
  }  
)  
RETURNS `om.DomNode` | Create a DomNode with the type information and values of a program variable. |

**Important:** This feature is deprecated, its use is discouraged although not prohibited.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| **base.TypeInfo.describe** (  
  `field` {  
    primitive-type  
    record-type  
    array-type  
    dictionary-type  
  }  
) | Display the type information and values of a program variable. |

**base.TypeInfo.create()**

Create a DomNode with the type information and values of a program variable.

**Syntax**

```java
base.TypeInfo.create(  
  field {  
    primitive-type  
    record-type  
    array-type  
    dictionary-type  
  }  
)
```
1. *field* is the program variable to convert to DOM.
2. *primitive-type* is a primitive data type of Genero (INTEGER, DATE, VARCHAR)
3. *record-type* is a RECORD ... END RECORD type.
4. *array-type* is a DYNAMIC ARRAY OF ... or ARRAY[n] OF ... type.
5. *dictionary-type* is a DICTIONARY OF ... type.

**Usage**

Use the `base.TypeInfo.create()` class method to create an `om.DomNode` object from a program variable.

The DOM node contains type information and values of the program variable.

The program variable provided to the method is typically a RECORD, but it can be any sort of structured variable, including arrays.

The data is formatted based on current environment settings (DBDATE, DBFORMAT, and DBMONEY).

The method trims trailing blanks for STRING, CHAR and VARCHAR data, and therefore produces empty strings in the resulting DOM elements, when the source variable contains only blanks.

**Example**

```genero
MAIN
  DEFINE n om.DomNode
  DEFINE r RECORD
    key INTEGER, 
    lastname CHAR(20),
    birthdate DATE, 
    comment VARCHAR(200)
  END RECORD
  LET r.key = 234
  LET r.lastname = "Johnson"
  LET r.birthdate = MDY(12,24,1962)
  LET r.comment = "   
  LET n = base.TypeInfo.create( r )
  DISPLAY n.toString()
END MAIN
```

The generated node contains variable values and data type information:

```xml
<xml version="1.0" encoding="ISO-8859-1">
  <Record>
    <Field type="INTEGER" value="234" name="key"/>
    <Field type="CHAR(20)" value="Johnson" name="lastname"/>
    <Field type="DATE" value="12/24/1962" name="birthdate"/>
    <Field type="VARCHAR(200)" value="" name="comment"/>
  </Record>
</xml>
```

**Related concepts**

- The `DomNode` class on page 2584
- The `om.DomNode` class provides methods to manipulate a DOM node of a data tree.
- Records on page 408
- Records allow structured program variables definitions.
- Arrays on page 418
Arrays (static or dynamic) allow you to handle an ordered collection of elements.

`base.TypeInfo.describe()`
Display the type information and values of a program variable.

**Syntax**

**Important:** This feature is deprecated, its use is discouraged although not prohibited.

```java
base.TypeInfo.describe(
    field \| primitive-type
    \| record-type
    \| array-type
    \| dictionary-type
)
```

1. *field* is the program variable to convert to DOM.
2. *primitive-type* is a primitive data type of Genero (INTEGER, DATE, VARCHAR)
3. *record-type* is a RECORD ... END RECORD type.
4. *array-type* is a DYNAMIC ARRAY OF ... or ARRAY[n] OF ... type.
5. *dictionary-type* is a DICTIONARY OF ... type.

**Usage**

Use the `base.TypeInfo.describe()` class method to display the type definition and values of the program variable passed as argument. Type information and values are written to the stdout stream.

The method follows the same data formatting rules as `base.TypeInfo.create()` on page 246.

**Note:** The `base.TypeInfo.describe()` method is deprecated. As replacement, use the `base.TypeInfo.create()` method to create an `om.DomNode` object, and produce the XML string with the `om.DomNode.toString()` method.

**Example**

```java
MAIN
    DEFINE r RECORD
    key INTEGER,
    lastname CHAR(20),
    birthdate DATE,
    comment VARCHAR(200)
END RECORD
LET r.key = 234
LET r.lastname = "Johnson"
LET r.birthdate = MDY(12,24,1962)
LET r.comment = "   
CALL base.TypeInfo.describe( r )
END MAIN
```

**Output:**

Field type is:
```
RECORD (4 members)
    key:INTEGER
    lastname:CHAR(20)
    birthdate:DATE
    comment:VARCHAR(200)
```
Field type end.
Related concepts

`base>TypeInfo.create()` on page 2466
Create a DomNode with the type information and values of a program variable.

The `MessageServer` class

The `base.MessageServer` class allows a program to send a key action over the network to other programs using this service.

This class can be used to join a group of programs to be notified by simple messages (key events). The programs can run on different machines connected together in a network.

**Important:** This feature is experimental and subject to change.

The `base.MessageServer` uses network API capabilities with Sockets and the UDP protocol. The computers must be configured with a network. The UDP protocol does not guarantee the transmission of datagrams, therefore messages sent with the MessageServer can arrive out of order, duplicated, or go missing without notice.

The UDP port is 6600 and the IP address group is 224.0.1.1. These cannot be changed.

**Important:** This feature is only supported in direct connection with the GDC front-end. It is not supported when using other front-ends or when using the GAS.

`base.MessageServer` methods

**Table 521: Class methods**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>base.MessageServer.connect()</code></td>
<td>Connects to the group of programs to be notified by a message.</td>
</tr>
<tr>
<td><code>base.MessageServer.send(message STRING)</code></td>
<td>Sends a key event to the group of programs connected together.</td>
</tr>
</tbody>
</table>

`base.MessageServer.connect`

Connects to the group of programs to be notified by a message.

**Syntax**

`base.MessageServer.connect()`

**Usage**

Use the `connect()` method to join the group of programs that can be notified by a key event message.

`base.MessageServer.send`

Sends a key event to the group of programs connected together.

**Syntax**

`base.MessageServer.send(message STRING)`

1. `message` is a string expression defining the key event to be sent over the network.
**Usage**

Once connected to the message server group with `base.MessageServer.connect()`, a program calls the `base.MessageServer.send()` class method to notify other programs registered to the group.

```plaintext
CALL base.MessageServer.send("f1")
```

All programs registered to the message server group are notified, including the program which has sent the message. The messages can be treated by the current dialog with a simple `ON KEY()` interaction block.

**Examples**

`base.MessageServer` usage examples.

*Example 1: Simple MessageServer usage*

```plaintext
MAIN
    CALL base.MessageServer.connect()
    MENU "test"
        COMMAND "Send F1"
        CALL base.MessageServer.send("f1")
        COMMAND KEY (F1)
        ERROR SFMT("Key F1 received at %1",CURRENT HOUR TO SECOND)
    COMMAND "quit"
    EXIT MENU
END MENU
END MAIN
```

**The ui package**

These topics cover the built-in classes of the `ui` class

**The Interface class**

The `ui.Interface` class provides methods to manipulate the user interface.

This class does not have to be instantiated.
ui.Interface methods
Methods of the ui.Interface class
# Table 522: Class methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ui.Interface.frontCall</code> (moduleName STRING, functionName STRING, [ valueList ], [ variableList ])</td>
<td><code>ui.Interface.frontCall</code> performs a function call to the current front-end.</td>
</tr>
<tr>
<td><code>ui.Interface.filenameToURI</code> (path STRING) RETURNS STRING</td>
<td>Converts a file name to an URI to be used as a web component image resource.</td>
</tr>
<tr>
<td><code>ui.Interface.getChildCount</code> () RETURNS INTEGER</td>
<td>Get the number of children in a parent container.</td>
</tr>
<tr>
<td><code>ui.Interface.getChildInstances</code> (name STRING) RETURNS INTEGER</td>
<td>Get the number of child instances for a given program name.</td>
</tr>
<tr>
<td><code>ui.Interface.getContainer</code> () RETURNS STRING</td>
<td>Get the parent container of the current program.</td>
</tr>
<tr>
<td><code>ui.Interface.getFrontEndName</code> () RETURNS STRING</td>
<td>Returns the type of the front-end currently in use.</td>
</tr>
<tr>
<td><code>ui.Interface.getFrontEndVersion</code> () RETURNS STRING</td>
<td>Returns the version of the front-end currently in use.</td>
</tr>
<tr>
<td><code>ui.Interface.getName</code> () RETURNS STRING</td>
<td><code>ui.Interface.frontCall</code> performs a function call to the current front-end.</td>
</tr>
<tr>
<td><code>ui.Interface.getImage</code> () RETURNS STRING</td>
<td><code>ui.Interface.frontCall</code> performs a function call to the current front-end.</td>
</tr>
<tr>
<td><code>ui.Interface.getText</code> () RETURNS STRING</td>
<td>Returns the title of the program.</td>
</tr>
<tr>
<td><code>ui.Interface.getType</code> () RETURNS STRING</td>
<td>Returns the type of the program.</td>
</tr>
<tr>
<td><code>ui.Interface.getRootNode</code> () RETURNS <code>om.DomNode</code></td>
<td>Get the root DOM node of the abstract user interface.</td>
</tr>
<tr>
<td><code>ui.Interface.getUniversalClientName</code> () RETURNS STRING</td>
<td>Returns the name of the front-end used for Universal Rendering.</td>
</tr>
<tr>
<td><code>ui.Interface.getUniversalClientVersion</code> () RETURNS STRING</td>
<td>Returns the version of the front-end used for Universal Rendering.</td>
</tr>
</tbody>
</table>
ui.Interface.frontCall

ui.Interface.frontCall performs a function call to the current front-end.

Syntax

```plaintext
ui.Interface.frontCall(
    moduleName STRING,
    functionName STRING,
    [ valueList ],
    [ variableList ]
)
```

1. **moduleName** defines the shared library or classpath where the function is implemented.
2. **functionName** defines the name of the function to be called.
3. **valueList** is a list of input parameters.
4. **variableList** is a list of output parameters.

**Important:** The variables for output parameters are passed by reference to the `frontCall()` method.

Usage

The `ui.Interface.frontCall()` class method can be used to execute a procedure on the front-end workstation through the front-end software component. You can for example launch a front-end specific application like a browser or a text editor, or manage the clipboard content.

**Important:** When calling the `ui.Interface.frontCall()` method, the connection to the front-end is initiated, if it is not yet established. Consider avoiding front calls in batch programs and interactive programs using the text mode. This is also important to consider in graphical mode, if no interactive instruction was issued before the front call. Furthermore, each front call will sync the AUI tree with the front end.

The method takes four parameters:

1. The module, identifying the shared library (.so or .DLL) or the Java class (GMA) implementing the front call function.
2. The function of the module to be executed.
3. The list of input parameters, using the square brace notation.
4. The list of output parameters, using the square brace notation.

Input and output parameters are provided as a **variable list of parameters**, by using the square brackets notation ([param1,param2,...]):

- Input and output parameters can be of any simple type like INTEGER, a RECORD or a DYNAMIC ARRAY.
- An empty list of input or output parameters is specified with [].
- Input parameters can be an expression such as \((10 * \text{var})\).
- Output parameters must be variables only, to receive the returning values.
- Output parameters are optional. If the front call returns values, these values will be ignored by the runtime system, if no output parameters are provided to receive these values.

Simple front call example:

```plaintext
FUNCTION call()
    DEFINE info STRING
    CALL ui.Interface.frontCall( "standard", "feInfo", ["feName"], [info] )
END FUNCTION
```

Some front calls need a file path as parameter. File paths must follow the syntax of the front end workstation file system. You may need to escape backslash characters in such parameters. The following example shows how to pass a file path with a space in a directory name to a front-end running on a Microsoft™ Windows® workstation:

```plaintext
FUNCTION call()
    DEFINE path STRING, res INTEGER
```
LET path = "\c:\work dir\my report.doc"
-- This is: "c:\work dir\my report.doc"
CALL ui.Interface.frontCall( "standard", "shellExec", [path], [res] )
END FUNCTION

When using RECORD and DYNAMIC ARRAY as front call input or output parameters, the runtime system will use JSON serialization, to pass and return such structured data to/from the front-end. This is important to know when implementing your own custom front calls. Note that one can use the json_null and json_name variable definition attributes to control JSON serialization:

DEFINE optrec RECORD
    mode INTEGER ATTRIBUTES(json_null="null"),
    filter STRING ATTRIBUTES(json_name="Data Filter")
END RECORD
DEFINE flags DYNAMIC ARRAY OF INTEGER ATTRIBUTES(json_null="undefined")
DEFINE result_list DYNAMIC ARRAY OF STRING
LET optrec.mode = 999
LET optrec.filter = "*A*"
LET flags[1] = 111
LET flags[3] = 333
CALL ui.Interface.frontCall( "m1", "fc1", [optrec, flags], [result_list] )

Front call cost
A front call is a remote procedure call requiring a full network round trip between the server app and the front-end. Depending on the current network speed, this may result in delays in the millisecond to sub second range.

Note: In mobile application development or runOnServer mode, the execution time of a front can be much slower when running the app on the server, compared to embedded apps.

Front call error handling
Exception handling instructions can be used to check the execution status of a front call. Both WHENEVER ERROR directives or TRY/CATCH blocks can surround the front call to avoid program stopping in case of error, and to check the error number returned in the STATUS variable.

Note: There is no need to surround front calls with exception handlers such as TRY/CATCH, if the front call is always supposed to execute without error. For example, the feInfo front call will never produce an exception.

Example of front call error handling with a TRY/CATCH block:

FUNCTION takePhoto()
    DEFINE path STRING
    TRY -- This front call may fail if the front-end is not a mobile device:
        CALL ui.Interface.frontCall( "mobile", "takePhoto", [], [path] )
    CATCH
        MESSAGE "Cannot take photo: ", STATUS, " ", err_get(STATUS)
        LET path = NULL
    END TRY
    RETURN path
END FUNCTION

If the front call module name or the function name is invalid, the errors -6331 or -6332 will be raised, respectively.

If the front call execution fails for some reason, the error -6333 will be raised. The description of the problem can be found in the second part of the error message, returned by a call to the ERR_GET() function.

The error -6334 can be raised in case of input or output parameter mismatch. The control of the number of input and output parameters is in the hands of the front-end. Most of the standard front calls have optional returning parameters and will not raise error -6334, if the output parameter list is left empty. However, front-end specific extensions or user-defined front-end functions may return an invalid execution status in case of input or output parameter mismatch,
raising error -6334. If the front-end sends a call execution status of zero (OK), and the number of returned values does not match the number of program variables, the runtime system will set unmatched program variables to NULL. As a general rule, it is recommended that the program provides the expected input and output parameters as specified in the documentation.

Related concepts

Front calls on page 583

Front call functions execute on the platform where the front-end is installed.

ui.Interface.filenameToURI

Converts a file name to an URI to be used as a web component image resource.

Syntax

```plaintext
ui.Interface.filenameToURI(
    path STRING
) RETURNS STRING
```

1. path is the local file name to be converted to a URI.

Usage

The `ui.Interface.filenameToURI()` class method converts a local (VM context / server) file name to an URI that can be accessed by the front-ends to get the resource.

This method is typically used to provide application image files in Web Components. It can also be used to provide other resource or media files to the front-end, for specific usage. For example, to display a PDF file with GBC using the "launchUrl" front call. Another use case is to play an audio file with the "playSound" front call.

Note: The runtime system uses the same mechanism to provide the front-end with images referenced in form elements. Thus, there is no need to call this method except when using application images in web components.

The `ui.Interface.filenameToURI()` method can be used when executing applications behind a GAS, but it can also be used with direct connection to the (GDC) front-end without using the GAS, or when running apps on a mobile device.

The VM context file name to URI mapping is done as follows:

- If the path parameter is already an URI (it has a scheme like http:, https:, file:), the file name is returned as is.
- If the path parameter is an absolute, relative file path, or a simple file name:
  - When the program is executing behind a GAS, user agents can access files via HTTP. In this architecture, the method will produce an URI that can be referenced in HTML elements of a web component: The image resource will be available from this location.
  - When using a direct connection to the (GDC) front-end without using the GAS, the method returns the file name as is, and the image resources will be transmitted to the GDC through the FGLIMAGEPATH mechanism.
  - When executing an app on a mobile device, both front-end and runtime system coexist on the same platform and can access the file on the system. In this architecture, the method builds the complete local path to the file from the list of directories defined in the FGLIMAGEPATH environment variable.

Note: The URI or file path returned by the `filenameToURI()` method is only valid during the program's lifetime. Do not store URIs returned by `filenameToURI()` in a persistent way.

For more details, see Providing the image resource on page 1149 and Using image resources with the gICAPI web component on page 1952
DEFINE uri STRING
    LET uri = ui.Interface.filenameToURI("myimage.png")
    CALL ui.Interface.frontCall("standard", "launchURL", [uri], [])
END MAIN

Related concepts

Web components on page 1922
This section describes how to use web components in your application.

ui.Interface.getChildCount
Get the number of children in a parent container.

Syntax

ui.Interface.getChildCount()
RETURNS INTEGER

Usage

Important: The Window Container Interface is a desktop application feature (for GDC), and is not supported on other front-ends (web and mobile). This feature is also denied when using Universal Rendering in GDC.

The ui.Interface.getChildCount() class method returns the number of child programs attached to the current parent WCI program.

WCI child programs are attached to a given container by using the ui.Interface.setContainer() method. Container and child program identifiers/names are defined by the ui.Interface.setName() method.

Related concepts

Window containers (WCI) on page 2023
WCI containers define window containers to group several programs in a parent multiple document interface presentation.

ui.Interface.getChildInstances
Get the number of child instances for a given program name.

Syntax

ui.Interface.getChildInstances(name STRING)
RETURNS INTEGER

1. name is the name of a child program attached to the container of the current program.

Usage

Important: The Window Container Interface is a desktop application feature (for GDC), and is not supported on other front-ends (web and mobile). This feature is also denied when using Universal Rendering in GDC.

The ui.Interface.getChildInstances() class method returns the number of child instances of a program attached to the current parent WCI program, based on the name of the child program passed as parameter.

The name of a child program is defined by the ui.Interface.setName() method.

The getChildInstances() method is typically used to check if a given child program is already started, to avoid multiple instances of the same program in a WCI container.

Related concepts

Window containers (WCI) on page 2023
WCI containers define window containers to group several programs in a parent multiple document interface presentation.

`ui.Interface.getContainer`
Get the parent container of the current program.

**Important:** The Window Container Interface is a desktop application feature (for GDC), and is not supported on other front-ends (web and mobile). This feature is also denied when using Universal Rendering in GDC.

**Syntax**

```javascript
ui.Interface.getContainer ()
RETURNS STRING
```

**Usage**
The `ui.Interface.getContainer()` class method returns the name of the parent WCI container defined with `ui.Interface.setContainer()`.

**Related concepts**
- `ui.Interface.setContainer` on page 2485
  Define the parent container for the current program.

- `ui.Interface.getDocument`
  Returns the DOM document of the abstract user interface tree.

**Syntax**

```javascript
ui.Interface.getDocument ()
RETURNS om.DomDocument
```

**Usage**
The `ui.Interface.getDocument()` method returns the DOM document of the abstract user interface tree.

Define a variable with the type `om.DomDocument` to receive the result of this method.

Consider using the `getRootNode()` method instead of getting the root DOM node of the AUI tree directly.

**Related concepts**
- The `DomDocument class` on page 2577
  The `om.DomDocument` class provides methods to manipulate a data tree, following the DOM standards.
- User interface basics on page 1105
  This section introduces to the foundation of the Genero user interface.
- `ui.Interface.getRootNode` on page 2479
  Get the root DOM node of the abstract user interface.

- `ui.Interface.getFrontEndName`
  Returns the type of the front-end currently in use.

**Syntax**

```javascript
ui.Interface.getFrontEndName ()
RETURNS STRING
```
Usage
The `ui.Interface.getFrontEndName()` class method returns the type of the front-end used by the program.

Table 523: Front-end names

<table>
<thead>
<tr>
<th>Front-end name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GDC</td>
<td>Desktop front-end</td>
</tr>
<tr>
<td>GMA</td>
<td>Mobile front-end for Android™</td>
</tr>
<tr>
<td>GMI</td>
<td>Mobile front-end for iOS</td>
</tr>
<tr>
<td>GWC</td>
<td>Web browser front-end</td>
</tr>
<tr>
<td>GBC</td>
<td>Genero Browser Client front-end</td>
</tr>
<tr>
<td>Console</td>
<td>Text front-end (dumb terminal)</td>
</tr>
</tbody>
</table>

**Important:** When calling this method, the user interface module of the runtime system will initialize. As result, in text mode, the terminal will be initialized and get some escape sequences. This may corrupt standard output when executing batch programs. Consider testing the FGLGUI environment variable if you want to check that the batch program is executing in text mode, instead of using the `getFrontEndName()` method.

Related concepts

Example 1: Get the type and version of the front-end on page 2488

**ui.Interface.getFrontEndVersion** on page 2478

Returns the version of the front-end currently in use.

**ui.Interface.getUniversalClientName** on page 2480

Returns the name of the front-end used for Universal Rendering.

**ui.Interface.getFrontEndVersion**

Returns the version of the front-end currently in use.

Syntax

```java
ui.Interface.getFrontEndVersion()
RETURNS STRING
```

Usage
The `ui.Interface.getFrontEndVersion()` class method returns the version number of the front-end used by the program.

**Note:** This method is primarily used for debugging purposes.

**Important:** When calling this method, the user interface module of the runtime system will initialize. As result, in text mode, the terminal will be initialized and get some escape sequences. This may corrupt standard output when executing batch programs. Consider testing the FGLGUI environment variable if you want to check that the batch program is executing in text mode, instead of using the `getFrontEndName()` method.

Related concepts

Example 1: Get the type and version of the front-end on page 2488

**ui.Interface.getFrontEndName** on page 2477
Returns the type of the front-end currently in use.

ui.Interface.getImage
Returns the icon image of the program.

Syntax

```plaintext
ui.Interface.getImage()
RETURNS STRING
```

Usage

Use the `ui.Interface.getImage()` class method to get the icon image name of the program previously set by `setImage()`.

Related concepts

- `ui.Interface.setImage` on page 2485
  Defines the icon image of the program.

- `ui.Interface.getName`
  Returns the name of the program.

Syntax

```plaintext
ui.Interface.getName()
RETURNS STRING
```

Usage

The `ui.Interface.getName()` class method returns the name of the program that was defined with the `setName()` method.

Related concepts

- `ui.Interface.setName` on page 2486
  Define the name of the current program for the front-end.

- `ui.Interface.getRootNode`
  Get the root DOM node of the abstract user interface.

Syntax

```plaintext
ui.Interface.getRootNode()
RETURNS om.DomNode
```

Usage

The `ui.Interface.getRootNode()` method returns the root DOM node of the abstract user interface tree. Define a variable of the type `om.DomNode` to receive the result of this method.

```plaintext
DEFINE rn om.DomNode
LET rn = ui.Interface.getRootNode()
-- use d to inspect/change the AUI tree
```

Example

```plaintext
MAIN
  DEFINE rn om.DomNode
```
Related concepts

The DomDocument class on page 2577
The om.DomDocument class provides methods to manipulate a data tree, following the DOM standards.

User interface basics on page 1105
This section introduces to the foundation of the Genero user interface.

ui.Interface.getText
Returns the title of the program.

Syntax

```
ui.Interface.getText()
RETURNS STRING
```

Usage

Use the `ui.Interface.getText()` class method to get the title of the program previously set by `setText()`.

Related concepts

ui.Interface.setText on page 2487
Defines the title for the program.

ui.Interface.getType
Returns the type of the program.

Syntax

```
ui.Interface.getType()
RETURNS STRING
```

Usage

Use the `ui.Interface.getType()` class method to get the type of the program previously set by `setType()`.

Related concepts

ui.Interface.setType on page 2487
Defines the type of the program for the front-end.

ui.Interface.getUniversalClientName
Returns the name of the front-end used for Universal Rendering.

Syntax

```
ui.Interface.getUniversalClientName()
RETURNS STRING
```

Usage

The `ui.Interface.getUniversalClientName()` class method returns the name of the front-end used to achieve Universal Rendering.
Note: The `getUniversalClientName()` method is especially useful for debugging purposes. Consider writing code that is not dependent on the Native Rendering or Universal Rendering mode.

When Universal Rendering mode is used, this method returns "GBC".

Important: When calling this method, the user interface module of the runtime system will initialize. As a result, in text mode, the terminal will be initialized and get some escape sequences. This may corrupt standard output when executing batch programs. Consider testing the FGLGUI environment variable, if you want to check that the batch program is executing in text mode.

Related concepts

`ui.Interface.getUniversalClientVersion` on page 2481
Returns the version of the front-end used for Universal Rendering.

`ui.Interface.getFrontEndName` on page 2477
Returns the type of the front-end currently in use.

`ui.Interface.getUniversalClientVersion` Returns the version of the front-end used for Universal Rendering.

Syntax

```java
ui.Interface.getUniversalClientVersion()
RETURNS STRING
```

Usage

The `ui.Interface.getUniversalClientVersion()` class method returns the version number of the front-end used to achieve Universal Rendering.

Note: This method is primarily used for debugging purposes.

Important: When calling this method, the user interface module of the runtime system will initialize. As a result, in text mode, the terminal will be initialized and get some escape sequences. This may corrupt standard output when executing batch programs. Consider testing the FGLGUI environment variable, if you want to check that the batch program is executing in text mode.

Related concepts

`ui.Interface.getUniversalClientName` on page 2480
Returns the name of the front-end used for Universal Rendering.

`ui.Interface.loadActionDefaults` Load the default action defaults file.

Syntax

```java
ui.Interface.loadActionDefaults (filename STRING)
```

1. `filename` is the name of the action defaults file, without the extension.

Usage

Use the `ui.Interface.loadActionDefaults()` class method to load a .4ad file defining action defaults for all program forms.

If the interface already contains action defaults, these will be replaced by the new action defaults loaded by this method.

Specify the action defaults file name without the ".4ad" extension.
The resource file is searched in several directories in a given order, as described in the FGLRESOURCEPATH reference topic.

**Example**

```
CALL ui.Interface.loadActionDefaults("mydefaults")
```

For a complete example, see Example 4: Loading custom resources on page 2489.

**Related concepts**

FGLRESOURCEPATH on page 279
Defines search path for resource files.

Configuring actions on page 1744
Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with action attributes.

**ui.Interface.loadStartMenu**

Load the start menu file.

**Syntax**

```
ui.Interface.loadStartMenu(
    filename STRING )
```

1. *filename* is the name of a start menu file, without the extension.

**Usage**

Use the `ui.Interface.loadStartMenu()` class method to load a .4sm file defining a start menu.

If the interface already contains a start menu, it will be replaced by the new start menu loaded by this method.

Specify the start menu file name without the ".4sm" extension.

The resource file is searched in several directories in a given order, as described in the FGLRESOURCEPATH reference topic.

**Example**

```
CALL ui.Interface.loadStartMenu("mystartmenu")
```

**Related concepts**

FGLRESOURCEPATH on page 279
Defines search path for resource files.

Start menus on page 2019
Start menus define a tree of application programs that can be started.

**ui.Interface.loadStyles**

Load the presentation styles file.

**Syntax**

```
ui.Interface.loadStyles(
    filename STRING )
```

1. *filename* is the name of presentation styles file, without the extension.
Usage

Use the `ui.Interface.loadStyles()` class method to load a `.4st` file defining presentation styles for all program forms.

If the interface already contains a set of presentation styles, it will be replaced by the new presentation styles loaded by this method.

Specify the presentation styles file name without the ".4st" extension.

The resource file is searched in several directories in a given order, as described in the `FGLRESOURCEPATH` reference topic.

Example

```
CALL ui.Interface.loadStyles("mystyles")
```

For a complete example, see `Example 4: Loading custom resources` on page 2489.

Related concepts

[FGLRESOURCEPATH](#) on page 279
Defines search path for resource files.

[Presentation styles](#) on page 1165
Use presentation styles to specify decoration attributes for window and form elements.

`ui.Interface.loadToolBar`
Load a default/global toolbar file for all forms of the program.

Syntax

```
ui.Interface.loadToolBar(
    filename STRING )
```

1. `filename` is the name of a toolbar file, without the extension.

Usage

Use the `ui.Interface.loadToolBar()` class method to load a `.4tb` file defining a default/global toolbar for all forms.

The purpose of the default/global toolbar is to be displayed in all forms. However, the default/global toolbar may be displayed in the current window or in the parent window, depending on the front-end type and window type (modal or normal).

If the interface already contains a default/global toolbar, it will be replaced by the new toolbar loaded by this method.

Specify the toolbar file name without the ".4tb" extension.

The resource file is searched in several directories in a given order, as described in the `FGLRESOURCEPATH` reference topic.

The default/global toolbar loaded by this method is also used for the WCI container.

Example

```
CALL ui.Interface.loadToolBar("mytoolbar")
```

Related concepts

[FGLRESOURCEPATH](#) on page 279
Defines search path for resource files.

[Tabs](#) on page 1436
Toolbars define a bar of buttons that appears at the top of application forms.

**Window containers (WCI)** on page 2023
WCI containers define window containers to group several programs in a parent multiple document interface presentation.

**ui.Interface.loadTopMenu**
Load a default/global topmenu file for all forms of the program.

**Syntax**

```
ui.Interface.loadTopMenu(
    filename STRING )
```

1. `filename` is the name of a topmenu file, without the extension.

**Usage**

Use the `ui.Interface.loadTopMenu()` class method to load a .4tm file defining a default/global topmenu for all forms.

The purpose of the default/global topmenu is to be displayed in all forms. However, the default/global toolbar may be displayed in the current window or in the parent window, depending on the front-end type and window type (modal or normal).

If the interface already contains a global topmenu, it will be replaced by the new topmenu loaded by this method.

Specify the topmenu file name without the "4tm" extension.

The resource file is searched in several directories in a given order, as described in the `FGLRESOURCEPATH` reference topic.

The default/global topmenu loaded by this method is also used for the WCI container.

**Example**

```
CALL ui.Interface.loadTopMenu("mytopmenu")
```

**Related concepts**

- **Topmenus** on page 1444
  Topmenus define typical pull-down menus that appear at the top of application forms.

- **Window containers (WCI)** on page 2023
  WCI containers define window containers to group several programs in a parent multiple document interface presentation.

- **ui.Interface.refresh**
  Synchronize the user interface with the front-end.

**Syntax**

```
ui.Interface.refresh()
```

**Usage**

The `ui.Interface.refresh()` class method forces a synchronization of the abstract user interface tree with the front-end. This means that the end user will immediately see the recent form modifications made by the program.

By default, during an interactive instruction like `DIALOG`, the AUI tree is refreshed automatically, when the runtime system gets the control back after user code execution. Thus, there is no need to call the refresh method in regular code.
**Important**: The AUI tree is automatically synchronized by the runtime system, when dialog instruction gives the control back to the end user. The `ui.Interface.refresh()` method must only be used in specific cases, to refresh the display while processing. For example, to show a "Please wait" message, or to implement a progress dialog window with a PROGRESSBAR. The `ui.Interface.refresh()` method should not be called more often than once in one second.

**Example**

```plaintext
MAIN
    DEFINE i INT
    FOR i = 1 TO 10
        DISPLAY SFMT("Please wait, doing step #%1", i) AT 1,1
        CALL ui.Interface.refresh()
        SLEEP 1
    END FOR
END MAIN
```

**Related concepts**

**Refreshing the display when processing** on page 1710

This topic explains when to use the `ui.Interface.refresh()` method.

**ui.Interface.setContainer**

Define the parent container for the current program.

**Syntax**

```plaintext
ui.Interface.setContainer(
    name STRING )
```

1. `name` is the name of the parent container.

**Usage**

The `ui.Interface.setContainer(name)` class method is used to specify the name of the parent WCI container where the current program windows must be displayed. This creates a WCI relation between two independent programs running with distinct fglrun processes.

Each WCI program must be identified by a name, to be set with the `ui.Interface.setName()` class method.

**Related concepts**

**Window containers (WCI)** on page 2023

WCI containers define window containers to group several programs in a parent multiple document interface presentation.

**Example 3: Using the Window Container Interface** on page 2488

**ui.Interface.setImage**

Defines the icon image of the program.

**Syntax**

```plaintext
ui.Interface.setImage(
    image STRING )
```

1. `image` is the image file name to be used as program icon.

**Usage**

Use the `ui.Interface.setImage()` class method to define the icon image for the program to be used by the front-ends. This icon will be used in taskbars, for example.
Call the method at the beginning of the program, before any interactive instruction.

**Related concepts**

- `ui.Interface.setName` on page 2486
  Define the name of the current program for the front-end.

- `ui.Interface.setText` on page 2487
  Defines the title for the program.

- Window containers (WCI) on page 2023
  WCI containers define window containers to group several programs in a parent multiple document interface presentation.

- Example 3: Using the Window Container Interface on page 2488

### `ui.Interface.setName`

Define the name of the current program for the front-end.

**Syntax**

```java
ui.Interface.setName(
    name STRING )
```

1. `name` is the identifier of the program.

**Usage**

Use the `ui.Interface.setName()` class method to define the identifier for the program to be used by the front-ends, for example in case of window container usage.

The name passed to this method will be passed to the front-end in order to identify the program.

Call the method at the beginning of the program, before any interactive instruction.

By default, it is the program name (without `.42m` or `.42r` extension).

**Related concepts**

- `ui.Interface.setText` on page 2487
  Defines the title for the program.

- `ui.Interface.setImage` on page 2485
  Defines the icon image of the program.

- Window containers (WCI) on page 2023
  WCI containers define window containers to group several programs in a parent multiple document interface presentation.

- Example 3: Using the Window Container Interface on page 2488

### `ui.Interface.setSize`

Specify the initial size of the parent container window.

**Syntax**

```java
ui.Interface.setSize(
    h INTEGER,
    w INTEGER )
```

1. `h` is the initial height of the main window.
2. `w` is the initial width of the main window.
Usage

Use the `ui.Interface.setSize()` class method to define the initial size of the parent container window of a window container application. The parameters can be integer or string values.

By default, the unit is the character grid cells, but you can add the `px` unit to specify the height and width in pixels.

The `setSize()` method can also be used to configure the size of the main window when using traditional mode, as a replacement for the `fgl_setsize()` built-in function.

Call the method at the beginning of the program, before any interactive instruction.

Related concepts

Window containers (WCI) on page 2023
WCI containers define window containers to group several programs in a parent multiple document interface presentation.

Example 3: Using the Window Container Interface on page 2488

`ui.Interface.setText`
Defines the title for the program.

Syntax

```plaintext
ui.Interface.setText (title STRING )
```

1. `title` is the text to be used as program title.

Usage

Use the `ui.Interface.setText()` class method to define the title for the program to be used by the front-ends, for example in case of window container usage (as title for the main window), or for the text to be displayed in the task bars.

Call the method at the beginning of the program, before any interactive instruction.

Related concepts

`ui.Interface.setName` on page 2486
Define the name of the current program for the front-end.

`ui.Interface.setImage` on page 2485
Defines the icon image of the program.

Window containers (WCI) on page 2023
WCI containers define window containers to group several programs in a parent multiple document interface presentation.

Example 3: Using the Window Container Interface on page 2488

`ui.Interface.setType`
Defines the type of the program for the front-end.

Syntax

```plaintext
ui.Interface.setType (type STRING )
```

1. `type` is the identifier of the program.
Usage

Use the `ui.Interface.setType()` class method to define the type for the program to be used by the front-ends, for example in case of window container usage.

Possible values are: `normal, container, child`.

The type passed to this method will be passed to the front-end in order to define the rendering and behavior of the program.

Call the method at the beginning of the program, before any interactive instruction.

Related concepts

- `ui.Interface.setText` on page 2487
  Defines the title for the program.
- `ui.Interface.setImage` on page 2485
  Defines the icon image of the program.
- `Window containers (WCI)` on page 2023
  WCI containers define window containers to group several programs in a parent multiple document interface presentation.
- `Example 3: Using the Window Container Interface` on page 2488

Examples

- `ui.Interface` usage examples.

  Example 1: Get the type and version of the front-end

  ```
  MAIN
  MENU "Test"
  COMMAND "Get"
    DISPLAY "Name = " || ui.Interface.getFrontEndName()
    DISPLAY "Version = " || ui.Interface.getFrontEndVersion()
  COMMAND "Exit"
  EXIT MENU
  END MENU
  END MAIN
  ```

  Example 2: Get the AUI root node and save it to a file in XML format

  ```
  MAIN
  DEFINE n om.DomNode
  MENU "Test"
  COMMAND "SaveUI"
    LET n = ui.Interface.getRootNode()
    CALL n.writeXml("auitree.xml")
  COMMAND "Exit"
  EXIT MENU
  END MENU
  END MAIN
  ```

  Example 3: Using the Window Container Interface

  The WCI parent program:

  ```
  MAIN
  CALL ui.Interface.setName("main1")
  CALL ui.Interface.setText("This is the parent container")
  CALL ui.Interface.setType("container")
  CALL ui.Interface.setSize("600px","600px")
  CALL ui.Interface.loadStartMenu("appmenu")
  MENU "Main"
    COMMAND "Help" CALL help()
  ```
COMMAND "About" CALL aboutbox()
COMMAND "Exit"
  IF ui.Interface.getChildCount()>0 THEN
    ERROR "You must first exit the child programs."
  ELSE
    EXIT MENU
  END IF
END MENU
END MAIN

The WCI child program:

MAIN
  CALL ui.Interface.setName("progl")
  CALL ui.Interface.setText("This is module 1")
  CALL ui.Interface.setType("child")
  CALL ui.Interface.setContainer("main1")
  MENU "Test"
    COMMAND "Exit"
  EXIT MENU
END MENU
END MAIN

**Example 4: Loading custom resources**

Form file (customer.per):

```
LAYOUT
GRID
{
  Num: [f1       ]
  Name: [f2                 ]
}
END
END
ATTRIBUTES
EDIT f1 = FORMONLY.cust_id;
EDIT f2 = FORMONLY.cust_name, STYLE="mandatory";
END
```

Program file:

```
MAIN
  DEFINE rec RECORD
    cust_id INT,
    cust_name VARCHAR(50)
  END RECORD
  CALL ui.Interface.loadActionDefaults("myactdefs")
  CALL ui.Interface.loadStyles("mystyles")
  OPEN FORM f1 FROM "customer"
  DISPLAY FORM f1
  INPUT BY NAME rec.*
END MAIN
```

Styles file (mystyles.4st):

```
<?xml version="1.0" encoding="ANSI_X3.4-1968"?>
<StyleList>
  <Style name="Window">
    <StyleAttribute name="windowType" value="normal" />
  </Style>
  <Style name="Edit.mandatory">
```

```
Action Defaults file (myactdefs.4ad):

```xml
<?xml version="1.0" encoding="ANSI_X3.4-1968"?>
<ActionDefaultList>
  <ActionDefault name="accept" text="Accept" acceleratorName="Return" acceleratorName2="Enter" />
  <ActionDefault name="cancel" validate="no" text="Cancel" acceleratorName="Escape" />
  <ActionDefault name="dialogtouched" validate="no" defaultView="no" contextMenu="no"/>
</ActionDefaultList>
```

**The Window class**

The `ui.Window` class provides an interface to the window objects create with the `OPEN WINDOW` instruction.

A window is typically created with a form with the `OPEN WINDOW WITH FORM` instruction. If the window contains a form, consider using the `ui.Form` class instead of `ui.Window`.

**ui.Window methods**

Methods of the `ui.Window` class.

**Table 524: Class methods**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ui.Window.forName(name STRING)</code></td>
<td>RETURNS <code>ui.Window</code></td>
</tr>
<tr>
<td><code>ui.Window.getCurrent()</code></td>
<td>RETURNS <code>ui.Window</code></td>
</tr>
<tr>
<td></td>
<td>Get a window object by name.</td>
</tr>
<tr>
<td></td>
<td>Get the current window object.</td>
</tr>
</tbody>
</table>
### Table 525: Object methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>createForm</code> (</td>
<td>Create a new empty form in a window.</td>
</tr>
<tr>
<td><code>name STRING</code> )</td>
<td>RETURNS <code>ui.Form</code></td>
</tr>
<tr>
<td><code>findNode</code> (</td>
<td>Search for a specific element in the window.</td>
</tr>
<tr>
<td><code>type STRING,</code></td>
<td>RETURNS <code>om.DomNode</code></td>
</tr>
<tr>
<td><code>name STRING</code> )</td>
<td>Get the current form of a window.</td>
</tr>
<tr>
<td><code>getForm()</code></td>
<td>RETURNS <code>ui.Form</code></td>
</tr>
<tr>
<td><code>getNode()</code></td>
<td>Get the DOM node of a window.</td>
</tr>
<tr>
<td><code>getForm()</code></td>
<td>RETURNS <code>om.DomNode</code></td>
</tr>
<tr>
<td><code>getText()</code></td>
<td>Get the window title.</td>
</tr>
<tr>
<td><code>getImage()</code></td>
<td>RETURNS <code>STRING</code></td>
</tr>
<tr>
<td><code>setImage</code> (</td>
<td>Set the window icon.</td>
</tr>
<tr>
<td><code>image STRING</code> )</td>
<td>RETURNS <code>STRING</code></td>
</tr>
<tr>
<td><code>setText</code> (</td>
<td>Set the window title.</td>
</tr>
<tr>
<td><code>text STRING</code> )</td>
<td>RETURNS <code>STRING</code></td>
</tr>
</tbody>
</table>

### Related concepts

**Defining the window title** on page 1145
Use the `TEXT` attribute to define a title for a window.

`ui.Window.forName`
Get a window object by name.

### Syntax

```plaintext
ui.Window.forName ( 
  `name STRING` ) 
RETURNS `ui.Window`
```

1. `name` defines the name of the window.

### Usage

The `ui.Window.forName()` class method returns the `ui.Window` object corresponding to an identifier used to create the window with the `OPEN WINDOW` instruction.
Declare a variable of type `ui.Window` to hold the window object reference.

The name of the window passed as parameter can use the same letter case as in the `OPEN WINDOW` instruction: The lookup is case-insensitive.

**Example**

```<code>
define w ui.Window
open window w1 with form "custform"
let w = ui.Window.forName("w1")
...
</code>```

For a complete example, see Example 1: Get a window by name and change the title on page 2496.

**Related concepts**

- Example 1: Get a window by name and change the title on page 2496
- `ui.Window.getCurrent` on page 2493
  
  Get the current window object.

`ui.Window.createForm`

Create a new empty form in a window.

**Syntax**

```<code>
createForm(    
    name STRING )
returns ui.Form
</code>```

1. `name` is the name for the form.

**Usage**

The `createForm()` method can be used to create a new empty form in the window object. This is typically used to build forms dynamically, by creating the elements with the OM API.

**Important:** It is mandatory to create a form in a window with the `createForm()` method, otherwise it is not usable.

The method returns a new `ui.Form` instance or `NULL` if the form name passed as the parameter identifies an existing form used by the window.

**Example**

```<code>
define w ui.Window, 
    f ui.Form, 
    n, g om.DomNode
open window w1 with 10 rows, 20 columns
let w = ui.Window.getCurrent()
let f = w.createForm("myform")
let n = f.getNode()
let g = n.createChild("Grid")
...
</code>```

**Related concepts**

- Create forms dynamically on page 2004
  
  Dynamic dialogs are typically used with forms that are generated at runtime.

- The `DomNode` class on page 2584
The `DomNode` class provides methods to manipulate a DOM node of a data tree.

**ui.Window.getCurrent**
Get the current window object.

**Syntax**

```java
ui.Window.getCurrent()  
RETURNS ui.Window
```

**Usage**

The `ui.Window.getCurrent()` class method returns the `ui.Window` object corresponding to the current window.

Declare a variable of type `ui.Window` to hold the window object reference.

**Example**

```java
DEFINE w ui.Window
OPEN WINDOW w1 WITH FORM "custform"
LET w = ui.Window.getCurrent()
...
```

For a complete example, see Example 2: Get a the current form and hide a groupbox on page 2496.

**Related concepts**

Example 2: Get a the current form and hide a groupbox on page 2496
`ui.Window.forName` on page 2491
Get a window object by name.

**ui.Window.getForm**
Get the current form of a window.

**Syntax**

```java
getForm()  
RETURNS ui.Form
```

**Usage**

The `getForm()` method returns the `ui.Form` object corresponding to the current form used by the window object.

Declare a variable of type `ui.Form` to hold the form object reference.

Consider using the `ui.Dialog.getForm()` method to get the form object reference.

**Example**

```java
DEFINE f ui.Form
OPEN WINDOW w1 WITH FORM "custform"
LET w = ui.Window.getCurrent()
LET f = w.getForm()
...
```

For a complete example, see Example 2: Get a the current form and hide a groupbox on page 2496.

**Related concepts**
The Form class on page 2496
The `ui.Form` class provides an interface to form objects created by an `OPEN WINDOW WITH FORM` or `DISPLAY FORM` instruction.

`ui.Dialog.getForm()` on page 2534
Returns the current form used by the dialog.

`ui.Window.getNode`
Get the DOM node of a window.

**Syntax**

```plaintext
getNode()  
RETURNS om.DomNode
```

**Usage**

The `getNode()` method returns the `om.DomNode` object corresponding to the window object.

Declare a variable of type `om.DomNode` to hold the DOM node object reference.

Consider using the `ui.Dialog.getForm()` method to get the form used by the current dialog.

**Example**

```plaintext
DEFINE w ui.Window, n om.DomNode  
OPEN WINDOW w1 WITH FORM "custform"  
LET w = ui.Window.getCurrent()  
LET n = w.getNode()  
...  
```

**Related concepts**

- The `DomNode class` on page 2584
  The `om.DomNode` class provides methods to manipulate a DOM node of a data tree.

- `ui.Dialog.getForm` on page 2534
  Returns the current form used by the dialog.

- `ui.Window.findNode`
  Search for a specific element in the window.

**Syntax**

```plaintext
findNode(  
    type STRING,  
    name STRING  
)  
RETURNS om.DomNode
```

1. `type` defines the type of the node.
2. `name` defines the name of the node.

**Usage**

The `findNode()` method allows you to search for a specific DOM node in the abstract representation of the window. You search for a child node by giving its type and the name of the element (the tagname and the value of the 'name' attribute).

The method returns the first element found matching the specified type (tagname) and node name. Window element names must be unique for the same type of nodes, if you want to distinguish all elements.

The `findNode()` method is provided for `ui.Window` class for specific cases when the window does not contain a form. For windows containing a form, use the `ui.Form.findNode()` method instead.
Example

```
DEFINE w ui.Window, n om.DomNode
OPEN WINDOW w1 WITH FORM "custform"
LET w = ui.Window.getCurrent()
LET n = w.findNode("FormField","customer.cust_name")
...
```

Related concepts

`ui.Form.findNode` on page 2503
Search for a child node in the form.

`ui.Window.getImage`
Get the window icon.

**Syntax**

```
getImage()
RETURNS STRING
```

**Usage**

Use the `getImage()` method to get the current icon of a window.

`ui.Window.getText`
Get the window title.

**Syntax**

```
getText()
RETURNS STRING
```

**Usage**

Use the `getText()` method to get the current title of a window.

`ui.Window.setImage`
Set the window icon.

**Syntax**

```
setImage(
    image STRING
)
```

1. `image` is the image name for the icon of the window.

**Usage**

The `setImage()` method specifies the icon of the window.

By default, the icon of a window is defined by the `IMAGE` attribute of the `LAYOUT` definition in form files.

`ui.Window.setText`
Set the window title.

**Syntax**

```
setText()
```
Usage

The `setText()` method defines the title of the window.

By default, the title of a window is defined by the `TEXT` attribute of the `LAYOUT` definition in form files.

Related concepts

Example 1: Get a window by name and change the title on page 2496

Windows and forms on page 1131

The section describes the concept of windows and forms in the language.

Examples

`ui.Window` usage examples.

Example 1: Get a window by name and change the title

```plaintext
MAIN
DEFINE w ui.Window
OPEN WINDOW w1 WITH FORM "customer" ATTRIBUTES(TEXT="Unknown")
LET w = ui.Window.forName("w1")
IF w IS NULL THEN
  EXIT PROGRAM
END IF
CALL w.setText("Customer")
MENU "Test"
  COMMAND "exit" EXIT MENU
END MENU
CLOSE WINDOW w1
END MAIN
```

Example 2: Get the current form and hide a groupbox

```plaintext
MAIN
DEFINE w ui.Window
DEFINE f ui.Form
OPEN WINDOW w1 WITH FORM "customer"
LET w = ui.Window.getCurrent()
IF w IS NULL THEN
  EXIT PROGRAM
END IF
LET f = w.getForm()
MENU "Test"
  COMMAND "hide" CALL f.setElementHidden("gb1",1)
  COMMAND "exit" EXIT MENU
END MENU
CLOSE WINDOW w1
END MAIN
```

The Form class

The `ui.Form` class provides an interface to form objects created by an `OPEN WINDOW WITH FORM` or `DISPLAY FORM` instruction.

A form object allows you to manipulate form elements by program. For example, you can hide parts of a form with the `setElementHidden()` method. The runtime system is able to handle hidden fields during a dialog instruction. You can, for example, hide a `GROUP` containing fields and labels.

Outside dialogs, get a `ui.Form` instance of the current form with the `ui.Window.getForm()` method. When executing a dialog, use the `ui.Dialog.getForm()` method.
Note that the OPEN FORM instruction does not load a form; it simply declares a handle. The form will be created in the AUI tree when executing the DISPLAY FORM instruction. Therefore, the corresponding ui.Form object is only available after DISPLAY FORM is executed.

**ui.Form methods**
Methods of the ui.Form class.

**Table 526: Class methods**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ui.Form.setDefaultInitializer( initializer STRING )</code></td>
<td>Define the default initializer for all forms.</td>
</tr>
<tr>
<td><code>ui.Form.displayTo( value { base-type { RECORD } }, formFieldName STRING, screenLine INTEGER, attributes STRING )</code></td>
<td>Displays values to form fields or screen arrays.</td>
</tr>
</tbody>
</table>
### Table 527: Object methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ensureElementVisible</code> (name STRING)</td>
<td>Ensure the visibility of a form element.</td>
</tr>
<tr>
<td><code>ensureFieldVisible</code> (name STRING)</td>
<td>Ensure visibility of a form field.</td>
</tr>
<tr>
<td><code>findNode</code> (tagName STRING, name STRING)</td>
<td>RETURNS om.DomNode</td>
</tr>
<tr>
<td><code>getNode</code> ()</td>
<td>RETUNS om.DomNode</td>
</tr>
<tr>
<td><code>loadActionDefaults</code> (path STRING)</td>
<td>Load form action defaults.</td>
</tr>
<tr>
<td><code>loadToolBar</code> (path STRING)</td>
<td>Load the form toolbar.</td>
</tr>
<tr>
<td><code>loadTopMenu</code> (path STRING)</td>
<td>Load the form topmenu.</td>
</tr>
<tr>
<td><code>setElementComment</code> (name STRING, comment STRING)</td>
<td>Set the comment/hint of form elements.</td>
</tr>
<tr>
<td><code>setElementHidden</code> (name STRING, hidden INTEGER)</td>
<td>Show or hide form elements.</td>
</tr>
<tr>
<td><code>setElementImage</code> (name STRING, image STRING)</td>
<td>Change the image of form elements.</td>
</tr>
<tr>
<td><code>setElementStyle</code> (name STRING, style STRING)</td>
<td>Change the style of form elements.</td>
</tr>
<tr>
<td><code>setElementText</code> (name STRING, text STRING)</td>
<td>Change the text of form elements.</td>
</tr>
<tr>
<td><code>setFieldComment</code> (name STRING, comment STRING)</td>
<td>Set the comment/hint of a form field.</td>
</tr>
<tr>
<td><code>setFieldHidden</code> (name STRING, hide INTEGER)</td>
<td>Show or hide a form field.</td>
</tr>
</tbody>
</table>
**ui.Form.setDefaultInitializer**
Define the default initializer for all forms.

**Syntax**

```plaintext
ui.Form.setDefaultInitializer(
    initializer STRING )
```

1. `initializer` is the name of a function in the program.

**Usage**

Specify a default initialization function with the `ui.Form.setDefaultInitializer()` method, to implement global processing when a form is opened.

Use this method to define a global/default initialization function for all forms of the program.

The method takes the name of the initialization function as a parameter.

**Important:** The initialization function name is case insensitive.

The initialization function is called with the `ui.Form` object as the parameter.

When a form is loaded with `OPEN FORM / DISPLAY FORM` or with `OPEN WINDOW ... WITH FORM`, the initialization function is called with a `ui.Form` object as a parameter.

**Example**

The form file `form1.per`:

```plaintext
LAYOUT
GRID
{
    [f1          ]
}
END
END
ATTRIBUTES
EDIT f1 = FORMONLY.cust_name;
END
```

The main module:

```plaintext
IMPORT FGL setup
MAIN
    DEFINE cust_name STRING
    CALL setup.init_form_setup(FALSE)
    CALL ui.Form.setDefaultInitializer("form_init")
    OPEN FORM f1 FROM "form1"
    DISPLAY FORM f1 -- initialization function is called
    INPUT BY NAME cust_name
END MAIN
```

The imported module `setup.4gl`:

```plaintext
PRIVATE DEFINE with_toolbar BOOLEAN

PUBLIC FUNCTION init_form_setup(tb)
    DEFINE tb BOOLEAN
    LET with_toolbar = tb
END FUNCTION
```
PUBLIC FUNCTION form_init(f) 
  DEFINE f ui.Form 
  IF with_toolbar THEN 
    CALL f.loadToolBar("common_toolBar") 
  END IF 
END FUNCTION

Related concepts

OPEN WINDOW on page 1131
Creates and displays a new window.

OPEN FORM on page 1137
 Declares a compiled form in the program.

DISPLAY FORM on page 1138
 Displays and associates a form with the current window.

ui.Form.displayTo
 Displays values to form fields or screen arrays.

Syntax

ui.Form.displayTo(
  value ↓ base-type ↓ RECORD ↓,
  formFieldName STRING,
  screenLine INTEGER,
  attributes STRING
)

1. value is the scalar value or RECORD to be displayed, and base-type is a base data type of Genero (INTEGER, DATE, VARCHAR)
2. formFieldName is a simple form field name, like "customer.cust_name" or the name of a SCREEN RECORD defined in the current form.
3. screenLine is a line number in the screen array.
4. attributes is a space-separated list of TTY attribute names (like "RED REVERSE").

Usage

The displayTo() class method is equivalent to the DISPLAY TO instruction.

Use of this method is only recommended, if the name of the form field, the name of the screen record, and/or the display attributes are not known at compile-time. With regular (static) dialogs, the DISPLAY TO or DISPLAY BY NAME instructions should be used instead.

This method is typically used in the context of dynamic dialog programming.

The displayTo() method can be used to:

• display a simple value to a form field,
• display a complete RECORD to all fields of a SCREEN RECORD,
• display a complete RECORD to all cells of a screen array line of a SCREEN RECORD.

Important: When using a RECORD as value parameter, do not use the .* notation. However, in the second parameter of the ui.Form.displayTo() method, the screen record must be specified with a .* notation.

When displaying a complete RECORD to a screen array, you must specify the screen line as third parameter. Otherwise, for simple form fields or flat screen records, the screen line is ignored.

To display a simple value to a form field:

CALL ui.Form.displayTo("foo", "f1", NULL, NULL) 
-- is the same as:

CALL ui.Form.displayTo("foo", "f1", NULL, NULL)
DISPLAY "foo" TO f1

Assuming that in the following code examples, the rec variable is defined as a RECORD, and the "sr" name is used in the definition of the SCREEN RECORD in the current form:

To display all elements of a complete RECORD to all fields grouped in a screen record:

```
CALL ui.Form.displayTo(rec, "sr.*", NULL, NULL)
-- is the same as:
DISPLAY rec.* TO sr.*
```

To display a complete RECORD to a specific screen array line:

```
CALL ui.Form.displayTo(rec, "sr.*", 2, "REVERSE, RED")
-- is the same as:
DISPLAY rec.* TO sr[2].* ATTRIBUTES(REVERSE, RED)
```

When passing NULL for the attributes parameter, the displayTo() method will behave as DISPLAY TO without an ATTRIBUTE clause, and the default TTY attributes at dialog, form or window level will apply. If you want to display the value without any TTY attribute, specify the string "normal" in the attributes parameter:

```
CALL ui.Form.displayTo("foo", "f1", NULL, "NORMAL")
-- is the same as:
DISPLAY "foo" TO f1 ATTRIBUTES(NORMAL)
```

**Related concepts**

- **Screen records / arrays** on page 1251
  Form fields can be grouped in a screen record or screen array definition.

- **Form fields** on page 1242
  Form fields are form elements designed for data input and/or data display.

- **DISPLAY TO / BY NAME instruction** on page 1639

- **ui.Form.ensureElementVisible**
  Ensure the visibility of a form element.

**Syntax**

```
ensureElementVisible(
   name STRING
)
```

1. `name` defines the name of the form element, see Identifying elements in ui.Form methods on page 2510.

**Usage**

Use the ensureElementVisible() method to make sure that the given form element (not form field) is visible to the user. This method can, for example, be used to show a folder page by passing a field that is located in the folder page, even if the field is not used in a dialog.

This method must be used for static form elements, to make form fields visible, use the ensureFieldVisible() method instead.

The form element is identified by its name. If several form elements have the same name, the first form element found is selected.

**Note:** The ensureElementVisible() method can only show the specified element, if the focus handling in the current active dialog allows it. For more details, see the ensureFieldVisible() instead.

**Related concepts**

- **Giving the focus to a form element** on page 1731
How to force the focus to move or stay in a specific form element using program code.

**PAGE item type** on page 1282
Defines the content of a folder page.

`ui.Form.ensureFieldVisible`
Ensure visibility of a form field.

**Syntax**

```plaintext
ensureFieldVisible(
    name STRING )
```

1. `name` defines the name of the form field, see *Identifying elements in ui.Form methods* on page 2510.

**Usage**

The `ensureFieldVisible()` method makes the given form field visible to the user. This method can for example be used to show a folder page by passing a field that is located in the folder page, even if the field is not used in a dialog.

The form field is identified by name, with an optional prefix (table.column or column).

This method does not give the focus to the field passed as parameter: The folder page or screen area shown by this method call is temporarily visible and can disappear at the next user interaction, depending on focus management.

For example, consider a folder having two pages. The focus is in a field on the first page. A call to the `ensureFieldVisible()` method makes the second folder page visible, passing a field located in the second page. When the user presses the TAB key, the focus goes to the next field on the first page, bringing the first page to the top. If you want to show a folder page and give the focus to a specific field in that page, you must explicitly give the focus to a field of the page, with `NEXT FIELD`.

The `ensureFieldVisible()` method is used for form fields. To show static form elements such as labels or images, use the `ensureElementVisible()` method instead.

`ui.Form.getNode`
Get the DOM node of the form.

**Syntax**

```plaintext
getNode()
RETURNS om.DomNode
```

**Usage**

The `getNode()` method returns the DOM node containing the abstract representation of the window/form.

After loading and displaying a form with `OPEN FORM / DISPLAY FORM` or with `OPEN WINDOW ... WITH FORM`, get the form object for example with `ui.Dialog.getForm()` and use the `getNode()` method to query the DOM node corresponding to the form.

**Example**

```plaintext
MAIN
    DEFINE f ui.Form
    DEFINE n om.DomNode
    DEFINE rec RECORD
        custid INTEGER,
        custname VARCHAR(40)
    END RECORD
    OPEN FORM f1 FROM "customer"
```
Related concepts
The DomNode class on page 2584
The om.DomNode class provides methods to manipulate a DOM node of a data tree.

OPEN WINDOW on page 1131
Creates and displays a new window.

OPEN FORM on page 1137
Declares a compiled form in the program.

DISPLAY FORM on page 1138
Displays and associates a form with the current window.

ui.Form.findNode
Search for a child node in the form.

Syntax

```plaintext
findNode(
    tagName STRING,
    name STRING )
RETURNS om.DomNode
```

1. `tagName` defines the type of the node.
2. `name` defines the name of the node.

Usage
The `findNode()` method allows you to search for a specific DOM node in the abstract representation of the form. You search for a child node by giving its type and the name of the element (that is the `tagName` and the value of the 'name' attribute).

The method returns the first element found matching the specified type (`tagName`) and node name. Form element names must be unique for the same type of nodes, if you want to distinguish all elements.

Example

```plaintext
MAIN
DEFINE f ui.Form
DEFINE n, c om.DomNode
DEFINE rec RECORD
    custid INTEGER,
    custname VARCHAR(40)
END RECORD
OPEN FORM f1 FROM "customer"
DISPLAY FORM f1
INPUT BY NAME rec.*
BEFORE INPUT
    LET f = DIALOG.getForm()
    LET n = f.getNode()
    DISPLAY n.toString()
END INPUT
```
**Related concepts**

**The DomNode class** on page 2584
The DomNode class provides methods to manipulate a DOM node of a data tree.

**ui.Dialog.getForm** on page 2534
Returns the current form used by the dialog.

**The abstract user interface tree** on page 1106
The abstract user interface tree is the XML representation of the application forms displayed to the end user.

**ui.Form.loadActionDefaults**
Load form action defaults.

**Syntax**

```java
loadActionDefaults(
    path STRING)
```

1. `path` is the name of the action defaults file without extension.

**Usage**

Load form specific action defaults at runtime with the `loadActionDefaults()` method.

The `loadActionDefaults()` method is commonly used in the form initialization function.

Specify the action defaults file name without the `.4ad` extension.

The resource file is searched in several directories in a given order, as described in the `FGLRESOURCEPATH` reference topic.

If the form already contains action defaults, it will be replaced by the new action defaults loaded by this method.

The `loadActionDefaults()` method of a form object is typically used in a generic form initializer function.

**Related concepts**

`FGLRESOURCEPATH` on page 279
Defines search path for resource files.

`Configuring actions` on page 1744
Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with action attributes.

**ui.Form.loadToolBar**
Load the form toolbar.

**Syntax**

```java
loadToolBar(
    path STRING)
```

1. `path` is the name of the toolbar file without extension.

**Usage**

Load a toolbar XML definition file into the form with the `loadToolBar()` method.

The `loadToolBar()` method is commonly used in the form initialization function.

Specify the toolbar file name without the “.4tb” extension.
The resource file is searched in several directories in a given order, as described in the FGLRESOURCEPATH reference topic.

If the form already contains a toolbar (at the form level, not the global default toolbar), it will be replaced by the new toolbar loaded from this method.

Related concepts
FGLRESOURCEPATH on page 279
Defines search path for resource files.

Toolbars on page 1436
Toolbars define a bar of buttons that appears at the top of application forms.

ui.Form.loadTopMenu
Load the form topmenu.

Syntax

```plaintext
loadTopMenu ( path STRING )
```

1. `path` is the name of the topmenu file without extension.

Usage
Load a topmenu XML definition file into the form with the `loadTopMenu()` method.
The `loadTopMenu()` method is commonly used in the form initialization function.
Specify the topmenu file name without the `.4tm` extension.
The resource file is searched in several directories in a given order, as described in the FGLRESOURCEPATH reference topic.
If the form already contains a topmenu (at the form level, not the global default topmenu), it will be replaced by the new topmenu loaded by this method.

Related concepts
FGLRESOURCEPATH on page 279
Defines search path for resource files.

Topmenus on page 1444
Topmenus define typical pull-down menus that appear at the top of application forms.

ui.Form.setElementComment
Set the comment/hint of form elements.

Syntax

```plaintext
setElementComment ( name STRING, comment STRING )
```

1. `name` defines the name of the node, see Identifying elements in ui.Form methods on page 2510.
2. `comment` the text of the element comment.

Usage
Change the comment attribute of a form element with the `setElementComment()` method.
The comment is used to display a hint text, for example with a tooltip when mouse hovers a toolbar item.
Pass the identifier of the form element. The identifier is the element name as defined in the form definition.

**Note:** All elements with this name will be affected. If you want to distinguish all form elements, use unique names in the form definition file.

**Related concepts**

ui.Form.setElementImage on page 2507
Change the image of form elements.

ui.Form.setFieldComment on page 2508
Set the comment/hint of a form field.

**setElementHidden**
Show or hide form elements.

**Syntax**

```plaintext
setElementHidden(
    name STRING,
    hidden INTEGER
)
```

1. `name` defines the name of the node, see Identifying elements in ui.Form methods on page 2510.
2. `hidden` the integer value to show or hide the element.

**Usage**

Change the visibility of a form element with the `setElementHidden()` method.

**Note:** Make sure that the form layout is prepared to adapt, when elements are hidden dynamically. Containers such as TABLE allow hiding form fields / columns with little impact on the layout.

Pass the identifier of the form element. The identifier is the element name as defined in the form definition.

**Note:** All elements with this name will be affected. If you want to distinguish all form elements, use unique names in the form definition file.

The `setElementHidden()` method changes the hidden attribute of all form elements identified by the name.

The value passed to hide/show the element can be 0, 1 or 2:

**Table 528: Hidden attribute integer values**

<table>
<thead>
<tr>
<th>Hidden value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Makes the element visible.</td>
</tr>
<tr>
<td>1</td>
<td>The element is hidden and the user cannot make it visible. Typically used to hide information the user is not allowed to see.</td>
</tr>
<tr>
<td>2</td>
<td>The element is hidden and the user can make it visible.</td>
</tr>
</tbody>
</table>

**Note:** Do not hide all fields of a dialog, otherwise the dialog execution stops. At least one field must get the focus during a dialog execution.

**Related concepts**

Example 2: Hide form elements dynamically on page 2511
ui.Form.setElementImage
Change the image of form elements.

Syntax

```plaintext
setElementImage (  
    name STRING,  
    image STRING )
```

1. *name* defines the name of the node, see Identifying elements in ui.Form methods on page 2510.
2. *image* is the image to be set.

Usage

Change the image/icon of a form element with the `setElementImage()` method.

Pass the identifier of the form element. The identifier is the element name as defined in the form definition.

**Note:** All elements with this name will be affected. If you want to distinguish all form elements, use unique names in the form definition file.

Related concepts

- [ui.Form.setElementText](#) on page 2508
  Change the text of form elements.
- [ui.Form.setElementStyle](#) on page 2507
  Change the style of form elements.

ui.Form.setElementStyle
Change the style of form elements.

Syntax

```plaintext
setElementStyle (  
    name STRING,  
    style STRING )
```

1. *name* defines the name of the node, see Identifying elements in ui.Form methods on page 2510.
2. *style* is the style name to be set.

Usage

Change the style of a form element with the `setElementStyle()` method.

Pass the identifier of the form element. The identifier is the element name as defined in the form definition.

**Note:** All elements with this name will be affected. If you want to distinguish all form elements, use unique names in the form definition file.

Related concepts

- [ui.Form.setElementImage](#) on page 2507
  Change the image of form elements.
- [ui.Form.setElementText](#) on page 2508
  Change the text of form elements.
- [ui.Form.setFieldStyle](#) on page 2509
Change the style of a form field.

*ui.Form.setElementText*

Change the text of form elements.

**Syntax**

```plaintext
setElementText(
  name STRING,
  text STRING
)
```

1. *name* defines the name of the node, see Identifying elements in *ui.Form* methods on page 2510.
2. *text* is the text to be set.

**Usage**

Change the text of a form element with the `setElementText()` method, for example to modify the text of a static label or group box during program execution.

Pass the identifier of the form element. The identifier is the element name as defined in the form definition.

**Note:** All elements with this name will be affected. If you want to distinguish all form elements, use unique names in the form definition file.

**Related concepts**

- Example 3: Change the title of table column headers on page 2511
- *ui.Form.setElementImage* on page 2507
  Change the image of form elements.
- *ui.Form.setElementStyle* on page 2507
  Change the style of form elements.

*ui.Form.setFieldComment*

Set the comment/hint of a form field.

**Syntax**

```plaintext
setFieldComment(
  name STRING,
  comment STRING
)
```

1. *name* defines the name of the form field, see Identifying elements in *ui.Form* methods on page 2510.
2. *comment* the text of the element comment.

**Usage**

Change the comment attribute of a form field with the `setFieldComment()` method.

The comment attribute is used to display a message when entering the field.

**Related concepts**

- *ui.Form.setFieldStyle* on page 2509
  Change the style of a form field.
- *ui.Form.setElementComment* on page 2505
Set the comment/hint of form elements.

ui.Form.setFieldHidden
Show or hide a form field.

Syntax

```
setFieldHidden(
  name STRING,
  hide INTEGER )
```

1. `name` defines the name of the form field, see Identifying elements in ui.Form methods on page 2510.
2. `hide` the integer value to show or hide the element.

Usage

Change the visibility of a form field with the `setFieldHidden()` method.

Note: Make sure that the form layout is prepared to adapt, when elements are hidden dynamically. Containers such as `TABLE` allow hiding form fields / columns with little impact on the layout.

Pass the identifier of the form field, as defined in the form definition. The form field is identified by column name, with an optional prefix (`table.column` or `column`).

The value passed to hide/show the element can be 0, 1 or 2:

<table>
<thead>
<tr>
<th>Hidden value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Makes the field visible.</td>
</tr>
<tr>
<td>1</td>
<td>The field is hidden and the user cannot make it visible. Typically used to hide information the user is not allowed to see.</td>
</tr>
<tr>
<td>2</td>
<td>The element is hidden and the user can make it visible.</td>
</tr>
</tbody>
</table>

Note: Do not hide all fields of a dialog, otherwise the dialog execution stops. At least one field must get the focus during a dialog execution.

Related concepts

Example 2: Hide form elements dynamically on page 2511

ui.Form.setFieldStyle on page 2509
Change the style of a form field.

ui.Form.setElementStyle on page 2507
Change the style of form elements.

```
ui.Form.setFieldStyle(
  name STRING,
  style STRING )
```

1. `name` defines the name of the form field, see Identifying elements in ui.Form methods on page 2510.
2. `style` is the style name to be set.
Usage
Change the style of a form field with the `setFieldStyle()` method.

Pass the identifier of the form field, as defined in the form definition. The form field is identified by column name, with an optional prefix (`table.column` or `column`).

Related concepts
`ui.Form.setFieldHidden` on page 2509
Show or hide a form field.

`ui.Form.setElementStyle` on page 2507
Change the style of form elements.

Usage
Identifying elements in `ui.Form` methods

In `ui.Form` methods such as `setElementHidden()` and `setFieldHidden()`, the first parameter identifies an AUI node by its `name` attribute.

With `ui.Form.setElement*()` methods, the name specified as first parameter defines the AUI node(s) to be modified.

Note: When several AUI nodes share the same name, the `ui.Form.setElement*()` methods will change all matching nodes.

With `ui.Form.setField*()` methods, the name specified as first parameter is the name of a form field node, the parent node of the view node to be modified.

Note: Genero BDL distinguishes form object methods like `ui.Form.setFieldHidden()`, changing pure rendering attributes like `hidden` and `style`, from dialog object methods such as `ui.Dialog.setFieldActive()`, which are rather functional and related to dialog management.

AUI nodes are identified in XML/DOM trees by the `name` attribute and are case sensitive. When defining form elements in the form specification file, element names will be converted to lowercase letters in the `.42f` file. The name of the element passed as parameter to the `ui.Form` methods can use the same letter case as the in the form definition: The lookup is case-insensitive.

In the form file:

```
LAYOUT
... 
ATTRIBUTES
EDIT f1 = Customer.CustAddr;
... 
END
```

In the `.42f` file:

```
<formField name="customer.custaddr" ... >
  <edit ... />
</formField>
```

In the program code:

```
INPUT BY NAME r_cust.*
...
ON ACTION hide_field
  CALL DIALOG.getForm().setFieldHidden("Customer.CustAddr", 1)
...
```

Related concepts
Identifying fields in `ui.Dialog` methods on page 2556
Examples

ui.Form usage examples.

Example 1: Implement a global form initialization function

```
MAIN
   CALL ui.Form.setDefaultInitializer("init")
   OPEN FORM f1 FROM "items"
   DISPLAY FORM f1 -- Form appears in the default SCREEN window
   OPEN WINDOW w1 WITH FORM "customer"
   OPEN WINDOW w2 WITH FORM "orders"
   DISPLAY FORM f1 -- Form appears in w2 window
   MENU "Test"
      COMMAND "exit" EXIT MENU
END MENU
END MAIN

FUNCTION init(f)
   DEFINE f ui.Form
   DEFINE n om.DomNode
   CALL f.loadTopMenu("mymenu")
   LET n = f.getNode()
   DISPLAY "Init: ", n.getAttribute("name")
END FUNCTION
```

Example 2: Hide form elements dynamically

```
MAIN
   DEFINE w ui.Window
   DEFINE f ui.Form
   DEFINE rec RECORD
      custid INTEGER,
      custname VARCHAR(40)
   END RECORD
   OPEN FORM f1 FROM "customer"
   DISPLAY FORM f1
   LET w = ui.Window.getCurrent()
   LET f = w.getForm()
   INPUT BY NAME rec.*
      ON ACTION hide
         CALL f.setFieldHidden("customer.custid",1)
         CALL f.setElementHidden("label_custid",1)
      ON ACTION show
         CALL f.setFieldHidden("customer.custid",0)
         CALL f.setElementHidden("label_custid",0)
   END INPUT
END MAIN
```

Example 3: Change the title of table column headers

The form file (coltitle.per):

```
LAYOUT
GRID
{
 <TABLE t1 >
   Id   Name
   [c1  |c2   ]
   [c1  |c2   ]
   [c1  |c2   ]
   [c1  |c2   ]
   [c1  |c2   ]
}
END
```
The program file:

```
MAIN
  DEFINE f ui.Form, i INT
  DEFINE arr DYNAMIC ARRAY OF RECORD
      id INT,  
      name VARCHAR(40)
    END RECORD
  OPEN FORM f1 FROM "coltitle"
  DISPLAY FORM f1
  FOR i=1 TO 10
    LET arr[i].id = i
    LET arr[i].name = "aaa"||i
  END FOR
  DISPLAY ARRAY arr TO sr.* ATTRIBUTES(UNBUFFERED)
  BEFORE DISPLAY
      let f = dialog.getForm()
    ON ACTION change_title
        CALL f.setElementText("formonly.col1","ID")
        CALL f.setElementText("formonly.col2","NAME")
  END DISPLAY
END MAIN
```

**The Dialog class**

The `ui.Dialog` class provides a set of methods to configure, query and control the current interactive instruction.

A `ui.Dialog` object can for example be used to enable or disable actions and form fields dynamically during the dialog execution.

A dialog object is typically available inside a dialog block, with the predefined `DIALOG` keyword, and can only be referenced during the execution of that interactive instruction. After the interactive instruction, the dialog object is destroyed and its reference becomes invalid.

When using methods of the `ui.Dialog` class that alter the data model or change the current field or row, control blocks like `BEFORE FIELD`, `AFTER ROW`, `BEFORE DELETE` are not executed: These are only fired to detect and control end user activity. Program code is considered as part of the dialog implementation. For example, methods such as `ui.Dialog.deleteRow()` must not execute `BEFORE DELETE/AFTER DELETE` control blocks. These control blocks are only fired by an end-user "delete" action.

Dialog objects can also be created dynamically to handle forms created at runtime. This feature is only provided for specific needs.

**Related concepts**

Dialog instructions on page 1452
This section describes the dialog instructions to control application forms and the concepts related to dialog implementation.

The `DIALOG` control class on page 1709
This topic explains the purpose of the `ui.DIALOG` class.

Dynamic Dialogs on page 2004
Dialogs can be created at runtime with the `ui.Dialog` class.

### ui.Dialog methods
Methods of the `ui.Dialog` class.

#### Table 530: Class methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ui.Dialog.createConstructByName</code> (fields DYNAMIC ARRAY OF RECORD name STRING, type STRING END RECORD)</td>
<td>Creates an <code>ui.Dialog</code> object to implement a dynamic CONSTRUCT BY NAME.</td>
</tr>
<tr>
<td><code>ui.Dialog.createMultipleDialog</code> (</td>
<td>Creates an <code>ui.Dialog</code> object to implement a dynamic DIALOG multiple-dialog.</td>
</tr>
<tr>
<td><code>ui.Dialog.createDisplayArrayTo</code> (fields DYNAMIC ARRAY OF RECORD name STRING, type STRING END RECORD, screenRecord STRING)</td>
<td>Creates an <code>ui.Dialog</code> object to implement a dynamic DISPLAY ARRAY TO.</td>
</tr>
<tr>
<td><code>ui.Dialog.createInputArrayFrom</code> (fields DYNAMIC ARRAY OF RECORD name STRING, type STRING END RECORD, screenRecord STRING)</td>
<td>Creates an <code>ui.Dialog</code> object to implement a dynamic INPUT ARRAY FROM.</td>
</tr>
<tr>
<td><code>ui.Dialog.createInputByName</code> (fields DYNAMIC ARRAY OF RECORD name STRING, type STRING END RECORD)</td>
<td>Creates an <code>ui.Dialog</code> object to implement a dynamic INPUT BY NAME.</td>
</tr>
<tr>
<td><code>ui.Dialog.getCurrent()</code> RETURNS <code>ui.Dialog</code></td>
<td>Returns the current dialog object.</td>
</tr>
<tr>
<td><code>ui.Dialog.setDefaultUnbuffered</code> (on BOOLEAN)</td>
<td>Set the default unbuffered mode for all dialogs.</td>
</tr>
</tbody>
</table>
### Table 531: Object methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>accept()</code></td>
<td>Validates and terminates the dialog.</td>
</tr>
<tr>
<td><code>addTrigger(</code></td>
<td>Adds an event trigger to the dynamic dialog.</td>
</tr>
<tr>
<td><code>appendNode(</code></td>
<td>Appends a new node in the specified tree-view.</td>
</tr>
<tr>
<td><code>appendRow(</code></td>
<td>Appends a new row in the specified list.</td>
</tr>
<tr>
<td><code>arrayToVisualIndex(</code></td>
<td>Converts the program array index to the visual index for a given screen array.</td>
</tr>
<tr>
<td><code>cancel()</code></td>
<td>Cancels a parent dialog from a sub-dialog.</td>
</tr>
<tr>
<td><code>close()</code></td>
<td>Closes a dynamic dialog.</td>
</tr>
<tr>
<td><code>deleteAllRows(</code></td>
<td>Deletes all rows from the specified list.</td>
</tr>
<tr>
<td><code>deleteNode(</code></td>
<td>Deletes a node from the specified tree-view.</td>
</tr>
<tr>
<td><code>deleteRow(</code></td>
<td>Deletes a row from the specified list.</td>
</tr>
<tr>
<td><code>getArrayLength()</code></td>
<td>Returns the total number of rows in the specified list.</td>
</tr>
<tr>
<td><code>getCurrentItem()</code></td>
<td>Returns the current item having focus.</td>
</tr>
<tr>
<td><code>getCurrentRow()</code></td>
<td>Returns the current row of the specified list.</td>
</tr>
<tr>
<td><code>getBuffer()</code></td>
<td>Returns the input buffer of the specified field.</td>
</tr>
<tr>
<td><code>getFieldBuffer()</code></td>
<td>Returns the modification flag for a field.</td>
</tr>
<tr>
<td><code>getFieldTouched()</code></td>
<td>Returns the modification flag for a field.</td>
</tr>
</tbody>
</table>
**ui.Dialog.createConstructByName**

Creates an `ui.Dialog` object to implement a dynamic CONSTRUCT BY NAME.

### Syntax

```
ui.Dialog.createConstructByName(
    fields DYNAMIC ARRAY OF RECORD
      name STRING,
      type STRING
    END RECORD
)
```

1. `fields` is the list of form fields controlled by the dialog. This must be a DYNAMIC ARRAY of RECORD structure, with a name and type member of type STRING.

### Usage

The `ui.Dialog.createConstructByName()` class method creates a new dialog object to implement the equivalent of a static CONSTRUCT block.

**Note:** The current form will be attached to the created dialog.

The method takes a list of field definitions as parameter, as described in Field definition for Dynamic Dialogs on page 2558.

**Important:** Static CONSTRUCT dialog instructions use the data types of the fields defined in the .per form specification file. Unlike static CONSTRUCT, the dynamic construct uses the data type provided in the dynamic array defining the list of fields.

### Example

```
DEFINE fields DYNAMIC ARRAY OF RECORD
  name STRING,
  type STRING
END RECORD
DEFINE d ui.Dialog
...
LET d = ui.Dialog.createConstructByName(fields)
...
```

### Related concepts

- **ui.Dialog.getQueryFromField** on page 2534
  Returns the SQL condition of a field used in a query by example dialog.

- **Dynamic Dialogs** on page 2004
  Dialogs can be created at runtime with the `ui.Dialog` class.

**ui.Dialog.createDisplayArrayTo**

Creates an `ui.Dialog` object to implement a dynamic DISPLAY ARRAY TO.

### Syntax

```
ui.Dialog.createDisplayArrayTo(
    fields DYNAMIC ARRAY OF RECORD
      name STRING,
      type STRING
    END RECORD,
    screenRecord STRING
)
```

1. `fields` is the list of form fields controlled by the dialog. This must be a DYNAMIC ARRAY of a RECORD structure, with a name and type member of type STRING.
2. `screenRecord` is the name of the screen array (defined with the SCREEN RECORD instruction in form files).

**Usage**

The `ui.Dialog.createDisplayArrayTo()` class method creates a dialog object to implement the equivalent of a static DISPLAY ARRAY TO block.

**Note:** The current form will be attached to the created dialog.

The method takes a list of field definitions as parameter, as described in Field definition for Dynamic Dialogs on page 2558.

The second parameter passed to the `createDisplayArrayTo()` method is the name of the screen record which groups the fields together, for the list view of the form.

For example, in the following form definition, the screen record name is "sr_custlist":

```plaintext
... INSTRUCTIONS
SCREEN RECORD sr_custlist
{
    customer.cust_id,
    customer.cust_name,
    ...
};
END
```

For more details, see Screen records / arrays on page 1251.

**Example**

```plaintext
DEFINE fields DYNAMIC ARRAY OF RECORD
    name STRING,
    type STRING
END RECORD
DEFINE d ui.Dialog
...
LET d = ui.Dialog.createDisplayArrayTo(fields, "sr_custlist")
...
```

**Related concepts**

Dynamic Dialogs on page 2004

Dialogs can be created at runtime with the `ui.Dialog` class.

`ui.Dialog.createInputArrayFrom`

Creates an `ui.Dialog` object to implement a dynamic INPUT ARRAY FROM.

**Syntax**

```plaintext
ui.Dialog.createInputArrayFrom(
    fields DYNAMIC ARRAY OF RECORD
    name STRING,
    type STRING
    END RECORD,
    screenRecord STRING )
```

1. `fields` is the list of form fields controlled by the dialog. This must be a DYNAMIC ARRAY of a RECORD structure, with a name and type member of type STRING.

2. `screenRecord` is the name of the screen array (defined with the SCREEN RECORD instruction in form files).
Usage

The `ui.Dialog.createInputArrayFrom()` class method creates a dialog object to implement the equivalent of a static `INPUT ARRAY FROM` block.

Note: The current form will be attached to the created dialog.

The method takes a list of field definitions as parameter, as described in Field definition for Dynamic Dialogs on page 2558.

A dynamic input array dialog behaves like a static `INPUT ARRAY` using the `WITHOUT DEFAULTS` option: The values set in the internal rows before starting the dialog will be used. However, like with a static `INPUT ARRAY`, when adding a new row, the `DEFAULT` attributes of the form-fields are used.

The second parameter passed to the `createInputArrayFrom()` method is the name of the screen record which groups the fields together, for the list view of the form.

For example, in the following form definition, the screen record name is "sr_custlist":

```plaintext
... 
INSTRUCTIONS 
SCREEN RECORD sr_custlist 
  ( 
    customer.cust_id, 
    customer.cust_name, 
    ... 
  ); 
END
```

For more details, see Screen records / arrays on page 1251.

Example

```plaintext
DEFINE fields DYNAMIC ARRAY OF RECORD 
  name STRING, 
  type STRING 
END RECORD
DEFINE d ui.Dialog 
... 
LET d = ui.Dialog.createInputArrayFrom(fields, "sr_custlist") 
... 
```

Related concepts

Dynamic Dialogs on page 2004

Dialogs can be created at runtime with the `ui.Dialog` class.

`ui.Dialog.createInputByName`

Creates an `ui.Dialog` object to implement a dynamic `INPUT BY NAME`.

Syntax

```plaintext
ui.Dialog.createInputByName( 
  fields DYNAMIC ARRAY OF RECORD 
    name STRING, 
    type STRING 
  END RECORD 
) 
```

1. `fields` is the list of form fields controlled by the dialog. This must be a `DYNAMIC ARRAY` of a `RECORD` structure, with a `name` and `type` member of type `STRING`. 
Usage
The `ui.Dialog.createInputByName()` class method creates a dialog object to implement the equivalent of a static `INPUT BY NAME` block.

Note: The current form will be attached to the created dialog.

The method takes a list of field definitions as parameter, as described in Field definition for Dynamic Dialogs on page 2558.

A dynamic input dialog behaves like a static `INPUT` dialog using the WITHOUT DEFAULTS option: The DEFAULT attribute of the form-field is not used.

Example

```plaintext
DEFINE fields DYNAMIC ARRAY OF RECORD
    name STRING,
    type STRING
END RECORD
DEFINE d ui.Dialog
...
LET d = ui.Dialog.createInputByName(fields)
...
```

Related concepts
- Dynamic Dialogs on page 2004
  Dialogs can be created at runtime with the `ui.Dialog` class.

- `ui.Dialog.createMultipleDialog`
  Creates an `ui.Dialog` object to implement a dynamic DIALOG multiple-dialog.

Syntax

```plaintext
ui.Dialog.createMultipleDialog( )
```

Usage
The `ui.Dialog.createMultipleDialog()` class method creates a dialog object to implement the equivalent of a static `DIALOG / END DIALOG` block.

Note: The current form will be attached to the created dialog.

The method takes no parameters.

After creating the multiple dialog object, you must add sub-dialogs with the following methods:

- `ui.Dialog.addConstructByName` on page 2520
- `ui.Dialog.addDisplayArrayTo` on page 2521
- `ui.Dialog.addInputArrayFrom` on page 2522
- `ui.Dialog.addInputByName` on page 2523

Use the `addTrigger()` method, to add global or sub-dialog triggers. The scope of the trigger is defined by the `addTrigger()` call order. See `addTrigger()` for more details.

Example

```plaintext
DEFINE fields DYNAMIC ARRAY OF RECORD
    name STRING,
    type STRING
END RECORD
DEFINE d ui.Dialog
...
LET d = ui.Dialog.createMultipleDialog()
```
CALL d.addTrigger("ON ACTION close") -- Adds a global trigger for the dialog
...
CALL d.addDisplayArrayTo(fields, "sr_custlist") -- Adds a DISPLAY ARRAY sub-dialog
CALL d.addTrigger("ON ACTION refresh") -- Adds a trigger for the DISPLAY ARRAY sub-dialog
...

Related concepts

Dynamic Dialogs on page 2004
Dialogs can be created at runtime with the ui.Dialog class.

ui.Dialog.getCurrent
Returns the current dialog object.

Syntax

ui.Dialog.getCurrent()
  RETURNS ui.Dialog

Usage

To get the current active dialog object, use the ui.Dialog.getCurrent() class method.

The method returns NULL if there is no current active dialog.

Example

FUNCTION field_disable(name)
  DEFINE name STRING
  DEFINE d ui.Dialog
  LET d = ui.Dialog.getCurrent()
  IF d IS NOT NULL THEN
    CALL d.setFieldActive(name, FALSE)
  END IF
END FUNCTION

ui.Dialog.setDefaultUnbuffered
Set the default unbuffered mode for all dialogs.

Syntax

ui.Dialog.setDefaultUnbuffered(
  on BOOLEAN )

1. on is a boolean to enable the unbuffered mode.

Usage

By default, modal dialogs are not sensitive to variable changes. To make a dialog sensitive, use the UNBUFFERED attribute in the dialog instruction definition.

To define the default for all subsequent dialogs, use the setDefaultUnbuffered() class method:

CALL ui.Dialog.setDefaultUnbuffered(TRUE)

Note: Only singular and multiple dialogs are sensitive to this API, parallel dialogs implicitly use the unbuffered mode.
**Related concepts**

*The buffered and unbuffered modes* on page 1720

The buffered and unbuffered mode control the synchronization of program variables and form fields.

**ui.Dialog.accept**

Validates and terminates the dialog.

**Syntax**

```plaintext
accept()
```

**Usage**

Use the `accept()` method to validate field input and terminate the dialog.

The `accept()` method is equivalent to the `ACCEPT_INPUT/ACCEPT_DISPLAY/ACCEPT_DIALOG` instructions.

It can be used for example to terminate the dialog in a function, outside the context of a dialog block, where control instructions cannot be used.

**Tip:** To skip code following the call to the `accept()` method, use the `CONTINUE_DIALOG` instruction: Combining `accept() + CONTINUE_DIALOG` is equivalent to the `ACCEPT_DIALOG` instruction.

When calling the `accept()` method, the `AFTER_INPUT`, `AFTER_DISPLAY` or `AFTER_CONSTRUCT` block of the current subdialog is executed, then the `AFTER_DIALOG` block is executed.

Typical dialog validation rules are performed when calling this method. See `ACCEPT_DIALOG` for more details.

**Example**

```plaintext
DIALOG mysubinput()
   INPUT BY NAME ...
   ...
   ON ACTION accept
      CALL DIALOG.accept()
   END INPUT
END DIALOG
```

**Related concepts**

*ui.Dialog.cancel* on page 2527

Cancels a parent dialog from a sub-dialog.

*ui.Dialog.validate* on page 2553

Checks form level validation rules.

*ui.Dialog.addConstructByName*

Adds a sub-dialog of type `CONSTRUCT BY NAME` to an existing `ui.Dialog` dynamic dialog.

**Syntax**

```plaintext
ui.Dialog.addConstructByName(
   fields DYNAMIC ARRAY OF RECORD
      name STRING,
      type STRING
   END RECORD,
   name STRING
)
```

1. `fields` is the list of form fields controlled by the dialog. This must be a `DYNAMIC ARRAY OF RECORD` structure, with a `name` and `type` member of type `STRING`. 

2. *name* is the sub-dialog identifier. This name is used to identify the sub-dialog in dialog events and API calls.

**Usage**

The `ui.Dialog.addConstructByName()` class method adds a sub-dialog equivalent to a `CONSTRUCT BY NAME` block, to the dynamic multiple dialog created with `ui.Dialog.createMultipleDialog` on page 2518.

The method takes a list of field definitions as parameter, as described in *Field definition for Dynamic Dialogs* on page 2558.

This second parameter will be used to identify the sub-dialog.

**Important:** Static `CONSTRUCT` dialog instructions use the data types of the fields defined in the `.per` form specification file. Unlike static `CONSTRUCT`, the dynamic construct uses the data type provided in the dynamic array defining the list of fields.

**Example**

```plaintext
DEFINE fields DYNAMIC ARRAY OF RECORD
   name STRING,
   type STRING
END RECORD
DEFINE d ui.Dialog
...
LET d = ui.Dialog.createMultipleDialog()
...
CALL d.addConstructByName(fields,"const1")
...
```

**Related concepts**

`ui.Dialog.getQueryFromField` on page 2534
Returns the SQL condition of a field used in a query by example dialog.

`Dynamic Dialogs` on page 2004
Dialogs can be created at runtime with the `ui.Dialog` class.

`ui.Dialog.addDisplayArrayTo`
Adds a sub-dialog of type `DISPLAY ARRAY TO` to an existing `ui.Dialog` dynamic dialog.

**Syntax**

```plaintext
ui.Dialog.addDisplayArrayTo(
   fields DYNAMIC ARRAY OF RECORD
      name STRING,
      type STRING
   END RECORD,
   screenRecord STRING )
```

1. *fields* is the list of form fields controlled by the dialog. This must be a `DYNAMIC ARRAY` of a `RECORD` structure, with a *name* and *type* member of type `STRING`.
2. *screenRecord* is the name of the screen array (defined with the `SCREEN RECORD` instruction in form files). This name will also identify the sub-dialog in dialog events and API calls.

**Usage**

The `ui.Dialog.addDisplayArrayTo()` class method adds a sub-dialog equivalent to a `DISPLAY ARRAY TO` block, to the dynamic multiple dialog created with `ui.Dialog.createMultipleDialog` on page 2518.

The method takes a list of field definitions as parameter, as described in *Field definition for Dynamic Dialogs* on page 2558.
The second parameter passed to the addDisplayArrayTo() method is the name of the screen record which groups the fields together, for the list view of the form. This second parameter will also be used to identify the sub-dialog.

For example, in the following form definition, the screen record name is "sr_custlist":

```
... INSTRUCTIONS
SCREEN RECORD sr_custlist
  ( customer.cust_id,
    customer.cust_name,
    ... )
END
```

For more details, see Screen records / arrays on page 1251.

**Example**

```
DEFINE fields DYNAMIC ARRAY OF RECORD
  name STRING,
  type STRING
END RECORD
DEFINE d ui.Dialog
... LET d = ui.Dialog.createMultipleDialog() ... CALL d.addDisplayArrayTo(fields, "sr_custlist") ...
```

**Related concepts**

Dynamic Dialogs on page 2004
Dialogs can be created at runtime with the ui.Dialog class.

**ui.Dialog.addInputArrayFrom**
Adds a sub-dialog of type INPUT ARRAY FROM to an existing ui.Dialog dynamic dialog.

**Syntax**

```
ui.Dialog.addInputArrayFrom(
  fields DYNAMIC ARRAY OF RECORD
    name STRING,
    type STRING
END RECORD,
  screenRecord STRING )
```

1. *fields* is the list of form fields controlled by the dialog. This must be a DYNAMIC ARRAY of a RECORD structure, with a name and type member of type STRING.
2. *screenRecord* is the name of the screen array (defined with the SCREEN RECORD instruction in form files). This name will also identify the sub-dialog in dialog events and API calls.

**Usage**

The ui.Dialog.addInputArrayFrom() class method adds a sub-dialog equivalent to a INPUT ARRAY FROM block, to the dynamic multiple dialog created with ui.Dialog.createMultipleDialog on page 2518.

The method takes a list of field definitions as parameter, as described in Field definition for Dynamic Dialogs on page 2558.
The second parameter passed to the `addInputArrayFrom()` method is the name of the screen record which groups the fields together, for the list view of the form. This second parameter will also be used to identify the sub-dialog.

For example, in the following form definition, the screen record name is "sr_custlist":

```plaintext
... INSTRUCTIONS
SCREEN RECORD sr_custlist
(  customer.cust_id,
  customer.cust_name,
  ... );
END
```

For more details, see Screen records / arrays on page 1251.

Example

```plaintext
DEFINE fields DYNAMIC ARRAY OF RECORD
  name STRING,
  type STRING
END RECORD
DEFINE d ui.Dialog
... LET d = ui.Dialog.createMultipleDialog()
... CALL d.addInputArrayFrom(fields, "sr_custlist")
... 
```

Related concepts

Dynamic Dialogs on page 2004
Dialogs can be created at runtime with the `ui.Dialog` class.

`ui.Dialog.addInputByName`
Adds a sub-dialog of type INPUT BY NAME to an existing `ui.Dialog` dynamic dialog.

Syntax

```plaintext
ui.Dialog.addInputByName(
  fields DYNAMIC ARRAY OF RECORD
    name STRING,
    type STRING
  END RECORD,
  name STRING
)
```

1. `fields` is the list of form fields controlled by the dialog. This must be a DYNAMIC ARRAY of a RECORD structure, with a name and type member of type STRING.
2. `name` is the sub-dialog identifier. This name is used to identify the sub-dialog in dialog events and API calls.

Usage

The `ui.Dialog.addInputByName()` class method adds a sub-dialog equivalent to an INPUT BY NAME block, to the dynamic multiple dialog created with `ui.Dialog.createMultipleDialog` on page 2518.

The method takes a list of field definitions as parameter, as described in Field definition for Dynamic Dialogs on page 2558.

This second parameter will be used to identify the sub-dialog.
Example

```
DEFINE fields DYNAMIC ARRAY OF RECORD
                name STRING,
                type STRING
        END RECORD
DEFINE d ui.Dialog
...
LET d = ui.Dialog.createMultipleDialog()
...
CALL d.addInputByName(fields,"input1")
...
```

Related concepts

Dynamic Dialogs on page 2004
Dialogs can be created at runtime with the `ui.Dialog` class.

`ui.Dialog.addTrigger`
Adds an event trigger to the dynamic dialog

Syntax

```
addTrigger(
    trigger STRING )
```

1. `trigger` is the name of the dialog.

Usage

When implementing a dynamic dialog, the `addTrigger()` method must be used to register user-defined triggers:

```
CALL d.addTrigger("ON ACTION print")
```

Registered dialog triggers are then typically managed in a `WHILE` loop using the `nextEvent()` method, to wait for dialog events.

Predefined triggers such as "BEFORE ROW","BEFORE FIELD","ON CHANGE" are already registered in dynamic dialogs. Such triggers do not have to be added with the `addTrigger()` method. User code can be implemented for predefined triggers when returned from the `nextEvent()` method.

Dynamic multiple dialogs are implemented with `ui.Dialog.createMultipleDialog` on page 2518. When calling the `addTrigger()` method just after `createMultipleDialog()`, it will add a global dialog trigger. When called after methods such as `ui.Dialog.addDisplayArrayTo` on page 2521, it will add a local trigger to the last added sub-dialog.

Sub-dialog actions that are created in the context of a sub-dialog with `addTrigger("ON ACTION action-name")` will be returned as "ON ACTION sub-dialog-name.action-name" from the `nextEvent()` method.

The following triggers are accepted by the `addTrigger()` method:

**Table 532: User-defined triggers for dynamic dialogs**

<table>
<thead>
<tr>
<th>Trigger name</th>
<th>Description</th>
<th>Dialog block equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ACTION action-name</code></td>
<td>Action handler for the action identified by <code>action-name</code>.</td>
<td><code>ACTION block</code></td>
</tr>
<tr>
<td><code>APPEND</code></td>
<td>Row addition action handler for a display array dynamic dialog.</td>
<td><code>APPEND block</code></td>
</tr>
<tr>
<td>Trigger name</td>
<td>Description</td>
<td>Dialog block equivalent</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>ON DELETE</td>
<td>Row deletion action handler for a display array dynamic dialog.</td>
<td>ON DELETE block</td>
</tr>
<tr>
<td>ON FILL BUFFER</td>
<td>Trigger to fill the current page of a paged mode display array dynamic dialog.</td>
<td>ON FILL BUFFER block</td>
</tr>
<tr>
<td>ON INSERT</td>
<td>Row insertion action handler for a display array dynamic dialog.</td>
<td>ON INSERT block</td>
</tr>
<tr>
<td>ON UPDATE</td>
<td>Row modification action handler for a display array dynamic dialog.</td>
<td>ON UPDATE block</td>
</tr>
</tbody>
</table>

**Related concepts**

- **ui.Dialog.nextEvent** on page 2537
  Waits for a dialog event.

- **Dynamic Dialogs** on page 2004
  Dialogs can be created at runtime with the `ui.Dialog` class.

- **ui.Dialog.appendRow**
  Appends a new row in the specified list.

**Syntax**

```javascript
appendRow(  
  name STRING )
```

1. `name` is the name of the screen record, see [Identifying screen-arrays in ui.Dialog methods](#) on page 2557.

**Usage**

The `appendRow()` method appends a row to the end of the array controlled by the dialog.

**Important:** This method is designed to be used in an **ON ACTION** block. It must not be called in control blocks such as BEFORE ROW, AFTER ROW, BEFORE INSERT, AFTER INSERT, BEFORE DELETE, or AFTER DELETE.

The method is similar to appending a new element to the program array, except the internal dialog registers are automatically updated (like the total number of rows returned by `getArrayLength()`). If the list is decorated with **cell attributes**, the program array defining the attributes will also be synchronized. If **multi-row selection** is enabled, selection flags of existing rows are kept. The new row is inserted at the end of the list with the selection flag set to zero.

**Note:** The purpose of this method is to implement business logic required to modify the record list in the current dialog. It is typically used in a **DISPLAY ARRAY** dialog. Avoid using this method in **INPUT ARRAY**. To allow the end user to append, modify or delete rows in a **DISPLAY ARRAY**, use list modification interaction blocks.

After the method is called, a new row is created in the program array. You can assign values to the variables before the control goes back to the user. The `getArrayLength()` method will return the new row count.

The method does not set the current row and does not give the focus to the list; you need to call `setCurrentRow()` and execute **NEXT FIELD** to give the focus.

This method does not execute any **BEFORE ROW**, **BEFORE INSERT**, **AFTER INSERT** or **AFTER ROW** control blocks.

The `appendRow()` method does not create a **temporary row** as the implicit append action of **INPUT ARRAY**; the row is considered permanent once it is added.
Example
This example implements a user-defined action to append ten rows at the end of the list.

```
ON ACTION append_some_rows
  FOR i = 1 TO 10
    CALL DIALOG.appendRow("sa")
    LET r = DIALOG.getArrayLength("sa")
    LET p_items[r].item_quantity = 1.00
  END FOR
```

**ui.Dialog.appendNode**
Appends a new node in the specified tree-view.

### Syntax

```
appendNode(
  name STRING,
  parentIndex INTEGER )
```

1. *name* is the name of the screen record, see Identifying screen-arrays in **ui.Dialog** methods on page 2557.
2. *parentIndex* is the index of the parent node in the program array (starts at 1).

### Usage
The `appendNode()` method adds a new node under a given parent, when the dialog controls a tree view.

This method must be used when modifying the array of a tree view during the execution of the dialog, for example when implementing a dynamic tree with `ON EXPAND` / `ON COLLAPSE` triggers. Before the execution of the dialog, you can fill the program array directly. This includes the context of `BEFORE DISPLAY` or `BEFORE DIALOG` control blocks.

When adding rows for a tree view, the id of the parent node and new node matters because that information is used to build the internal tree structure. When calling `appendNode()`, you pass the index of the parent node under which the new node will be appended. In the program array, the parent-id member of the new node will automatically be initialized with the value of the id of the parent node identified by the index passed as parameter, then the internal tree structure is rebuilt.

If the parent index is zero, a new root node will be appended.

The method returns the index of the new inserted node.

In the program array, the parent-id member of the new node will automatically be initialized with the value of the id member of the parent node identified by the index.

```
DISPLAY ARRAY mytree TO sr.*
  ... 
  ON EXPAND(id)
    CALL DIALOG.appendNode("sr", id)
  ... 
```

### Related concepts
- Dynamic filling of very large trees on page 1896
How to optimize the implementation of large tree-views?

*ui.Dialog.arrayToVisualIndex*

Converts the program array index to the visual index for a given screen array.

**Syntax**

```plaintext
arrayToVisualIndex(
    name STRING,
    arrayIndex INTEGER )
```

1. *name* is the name of the screen record, see Identifying screen-arrays in *ui.Dialog methods* on page 2557.
2. *arrayIndex* is the index of the program array row.

**Usage**

When the end user sorts rows in a table, the program array index may differ from the visual row index. Use this method to convert a program array row index (`arr_curr()`) to a row index as seen by the end user. For example, if you want to display a typical message with `(current-row / total-rows)`, convert the current program array row to a visual row index before displaying the value:

```plaintext
MESSAGE SFMT( "Row: %1/%2",
    DIALOG.arrayToVisualIndex( "sr", DIALOG.getCurrentRow("sr") ),
    DIALOG.getArrayLength( "sr" )
)
```

**Related concepts**

- **Handling the current row** on page 1835
  Query and control the current row in a read-only or editable list of records.
- **ON SORT block** on page 1520
- *ui.Dialog.visualToArrayIndex* on page 2553
  Converts the visual index to the program array index for a given screen array.
- *ui.Dialog.getCurrentRow* on page 2531
  Returns the current row of the specified list.
- *ui.Dialog.getArrayLength* on page 2530
  Returns the total number of rows in the specified list.

*ui.Dialog.cancel*

Cancels a parent dialog from a sub-dialog.

**Syntax**

```plaintext
cancel()
```

**Usage**

The `cancel()` method can be used to terminate a dialog by ignoring the current input. This method can be used to terminate the dialog in a function, outside the context of a dialog block, where control instructions cannot be used.

**Tip:** To skip code following the call to the `cancel()` method, use the `CONTINUE DIALOG` instruction: Combining `cancel() + CONTINUE DIALOG` is equivalent to the `CANCEL DIALOG` instruction.

When calling the `cancel()` method, the `INT_FLAG` is set to `TRUE`, the `AFTER INPUT`, `AFTER DISPLAY` or `AFTER CONSTRUCT` block of the current subdialog is executed, then the `AFTER DIALOG` block is executed.
Note: The default settings regarding action attributes for the `cancel` action define the `validate` attribute to "no", in order to avoid current field validation for this action. This is important when using the UNBUFFERED mode. For more details, see Actions configuration for field validation on page 1722.

Example

```plaintext
DIALOG mysubinput()
   INPUT BY NAME ...
   ...
   ON ACTION cancel
      CALL DIALOG.cancel()
   END INPUT
END DIALOG
```

Related concepts

ui.Dialog.accept on page 2520
Validates and terminates the dialog.

ui.Dialog.close
Closes a dynamic dialog.

Syntax

```plaintext
close()
```

Usage

Use the `close()` method when the dynamic dialog is finished.

To fully destroy the dialog object, assign NULL to the variable referencing it.

Example

```plaintext
...
   WHEN "ON ACTION cancel"
      EXIT WHILE
   END WHILE
   CALL d.close()
   LET d = NULL
   CLOSE WINDOW w1
```

Related concepts

Ending dynamic dialogs on page 2009
Describes how to terminate dynamic dialogs.

ui.Dialog.deleteAllRows
Deletes all rows from the specified list.

Syntax

```plaintext
deleteAllRows(
   name STRING )
```

1. `name` is the name of the screen record, see Identifying screen-arrays in ui.Dialog methods on page 2557.
Usage

The `deleteAllRows()` method removes all the rows of a list driven by a `DISPLAY ARRAY` or `INPUT ARRAY`. This is equivalent to a `deleteRow()` call, but instead of deleting one particular row, it removes all rows of the specified list.

This method must not be called in control blocks such as `BEFORE ROW`, `AFTER ROW`, `BEFORE INSERT`, `AFTER INSERT`, `BEFORE DELETE`, `AFTER DELETE`, it is designed to be used in an `ON ACTION` block.

After the method is called, all rows are deleted from the program array, and the `getArrayLength()` method will return zero.

The method takes the name of the screen-array as parameter.

If the `deleteAllRows()` method is called during an `INPUT ARRAY`, the dialog will automatically append a new temporary row if the focus is in the list, to let the user enter new data. When using `AUTO APPEND = FALSE` attribute, no temporary row will be created and the current row register will be automatically changed to make sure that it will not be greater than the total number of rows.

If the `deleteAllRows()` method is called during an `INPUT ARRAY` or `DISPLAY ARRAY` that has the focus, the `BEFORE ROW` control block will be executed if you delete the current row. This is required to reset the internal state of the dialog.

If the list was decorated with cell attributes, the program array defining the attributes will be cleared. If multi-row selection is enabled, selection flags are cleared.

`ui.Dialog.deleteNode`

Deletes a node from the specified tree-view.

Syntax

```plaintext
deleNode(
    name STRING,
    index INTEGER )
```

1. `name` is the name of the screen record, see Identifying screen-arrays in `ui.Dialog methods` on page 2557.
2. `index` is the index of the node in the program array that has to be deleted (starts at 1).

Usage

The `deleteNode()` method is similar to `deleteRow()`, except that it has to be used when the dialog controls a tree view.

This method must be used when modifying the array of a tree view during the execution of the dialog, for example when implementing a dynamic tree with `ON EXPAND` / `ON COLLAPSE` triggers. Before the execution of the dialog, you can fill the program array directly. This includes the context of `BEFORE DISPLAY` or `BEFORE DIALOG` control blocks.

The main difference with `deleteRow()` is that `deleteNode()` will remove recursively all child nodes before removing the node identified by index.

If the index is zero, all root nodes will be deleted from the tree.

`ui.Dialog.deleteRow`

Deletes a row from the specified list.

Syntax

```plaintext
deleteRow(
    name STRING,
    index INTEGER )
```
1. *name* is the name of the screen record, see *Identifying screen-arrays in ui.Dialog methods* on page 2557.

2. *index* is the index of the row to be deleted (starts a 1).

**Usage**

The `deleteRow()` method deletes the row in the array controlled by the dialog.

**Important:** This method is designed to be used in an `ON ACTION` block. It must not be called in control blocks such as `BEFORE ROW`, `AFTER ROW`, `BEFORE INSERT`, `AFTER INSERT`, `BEFORE DELETE`, or `AFTER DELETE`.

The method is similar to deleting an element to the program array, except that internal dialog registers are automatically updated (like the total number of rows returned by `getArrayLength()`). If the list is decorated with cell attributes, the program array defining the attributes will also be synchronized. If multi-row selection is enabled, selection information is synchronized (selection flags are shifted up) for all rows after the deleted row.

**Note:** The purpose of this method is to implement business logic required to modify the record list in the current dialog. It is typically used in a `DISPLAY ARRAY` dialog. Avoid using this method in `INPUT ARRAY`. To allow the end user to append, modify or delete rows in a `DISPLAY ARRAY`, use list modification interaction blocks.

After the method is called, the row no longer exists in the program array, and the `getArrayLength()` method will return the new row count.

If the `deleteRow()` method is called during an `INPUT ARRAY` that `has the focus`, control blocks such as `BEFORE ROW` and `BEFORE FIELD` will be executed, if you delete the `current row`. This is required to reset the internal state of the dialog. However, the method does not execute any `BEFORE ROW` or `AFTER ROW` control blocks in a `DISPLAY ARRAY` dialog.

If the `deleteRow()` method is called during an `INPUT ARRAY`, and if no more rows are in the list after the call, the dialog will automatically append a new temporary row if the focus is in the list, to let the user enter new data.

When using `AUTO APPEND = FALSE` attribute, no temporary row will be created and the current row register will be automatically changed to make sure that it will not be greater than the total number of rows.

If you pass zero as row index, the method does nothing (if no rows are in the list, `getCurrentRow()` returns zero).

**Example**

This example implements a user-defined action to remove rows that have a specific property:

```
ON ACTION delete_invalid_rows
  FOR r = 1 TO DIALOG.getArrayLength("sa")
    IF NOT s_orders[r].is_valid THEN
      CALL DIALOG.deleteRow("sa",r)
      LET r = r - 1
    END IF
  END FOR
```

`ui.Dialog.getArrayLength`

Returns the total number of rows in the specified list.

**Syntax**

```
getArrayLength(  
  name STRING  
)  
RETURNS INTEGER
```

1. *name* is the name of the screen record, see *Identifying screen-arrays in ui.Dialog methods* on page 2557.

**Usage**

The `getArrayLength()` method returns the total number of rows of an `INPUT ARRAY` or `DISPLAY ARRAY` list. The name of the screen array is passed as parameter to identify the list.
Example

```plaintext
MAIN
    DEFINE custlist DYNAMIC ARRAY OF RECORD
        pkey INT, name VARCHAR(50)
    END RECORD

    OPEN FORM f1 FROM "form1"
    DISPLAY FORM f1

    DISPLAY ARRAY custlist TO sa_custlist.*
    BEFORE DISPLAY
        MESSAGE "Row count: " || DIALOG.getArrayLength("sa_custlist")
    END DISPLAY

END MAIN
```

**ui.Dialog.getCurrentItem**

Returns the current item having focus.

**Syntax**

```plaintext
getCurrentItem()
    RETURNS STRING
```

**Usage**

The `getCurrentItem()` method returns the name of the current form item having the focus.

- If the focus is on an action view (typically, a **BUTTON** in the form layout), `getCurrentItem()` returns the name of the corresponding action. If several action views are bound to the same action handler with a unique name, there is no way to distinguish which action view has the focus.
- If the focus is in a simple field controlled by an **INPUT** or **CONSTRUCT** sub-dialog, `getCurrentItem()` returns the `tab-name.[field-name]` of that current field. The `tab-name` prefix is added if a FROM clause is used with an explicit list of fields. No prefix is added if `FROM screen-record.*` is used or if BY NAME clause is used.
- If the focus is in a list controlled by a **DISPLAY ARRAY** sub-dialog, `getCurrentItem()` returns the `screen-array` name identifying the list for a regular **DISPLAY ARRAY**. If the **DISPLAY ARRAY** is defined with the **FOCUSONFIELD** attribute, the method returns `screen-array.field-name`.
- If the focus is in a field of a list controlled by an **INPUT ARRAY** sub-dialog, `getCurrentItem()` returns `screen-array.field-name`, identifying both the list and the current field. In some context, the current field is undefined. For example when entering the **INPUT ARRAY** sub-dialog, `getCurrentItem()` will return the `screen-array` only when in the **BEFORE INPUT** control block.

**ui.Dialog.getCurrentRow**

Returns the current row of the specified list.

**Syntax**

```plaintext
getCurrentRow(
    name STRING
)  
    RETURNS INTEGER
```

1. *name* is the name of the screen record, see **Identifying screen-arrays in ui.Dialog methods** on page 2557.

**Usage**

Use the `getCurrentRow()` method to retrieve the current row of an **INPUT ARRAY** or **DISPLAY ARRAY** list.
You must pass the name of the screen array to identify the list.

```
DIALOG
   DISPLAY ARRAY custlist TO sa_custlist.*
   BEFORE ROW
      MESSAGE "Current row: " || DIALOG.getCurrentRow("sa_custlist")
      ...
   END DISPLAY
   INPUT ARRAY ordlist TO sa_ordlist.*
   BEFORE ROW
      MESSAGE "Current row: " || DIALOG.getCurrentRow("sa_ordlist")
      ...
   END INPUT
   ...
```

`ui(Dialog.getFieldBuffer)`

Returns the input buffer of the specified field.

### Syntax

```
getFieldBuffer(
   name STRING )
RETURNS STRING
```

1. `name` is the form field name, see Identifying fields in ui.Dialog methods on page 2556.

### Usage

The `getFieldBuffer()` method returns the current input buffer of the specified field. The input buffer is used by the dialog to synchronize form fields and program variables. In some situations, especially when using the buffered mode or in a `CONSTRUCT`, you may want to access the field input buffer.

**Note:** Use of this method is only recommended in dialogs allowing field input (INPUT, INPUT ARRAY, CONSTRUCT). The behavior is undefined when used in DISPLAY ARRAY.

The parameter is a field specification, a string containing the field qualifier, with an optional prefix "([table.]column").

```
LET buff = DIALOG.getFieldBuffer("customer.cust_name")
```

The input buffer can be set with:

- A `DISPLAY TO` or `DISPLAY BY NAME` instruction
- The `FGL_DIALOG_SETBUFFER()` function (only for the current field)

`ui(Dialog.getFieldTouched)`

Returns the modification flag for a field.

### Syntax

```
getFieldTouched(
   fields STRING )
RETURNS BOOLEAN
```

1. `fields` is the string with the list of field specification, see Identifying fields in ui.Dialog methods on page 2556.

### Usage

The `getFieldTouched()` method returns `TRUE` if the modification flag of the specified field(s) is set.
The `fields` parameter is a string containing the field qualifier, with an optional prefix ("[table.]column"), a table prefix followed by a dot and an asterisk ("table.*"), or a simple asterisk ("*").

This code checks if a specific field has been touched:

```plaintext
AFTER FIELD cust_name
  IF DIALOG.getFieldTouched("customer.cust_address") THEN
  ...
```

If the parameter is a screen record following by dot-asterisk, the method checks the touched flags of all the fields that belong to the screen record:

```plaintext
ON ACTION quit
  IF DIALOG.getFieldTouched("customer.*") THEN
  ...
```

When passing a simple asterisk (*) to the method, the runtime system will check all fields used by the dialog:

```plaintext
ON ACTION quit
  IF DIALOG.getFieldTouched("*") THEN
  ...
```

**`ui.Dialog.getFieldValue`**

Returns the value of a field controlled by a dynamic dialog.

### Syntax

```plaintext
getFieldValue(
  name STRING )
RETURNS fgl-type
```

1. `name` is the name of the form field, see Identifying fields in `ui.Dialog` methods on page 2556.
2. `fgl-type` is one of the primitive data types.

### Usage

The `getFieldValue()` method can be used when implementing a dynamic dialog, to return the value of a field:

```plaintext
DISPLAY d.getFieldValue( "customer.cust_addr" )
```

In a dynamic dialog controlling a list of records (INPUT ARRAY / DISPLAY ARRAY), this method returns the value for a field in the current row.

**Important:**

- During dialog execution, the `getFieldValue()` method must only be used to get the value of a field for the current row. Calling the `setCurrentRow()` method to change the current row before calling `getFieldValue()` will have no effect.
- In a `DISPLAY ARRAY` using the paged mode (ON FILL BUFFER), `getFieldValue()` returns NULL, if the current row is not part of the visible page. In a `DISPLAY ARRAY` using the full list mode, a NULL value is returned, if there is no current row (when the array is empty).

**Related concepts**

- `ui.Dialog.setFieldValue` on page 2550
  Sets the value of a field controlled by the dialog object.

- Dynamic Dialogs on page 2004
Dialogs can be created at runtime with the `ui.Dialog` class.

`ui.Dialog.getForm`
Returns the current form used by the dialog.

**Syntax**

```java
getForm()
RETURNS ui.Form
```

**Usage**

The `getForm()` method returns a `ui.Form` object as a handle to the current form used by the dialog.

Use this form object to modify elements of the current form. For example, you can hide some parts of the form with the `ui.Form.setElementHidden()` method.

`ui.Dialog.getQueryFromField`
Returns the SQL condition of a field used in a query by example dialog.

**Syntax**

```java
getQueryFromField(
    name STRING
) 
RETURNS STRING
```

1. `name` is the name of the form field, see Identifying fields in `ui.Dialog` methods on page 2556.

**Usage**

The `getQueryFromField()` method generates the SQL condition from the value entered in the field specified by the `field-name` parameter.

This method is used in the context of a construct dynamic dialog.

The result of this method can be used to build the `WHERE` part of a `SELECT` statement to find rows in a database.

Collect and concatenate field conditions returned from `getQueryFromField()`, then add AND or OR boolean operators to create an executable SQL query.

**Note:** The SQL condition is generated based on the current type of database. The SQL syntax may vary depending on the target database. Therefore it is not recommended to reuse the generated SQL conditions. However, the user input of a query by example dialog can be reused for different types of databases (see `ui.Dialog.getFieldValue` on page 2533)

**Related concepts**

`ui.Dialog.createConstructByName` on page 2515
Creates an `ui.Dialog` object to implement a dynamic CONSTRUCT BY NAME.

`ui.Dialog.getSortKey`
Returns the name of the sort field selected by the user.

**Syntax**

```java
getSortKey(
    name STRING
) 
RETURNS STRING
```

1. `name` is the name of the screen record, see Identifying screen-arrays in `ui.Dialog` methods on page 2557.
**Usage**

The `getSortKey()` method returns the form field name selected by the user to sort rows.

This method is used in the context of the `ON SORT` trigger.

**Note**: If the sort is reset, the `getSortKey()` method returns `NULL`.

**Related concepts**

- **ON SORT block** on page 1520
- **Populating a DISPLAY ARRAY** on page 1841
  - The program array must be filled with rows to populate the DISPLAY ARRAY dialog.
- **ui.Dialog.isSortReverse** on page 2537
  - Indicates the sort order direction (FALSE=ascending, TRUE=descending)

**ui.Dialog.insertNode**

Inserts a new node in the specified tree.

**Syntax**

```plaintext
insertNode(
    name STRING,
    index INTEGER
)
```

1. `name` is the name of the screen record, see **Identifying screen-arrays in ui.Dialog methods** on page 2557.
2. `index` is the index of the next sibling node in the program array (starts at 1).

**Usage**

The `insertNode()` method is similar to `insertRow()`, except that it has to be used when the list dialog controls a tree view.

This method must be used when modifying the array of a tree view during the execution of the dialog, for example when implementing a dynamic tree with `ON EXPAND`/`ON COLLAPSE` triggers. Before the execution of the dialog, you can fill the program array directly. This includes the context of `BEFORE DISPLAY` or `BEFORE DIALOG` control blocks.

When adding rows for a tree view, the id of the parent node and new node matters because that information is used to build the internal tree structure. When calling `insertNode()`, you pass the index of the next sibling node. In the program array, the parent-id member of the new node will be automatically initialized with the value of the parent-id of the next sibling node, then the internal tree structure is rebuilt.

**ui.Dialog.insertRow**

Inserts a new row in the specified list.

**Syntax**

```plaintext
insertRow(
    name STRING,
    index INTEGER
)
```

1. `name` is the name of the screen record, see **Identifying screen-arrays in ui.Dialog methods** on page 2557.
2. `index` is the index where the row must be inserted (starts at 1).

**Usage**

The `insertRow()` method inserts a row in the list, at a given position.
**Important:** This method is designed to be used in an ON ACTION block. It must not be called in control blocks such as BEFORE ROW, AFTER ROW, BEFORE INSERT, AFTER INSERT, BEFORE DELETE, or AFTER DELETE.

The method is similar to inserting a new element in the program array, except the internal dialog registers are automatically updated (like the total number of rows returned by `getArrayLength()`). If the list is decorated with cell attributes, the program array defining the attributes will also be synchronized. If multi-row selection is enabled, selection flags of existing rows are kept. Selection information is synchronized (i.e., flags are shifted down) for all rows after the new inserted row.

**Note:** The purpose of this method is to implement business logic required to modify the record list in the current dialog. It is typically used in a DISPLAY ARRAY dialog. Avoid using this method in INPUT ARRAY. To allow the end user to append, modify or delete rows in a DISPLAY ARRAY, use list modification interaction blocks.

After the method is called, a new row is created in the program array, so you can assign values to the variables before the control goes back to the user. The `getArrayLength()` method will return the new row count.

The method does not set the current row and does not give the focus to the list; you need to call `setCurrentRow()` and execute `NEXT FIELD` to give the focus.

The `insertRow()` method must not be used when controlling a tree view. Use the `insertNode()` method instead.

This method does not execute any BEFORE ROW, BEFORE INSERT, AFTER INSERT or AFTER ROW control blocks.

If the index is greater than the number of rows, a new row is appended at the end of the list. This is the equivalent of calling the `appendRow()` method.

If the list is empty, `getCurrentRow()` returns zero. If zero is returned, use 1 to reference the first row, otherwise you can get a -1326 error when using the program array.

**Example**

This example shows a user-defined action to insert ten rows in the list at the current position:

```plaintext
ON ACTION insert_some_rows
  LET r = DIALOG.getCurrentRow("sa")
  IF r == 0 THEN LET r = 1 END IF
  FOR i = 10 TO 1 STEP -1
    CALL DIALOG.insertRow("sa", r)
    LET p_items[r].item_quantity = 1.00
  END FOR
```

**ui.Dialog.isRowSelected**

Queries row selection for a given list and row.

**Syntax**

```plaintext
isRowSelected(
  name STRING,
  row INTEGER )
RETURNS BOOLEAN
```

1. `name` is the name of the screen record, see Identifying screen-arrays in ui.Dialog methods on page 2557.
2. `row` is a row index.
Usage

If multi-row selection is enabled with `setSelectionMode()`, you can check whether a row is selected with the `isRowSelected()` method:

```
ON ACTION check_current_row_selected
    IF DIALOG.isRowSelected( "sr", DIALOG.getCurrentRow("sr") ) THEN
        MESSAGE "Current row is selected."
    END IF
```

If multi-row selection is off, the method returns `TRUE` for the current row and `FALSE` for other rows.

`ui.Dialog.isSortReverse`

Indicates the sort order direction (FALSE=ascending, TRUE=descending)

Syntax

```
isSortReverse( name STRING )
RETURNS BOOLEAN
```

1. `name` is the name of the screen record, see Identifying screen-arrays in `ui.Dialog` methods on page 2557.

Usage

The `isSortReverse()` method returns `FALSE` if the sort order is ascending, and `TRUE` if the sort is in descending order.

This method is used in the context of the `ON SORT` trigger.

Related concepts

- **ON SORT block** on page 1520
- **Populating a DISPLAY ARRAY** on page 1841
  The program array must be filled with rows to populate the DISPLAY ARRAY dialog.
- **`ui.Dialog`**
  Returns the name of the sort field selected by the user.

- **`ui.Dialog`**
  `nextEvent`

Waits for a dialog event.

Syntax

```
extEvent()
RETURNS STRING
```

1. `event` is the name of the dialog event that raised.

Usage

The `nextEvent()` waits for a dialog event to occur, and returns a string that identifies the dialog event that is raised.

This method is typically used in a `WHILE` loop, to implement a dynamic dialog.

The method can return `NULL`, if a dialog error occurs, or if the dialog terminates (with `ui.Dialog.close()`).

The recommended programming pattern for the event `WHILE` loop is to test for nulls:

```
DEFINE d ui.Dialog,
    t STRING
```
A dialog event can be a user-defined trigger such as "ON ACTION print", or an implicit trigger such as "BEFORE ROW", corresponding to the control blocks that can be defined in static dialog instructions such as DISPLAY ARRAY.

User-defined triggers are added to the dynamic dialog with the addTrigger() method:

**Table 533: User-defined triggers for dynamic dialogs**

<table>
<thead>
<tr>
<th>Trigger name</th>
<th>Description</th>
<th>Dialog block equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>ON ACTION action-name</td>
<td>Action handler for the action identified by action-name.</td>
<td>ON ACTION block</td>
</tr>
<tr>
<td>ON APPEND</td>
<td>Row addition action handler for a display array dynamic dialog.</td>
<td>ON APPEND block</td>
</tr>
<tr>
<td>ON DELETE</td>
<td>Row deletion action handler for a display array dynamic dialog.</td>
<td>ON DELETE block</td>
</tr>
<tr>
<td>ON FILL BUFFER</td>
<td>Trigger to fill the current page of a paged mode display array dynamic dialog.</td>
<td>ON FILL BUFFER block</td>
</tr>
<tr>
<td>ON INSERT</td>
<td>Row insertion action handler for a display array dynamic dialog.</td>
<td>ON INSERT block</td>
</tr>
<tr>
<td>ON UPDATE</td>
<td>Row modification action handler for a display array dynamic dialog.</td>
<td>ON UPDATE block</td>
</tr>
</tbody>
</table>

Implicit dialog triggers are predefined and can be detected and handled in the dialog WHILE loop if needed:

**Table 534: Implicit triggers for dynamic dialogs**

<table>
<thead>
<tr>
<th>Trigger name</th>
<th>Description</th>
<th>Dialog block equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEFORE DIALOG</td>
<td>Initialization of the multiple dynamic dialog.</td>
<td>BEFORE DIALOG block</td>
</tr>
<tr>
<td>AFTER DIALOG</td>
<td>End of the multiple dynamic dialog.</td>
<td>AFTER DIALOG block</td>
</tr>
<tr>
<td>BEFORE DISPLAY</td>
<td>In singular dynamic dialog, initialization of the display array. In multiple dynamic dialog, when display array gets focus.</td>
<td>BEFORE DISPLAY block</td>
</tr>
<tr>
<td>AFTER DISPLAY</td>
<td>In singular dynamic dialog, end of the display array. In multiple dynamic dialog, when display array loses focus.</td>
<td>AFTER DISPLAY block</td>
</tr>
<tr>
<td>BEFORE INPUT</td>
<td>In singular dynamic dialog, initialization of the input by name. In multiple dynamic dialog, when input by name gets focus.</td>
<td>BEFORE INPUT block</td>
</tr>
<tr>
<td>AFTER INPUT</td>
<td>In singular dynamic dialog, end of the input by name. In multiple dynamic dialog, when input by name loses focus.</td>
<td>AFTER INPUT block</td>
</tr>
<tr>
<td>BEFORE CONSTRUCT</td>
<td>In singular dynamic dialog, initialization of the construct. In multiple dynamic dialog, when construct gets focus.</td>
<td>BEFORE INPUT block</td>
</tr>
<tr>
<td>AFTER CONSTRUCT</td>
<td>In singular dynamic dialog, end of the construct. In multiple dynamic dialog, when construct loses focus.</td>
<td>AFTER INPUT block</td>
</tr>
</tbody>
</table>
### Related concepts

**Dynamic Dialogs** on page 2004  
Dialogs can be created at runtime with the `ui.Dialog` class.

**`ui.Dialog.nextField`**  
Registers the next field to go to.

### Syntax

```javascript
nextField(
    name STRING
)
```

1. `name` is the form field name, see Identifying fields in `ui.Dialog` methods on page 2556, or "+NEXT", "+PREV", "+CURR", to specify respectively the next, previous and current field.

### Usage

The `nextField()` method registers the name of the next field that must get the focus when control goes back to the dialog.

Like `NEXT FIELD`, the `nextField()` method is typically used in an editable dialog (`INPUT`, `CONSTRUCT`, `INPUT ARRAY`).

**Note:** In a `DISPLAY ARRAY` using the `FOCUSONFIELD` attribute, `nextField()` can be used in conjunction with `DIALOG.setCurrentRow()`, to set the focus to a specific cell in the list.

The `nextField()` method is similar to the `NEXT FIELD` instruction, except that it does not implicitly break the program flow. To get the same behavior as `NEXT FIELD`, the method call must be followed by a `CONTINUE DIALOG` instruction, or an equivalent instruction such as `CONTINUE INPUT`, in case of singular dialog.

<table>
<thead>
<tr>
<th>Trigger name</th>
<th>Description</th>
<th>Dialog block equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEFORE ROW</td>
<td>Moving to a new row in a display array or input array dynamic dialog.</td>
<td>BEFORE ROW block</td>
</tr>
<tr>
<td>AFTER ROW</td>
<td>Leaving the current row in a display array or input array dynamic dialog.</td>
<td>AFTER ROW block</td>
</tr>
<tr>
<td>BEFORE INSERT</td>
<td>Before a new row is created in an input array dynamic dialog.</td>
<td>BEFORE INSERT block</td>
</tr>
<tr>
<td>AFTER INSERT</td>
<td>After a new row is created in an input array dynamic dialog.</td>
<td>AFTER INSERT block</td>
</tr>
<tr>
<td>BEFORE DELETE</td>
<td>Before a new row is deleted in an input array dynamic dialog.</td>
<td>BEFORE DELETE block</td>
</tr>
<tr>
<td>AFTER DELETE</td>
<td>After a new row is deleted in an input array dynamic dialog.</td>
<td>AFTER DELETE block</td>
</tr>
<tr>
<td>BEFORE FIELD</td>
<td>Entering the field <code>field-name</code> in an input dynamic dialog.</td>
<td>BEFORE FIELD block</td>
</tr>
<tr>
<td>AFTER FIELD</td>
<td>Leaving the field <code>field-name</code> in an input dynamic dialog.</td>
<td>AFTER FIELD block</td>
</tr>
<tr>
<td>ON CHANGE</td>
<td>Value of field <code>field-name</code> changed in an input dynamic dialog.</td>
<td>ON CHANGE block</td>
</tr>
</tbody>
</table>
Since this method takes an expression as parameter, you can write generic code, when the name of the target field is not known at compile time. In the next example, the `check_value()` function returns a field name where the value does not satisfy the validation rules.

```plaintext
DEFINE fn STRING
...
ON ACTION save
  IF ( fn:= check_values() ) IS NOT NULL THEN
    CALL DIALOG.nextField(fn)
    CONTINUE DIALOG
  END IF
  CALL save_data()
...
```

When specifying "+NEXT" as parameter, the dialog will register the field next to the current field (like `NEXT FIELD NEXT`). When passing "+PREV", the dialog will register the previous field (like `NEXT FIELD PREVIOUS`). When passing "+CURR", it is equivalent to a `NEXT FIELD CURRENT` instruction:

```plaintext
...
ON ACTION save
  IF NOT valid_input() THEN
    CALL DIALOG.nextField("+CURR")
    CONTINUE DIALOG
  END IF
  CALL save_data()
...
```

Related concepts

**Giving the focus to a form element** on page 1731
How to force the focus to move or stay in a specific form element using program code.

**`ui.Dialog.selectionToString`**
Serializes data of the selected rows.

**Syntax**

```plaintext
selectionToString(
  name STRING )
RETURNS STRING
```

1. `name` is the name of the screen record, see **Identifying screen-arrays in `ui.Dialog` methods** on page 2557.

**Usage**

The `selectionToString()` method can be used to get a tab-separated value list of the selected rows.

When multi-row selection is disabled, the method serializes the current row.

You typically use this method along with drag & drop to fill the buffer, by using a text/plain MIME type, to export data to external applications.

```plaintext
ON ACTION serialize
  LET buff = DIALOG.selectionToString( "sr" )
```

Numeric and date data will be formatted based on current locale settings (**DBMONEY, DBDATE**).

The visual presentation of data is respected. The dialog will copy the rows in the sort order specified by the user, moved columns will appear in the same positions as in the table and hidden columns will be ignored. Note that phantom columns are not copied.
Items in the tab-separated record will be surrounded by double-quotes if the value contains special characters such as a newline, a double-quote, or controls characters with ASCII code < 0x20. Double-quotes in the value will be doubled.

*ui.Dialog.setActionActive*
Enabling and disabling dialog actions.

**Syntax**

```java
setActionActive(
    name STRING,
    active BOOLEAN )
```

1. `name` is the name of the action, see Identifying actions in dialog methods.
2. `active` is a boolean value.

**Usage**

Use the `setActionActive()` method to enable or disable an action.

```java
CALL DIALOG.setActionActive("zoom", FALSE)
```

The second parameter of the method must be a boolean expression that evaluates to 0 (FALSE) or 1 (TRUE).

**Related concepts**

Enabling and disabling actions on page 1763
By default, dialog actions are enabled. However, it is recommended that an action be disabled when not allowed in the current context.

*ui.Dialog.setActionComment*
Set the comment/hint of a default action view.

**Syntax**

```java
setActionComment(
    name STRING,
    comment STRING )
```

1. `name` is the name of the action, see Identifying actions in dialog methods.
2. `comment` is the comment to be set.

**Usage**

Use the `setActionComment()` method to define the comment/hint for the default view of an action.

```java
CALL DIALOG.setActionComment( "confirm", "Confirm current order" )
```

The first parameter identifies the action object of the dialog.

**Important:** This method will only affect the rendering of the default view for the specified action. Explicit action views (such as buttons in the form layout) will not get the new attribute value.

**Related concepts**

Configuring actions on page 1744
Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with *action attributes*.

`ui.Dialog.setActionHidden`
Showing or hiding a default action view.

**Syntax**

```java
setActionHidden(
    name STRING,
    hidden BOOLEAN )
```

1. `name` is the name of the action, see Identifying actions in dialog methods.
2. `hidden` is a boolean value.

**Usage**

Use the `setActionHidden()` method to hide the default view (and context menu option) of an action.

```java
CALL DIALOG.setActionHidden( "confirm", TRUE )
```

The first parameter identifies the action object of the dialog.

**Important:** This method will only affect the rendering of the default view for the specified action. Explicit action views (such as buttons in the form layout) will not get the new attribute value.

`ui.Dialog.setActionImage`
Set the image of a default action view.

**Syntax**

```java
setActionImage(
    name STRING,
    image STRING )
```

1. `name` is the name of the action, see Identifying actions in dialog methods.
2. `image` is the image name to be set.

**Usage**

Use the `setActionImage()` method to define the image of the default view of an action.

```java
CALL DIALOG.setActionImage( "confirm", "smiley" )
```

The first parameter identifies the action object of the dialog.

**Important:** This method will only affect the rendering of the default view for the specified action. Explicit action views (such as buttons in the form layout) will not get the new attribute value.

**Related concepts**

- `ui.Dialog.setActionText` on page 2543
  Defining the text of a default action view.
- `ui.Dialog.setActionComment` on page 2541
  Set the comment/hint of a default action view.

- Configuring actions on page 1744
Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with action attributes.

**ui.Dialog.setActionText**

Defining the text of a default action view.

**Syntax**

```Action
setActionText (  
    name STRING,  
    text STRING  
)
```

1. `name` is the name of the action, see Identifying actions in dialog methods.
2. `text` is the text to be set.

**Usage**

Use the `setActionText()` method to define the label of the default view of an action.

```Action
CALL DIALOG.setActionText( "confirm", "Confirm" )
```

The first parameter identifies the action object of the dialog.

**Important:** This method will only affect the rendering of the default view for the specified action. Explicit action views (such as buttons in the form layout) will not get the new attribute value.

**Related concepts**

- Configuring actions on page 1744
- Action attributes related to decoration, keyboard shortcuts, and behavior can be defined with action attributes.

**ui.Dialog.setArrayAttributes**

Define cell decoration attributes array for the specified list (singular or multiple dialogs).

**Syntax**

```Action
setArrayAttributes (  
    name STRING,  
    attributes dynamic-array-type  
)
```

1. `name` is the name of the screen record, see Identifying screen-arrays in ui.Dialog methods on page 2557.
2. `attributes` is a program array defining the cell attributes.
3. `dynamic-array-type` is a DYNAMIC ARRAY OF ... type, which can be:
   - A DYNAMIC ARRAY OF RECORD ... END RECORD (with the same structure as the data array)
   - A DYNAMIC ARRAY WITH DIMENSION 2 OF STRING (to define attributes in dynamic dialog when the row structure is defined at runtime)
   - A DYNAMIC ARRAY OF STRING (to define attributes for complete lines instead of individual cells)

**Usage**

In an INPUT ARRAY or DISPLAY ARRAY dialog, the `setArrayAttributes()` method can be used to specify display attributes for each cell, or for the complete row.

The `setArrayAttributes()` is typically used in a DIALOG block where several screen arrays are defined. The method takes the name of the screen array as first parameter, to identify the list to be decorated with colors. An equivalent method called `setCellAttributes()` can be used, for dialogs where only one screen array is defined.

Possible values for cell attributes are a combination of the following values:
• One of the supported color names, or an #RRGGBB value.
• The blink attribute: The text in the cell blinks.
• The bold attribute: The text in the cell renders with a bold font.
• The underline attribute: The text in the cell is underlined.
• The reverse attribute: When a color is specified, it is used as background color instead of foreground text color.

The cell attributes must be specified in lowercase, the color names must be in camelCase, and the separator is a blank space, for example:

**Note:** For the predefined (TTY) color names (like blue, red, yellow), the actual rendering color used by the front-ends is adapted to the reverse or normal mode: In normal mode, colors like "yellow" must be darker, to make the foreground text easily readable on an white background. In reverse mode (where the color specification applies to the background), the actual background color must be lighter, so that the black foreground text can be read more easily.

• "green": Foreground text color is green.
• "lightRed reverse": Background cell color is light red.
• "blue underline": Foreground text color is blue and the text font is underlined.
• "red bold reverse": Background cell color is red and the text font is bold.

The structure of the dynamic array containing the cell attributes can be:

• A DYNAMIC ARRAY OF RECORD, with the same structure as the data array
• A DYNAMIC ARRAY WITH DIMENSION 2 OF STRING, to define a flexible set of cell attributes (for dynamic dialogs)
• A DYNAMIC ARRAY OF STRING, to define attributes for complete lines

The following example defines a dynamic array with the same structure as the data array. Note however that the members of the attributes array use the STRING data type:

```plaintext
DEFINE data DYNAMIC ARRAY OF RECORD
  pkey INTEGER,
  name VARCHAR(50)
END RECORD
DEFINE attributes DYNAMIC ARRAY OF RECORD
  pkey STRING,
  name STRING
END RECORD
```

Similarly, the cell attributes array can be defined with a two-dimensional dynamic array of strings:

```plaintext
DEFINE attributes DYNAMIC ARRAY WITH DIMENSION 2 OF STRING
```

The advantage of a two-dimensional array is the flexibility, as it can define an unlimited number of cells for each row. This solution is typically used when implementing a dynamic dialog.

Finally, if you want to decorate complete lines instead of individual cells, use a simple dynamic array of strings:

```plaintext
DEFINE attributes DYNAMIC ARRAY OF STRING
```

Fill the display attributes array with color and video attributes:

```plaintext
FOR i=1 TO data.getLength() -- length from data array!
  LET attributes[i].name = "blue reverse"
END FOR
```

Then, attach the array to the dialog with the `setArrayAttributes()` method, in a BEFORE DIALOG, BEFORE INPUT or BEFORE DISPLAY block:

```plaintext
BEFORE DIALOG
```
CALL DIALOG.setArrayAttributes( "sr", attributes )

Like data values, if you change the cell attributes during the dialog, these are not displayed automatically unless the UNBUFFERED mode is used.

ON ACTION modify_cell_attribute
    LET attributes[arr_curr()].name = "red reverse"

If you set NULL to an element, the default TTY attributes will be reset:

ON ACTION clean_cell_attribute
    LET attributes[arr_curr()].name = NULL

Related concepts
Cell color attributes on page 1850
List controllers can display every cell in a specific color.

ui.Dialog.setCellAttributes on page 2546
Define cell decoration attributes array for the specified list (singular dialog only).

Example 4: Set display attributes for cells on page 2560

ui.Dialog.setArrayLength
Sets the total number of rows in the specified list.

Syntax

```plaintext
setArrayLength(
    name STRING,
    length INTEGER )
```

1. `name` is the name of the screen record, see Identifying screen-arrays in ui.Dialog methods on page 2557.
2. `length` is the new size of the array.

Usage
The setArrayLength() method is used to specify the total number of rows when using the DISPLAY ARRAY paged mode. The name of the screen array is passed to identify the list, followed by an integer expression defining the number of rows.

When using a regular dynamic array without paged mode (without the ON FILL BUFFER clause), you don't need to specify the total number of rows to the DIALOG instruction. It is defined by the number of elements in the array. However, when using the paged mode in a DISPLAY ARRAY, the total number of rows does not correspond to the elements in the program array, because the program array holds only a page of the whole list. In any other case, a call to this method is just ignored.

**Note:** A call to setArrayLength() will not trigger the execution of the ON FILL BUFFER clause immediately: This trigger will be executed when the control goes back to the dialog instruction, after all user code following setArrayLength() has been executed.

The setArrayLength() method is also used to fix the final number of rows when using COUNT=-1 attribute. When implementing a paged list without knowing the total number of rows when the dialog starts.

**Note:** In a paged mode DISPLAY ARRAY using COUNT=-1, before calling setCurrentRow(screen-array, row-index), be sure to provide the actual number of rows with ui.Dialog.setArrayLength(screen-array, count) where count > row-index. Otherwise, the setCurrentRow() call will have no effect, if the dialog has not yet seen row-index rows through ON FILL BUFFER.

Related concepts
Populating a DISPLAY ARRAY on page 1841
The program array must be filled with rows to populate the DISPLAY ARRAY dialog.

Paged mode of DISPLAY ARRAY on page 1843
In order to handle very large result sets, use the paged mode of DISPLAY ARRAY.

ON FILL BUFFER block on page 1508

ui.Dialog.setCellAttributes
Define cell decoration attributes array for the specified list (singular dialog only).

Syntax

```plaintext
setCellAttributes(
    attributes dynamic-array-type )
```

1. `attributes` is a program array defining the cell attributes.
2. `dynamic-array-type` is a DYNAMIC ARRAY OF ... type, which can be:
   - A DYNAMIC ARRAY OF RECORD ... END RECORD (with the same structure as the data array)
   - A DYNAMIC ARRAY OF STRING (to define attributes for complete lines instead of individual cells)
   - A DYNAMIC ARRAY WITH DIMENSION 2 OF STRING (to define attributes in dynamic dialog when the row structure is defined at runtime)

Usage

In an INPUT ARRAY or DISPLAY ARRAY dialog, the `setCellAttributes()` method can be used to specify display attributes for each cell, or for the complete row.

The `setCellAttributes()` method is designed for dialog programming, where only one screen array is used (for example, in a singular DISPLAY ARRAY dialog). An equivalent method called `setArrayAttributes()` can be used, when several screen arrays are defined in a multiple dialog, to be able to identify the list by the name of the screen array.

Possible values for cell attributes are a combination of the following values:

- One of the supported color names, or an #RRGGBB value.
- The `blink` attribute: The text in the cell blinks.
- The `bold` attribute: The text in the cell renders with a bold font.
- The `underline` attribute: The text in the cell is underlined.
- The `reverse` attribute: When a color is specified, it is used as background color instead of foreground text color.

The cell attributes must be specified in lowercase, the color names must be in camelCase, and the separator is a blank space, for example:

**Note:** For the predefined (TTY) color names (like blue, red, yellow), the actual rendering color used by the front-ends is adapted to the reverse or normal mode: In normal mode, colors like "yellow" must be darker, to make the foreground text easily readable on an white background. In reverse mode (where the color specification applies to the background), the actual background color must be lighter, so that the black foreground text can be read more easily.

- "green": Foreground text color is green.
- "lightRed reverse": Background cell color is light red.
- "blue underline": Foreground text color is blue and the text font is underlined.
- "red bold reverse": Background cell color is red and the text font is bold.

The structure of the dynamic array containing the cell attributes can be:

- A DYNAMIC ARRAY OF RECORD, with the same structure as the data array
- A DYNAMIC ARRAY WITH DIMENSION 2 OF STRING, to define a flexible set of cell attributes (for dynamic dialogs)
- A DYNAMIC ARRAY OF STRING, to define attributes for complete lines
The following example defines a dynamic array with the same structure as the data array. Note however that the members of the attributes array use the STRING data type:

```
DEFINE data DYNAMIC ARRAY OF RECORD
  pkey INTEGER,
  name VARCHAR(50)
END RECORD
DEFINE attributes DYNAMIC ARRAY OF RECORD
  pkey STRING,
  name STRING
END RECORD
```

Similarly, the cell attributes array can be defined with a two-dimensional dynamic array of strings:

```
DEFINE attributes DYNAMIC ARRAY WITH DIMENSION 2 OF STRING
```

The advantage of a two-dimensional array is the flexibility, as it can define an unlimited number of cells for each row. This solution is typically used when implementing a dynamic dialog.

Finally, if you want to decorate complete lines instead of individual cells, use a simple dynamic array of strings:

```
DEFINE attributes DYNAMIC ARRAY OF STRING
```

Fill the display attributes array with color and video attributes:

```
FOR i=1 TO data.getLength()  -- length from data array!
  LET attributes[i].name = "blue reverse"
END FOR
```

Then, attach the array to the dialog with the `setCellAttributes()` method, in a BEFORE INPUT or BEFORE DISPLAY block:

```
BEFORE DISPLAY
  CALL DIALOG.setCellAttributes( attributes )
```

Like data values, if you change the cell attributes during the dialog, these are not displayed automatically unless the UNBUFFERED mode is used:

```
ON ACTION modify_cell_attribute
  LET attributes[arr_curr()].name = "red reverse"
```

If you set NULL to an element, the default TTY attributes will be reset:

```
ON ACTION clean_cell_attribute
  LET attributes[arr_curr()].name = NULL
```

**Related concepts**

- Cell color attributes on page 1850
  List controllers can display every cell in a specific color.
- `ui.Dialog.setArrayAttributes` on page 2543
  Define cell decoration attributes array for the specified list (singular or multiple dialogs).
- Example 4: Set display attributes for cells on page 2560
Define autocompletion items for a field defined with COMPLETER attribute.

**Syntax**

```java
setCompleterItems(
    items DYNAMIC ARRAY OF STRING )
```

1. `items` defines the list of completion proposals to be passed to the front-end.

**Usage**

The `setCompleterItems()` dialog method defines the list of completion proposals for the current field, to implement autocompletion.

The field must be defined in the form with the `COMPLETER` attribute.

The list of completion proposal items is passed as a dynamic array of strings:

```java
DEFINE items DYNAMIC ARRAY OF STRING
```

To clean up the completion proposal list for a given field, pass `NULL` as second parameter to the function.

**Important:** The method will raise error `-8114`, if the list of items contains more than 50 elements. Note that this error is not trappable with exception handlers like `TRY/CATCH`, the code must avoid reaching the limit.

See [Enabling autocompletion](#) on page 1733 for more details.

**Example**

```java
DEFINE items DYNAMIC ARRAY OF STRING
...
ON CHANGE firstname
    -- fill the array with items
    LET items[1] = "Ann"
    LET items[2] = "Anna"
    LET items[3] = "Annabel"
    CALL DIALOG.setCompleterItems(items)
```

**Related concepts**

- [Enabling autocompletion](#) on page 1733
- [COMPLETER attribute](#) on page 1360

Autocompletion allows a list of completion proposals to be displayed while the user is typing text into a field.

The `COMPLETER` attribute enables autocompletion for the edit field.

**ui.Dialog.setCurrentRow**

Sets the current row in the specified list.

**Syntax**

```java
setCurrentRow(
    name STRING,
    row INTEGER )
```

1. `name` is the name of the screen record, see [Identifying screen-arrays in ui.Dialog methods](#) on page 2557.
2. `row` is the new row in the array.
Usage

Use the `setCurrentRow()` method to change the current row in an `INPUT ARRAY` or `DISPLAY ARRAY` list. You must pass the name of the screen array to identify the list, and the new row number.

```
DEFINE x INTEGER
DIALOG
  DISPLAY ARRAY custlist TO sa_custlist.*
  ... 
END DISPLAY
ON ACTION goto_x
  CALL DIALOG.setCurrentRow("sa_custlist", x)
  ...
```

Moving to a different row with `setCurrentRow()` will not trigger control blocks such as `BEFORE ROW` / `AFTER ROW`, as the `fgl_set_arr_curr()` built-in function does.

The `setCurrentRow()` method will not set the focus; you need to use `NEXT FIELD` to set the focus to a list. (This works with `DISPLAY ARRAY` as well as with `INPUT ARRAY`.)

If the passed row index is lower than 1, the first row will be selected. If the row index is greater than the total number of rows, the last row will be selected.

If the new current row is not in the current view, the dialog will adapt the list offset to make the new current row visible.

If `multi-row selection` is enabled, all selection flags of rows are cleared, and the new current row gets automatically selected.

A `setCurrentRow(screen-array, row-index)` in a `paged mode DISPLAY ARRAY` will subsequently trigger the `ON FILL BUFFER` code, if the requested row is not in the current page of visible rows.

If `DISPLAY ARRAY` using `ON FILL BUFFER` was started with `COUNT=-1`, and the `row-index` provided by `setCurrentRow()` is greater as the actual number of rows already fetched through `ON FILL BUFFER`, `setCurrentRow()` will have no effect. The actual number of rows must be provided with `DIALOG.setArrayLength(screen-array, count)` where `count >= row-index`, before calling `setCurrentRow()`:

```
DISPLAY ARRAY ... ATTRIBUTES(COUNT=-1)
ON FILL BUFFER
  ...
ON ACTION goto_last_row
  SELECT COUNT(*) INTO cnt FROM ... -- Count total rows
  CALL DIALOG.setArrayLength("sr", cnt)
  CALL DIALOG.setCurrentRow("sr", cnt)
  ...
```

`ui.Dialog.setFieldActive`

Enable and disable form fields.

Syntax

```
setFieldActive(
  fields STRING,
  val BOOLEAN )
```

1. `fields` is the string with the list of field specification, see Identifying fields in `ui.Dialog` methods on page 2556.
2. `val` is a boolean value.

Usage

The `setFieldActive()` method can be used to enable / disable form fields.
The fields is a string containing the field qualifier, with an optional prefix ("[table.]column"), or a table prefix followed by a dot and an asterisk ("table.*").

```plaintext
CALL DIALOG.setFieldActive( "customer.cust_addr", (rec.cust_name IS NOT NULL) )
```

Do not disable all fields of a dialog, otherwise the dialog execution stops (at least one field must get the focus during a dialog execution).

When disabling the current field, the AFTER FIELD block of the current field and the BEFORE FIELD block on the next field in the tabbing order will be executed. As a general pattern, do not disable the current field having the focus.

**ui.Dialog.setFieldTouched**

Sets the modification flag of the specified field.

**Syntax**

```plaintext
setFieldTouched(
  fields STRING,
  val BOOLEAN )
```

1. fields is the string with the list of field specification, see Identifying fields in ui.Dialog methods on page 2556.
2. val is the boolean value to set the modification flag.

**Usage**

The setFieldTouched() method can be used to change the modification flag of the specified field(s).

The fields is a string containing the field qualifier, with an optional prefix ("[table.]column"), or a table prefix followed by a dot and an asterisk ("table.*").

You typically use this method to set the touched flag when assigning a variable, to emulate user input. Remember when using the UNBUFFERED mode, you don't need to display the value to the fields. The setFieldTouched() method is provided as a 3GL replacement for the DISPLAY BY NAME / TO instructions to set the modification flags.

```plaintext
ON ACTION zoom_city
  LET p_cust.cust_city = zoom_city()
  CALL DIALOG.setFieldTouched("customer.cust_city", TRUE)
  ...
```

If the parameter is a screen record followed by dot-asterisk, the method checks the modification flags of all the fields that belong to the screen record. You typically use this to reset the touched flags of a group of fields, after modifications have been saved to the database, to get back to the initial state of the dialog:

```plaintext
ON ACTION save
  CALL save_cust_record()
  CALL DIALOG.setFieldTouched("customer.*", FALSE)
  ...
```

The modification flags are reset to false when using an INPUT ARRAY list, every time you leave the modified row.

**ui.Dialog.setFieldValue**

Sets the value of a field controlled by the dialog object.

**Syntax**

```plaintext
setFieldValue(
  name STRING,
  value ANY )
```
value fgl-type )

1. **name** is the name of the field, see Identifying fields in ui.Dialog methods on page 2556.
2. **value** is the value to be set, where fgl-type is one of the primitive data types.

**Usage**

The `setFieldValue()` method can be used when implementing a dynamic dialog, to set the value of a field:

```pascal
DEFINE default_address STRING,
    default_creadate DATE
...
CALL d.setFieldValue( "customer.cust_addr", default_address )
CALL d.setFieldValue( "customer.cust_creadate", default_creadate )
```

The first parameter defines the field to be set.

In a dynamic dialog controlling a list of records (INPUT ARRAY / DISPLAY ARRAY), this method sets the value for a field in the current row. To fill the list of records before dynamic dialog execution, use `setCurrentRow()` to set the current row, then set field (cell values) with `setFieldValue()`.

**Important:**

- During dialog execution, the `setFieldValue()` method must only be used to set the value of a field for the current row. Calling the `setCurrentRow()` method to change the current row before calling `setFieldValue()` will have no effect.
- In a DISPLAY ARRAY using the paged mode (ON FILL BUFFER), `setFieldValue()` will produce error -8129, if the current row is not part of the visible page. In a DISPLAY ARRAY using the full list mode, the error -8129 is raised, if there is no current row (when the array is empty).

**Example**

The following code example implements a `FOR` loop to copy values of all fields of the `d_disparr` dialog to the field of the `d_recinp` dialog:

```pascal
DEFINE row, i INTEGER,
    h base.SqlHandle,
    fields DYNAMIC ARRAY OF RECORD
        name STRING,
        type STRING
    END RECORD,
    d_rec ui.Dialog,
    d_list ui.Dialog
...
-- Fill the array with rows from an SqlHandle object
CALL h.open()
LET row = 0
WHILE status == 0
    -- must set the current row before setting values
    CALL d_list.setCurrentRow("sr_custlist", row:=row+1 )
    FOR i = 1 TO h.getResultCount()
        CALL d_list.setFieldValue( h.getResultName(i),
            h.getResultValue(i) )
    END FOR
    h.getResultSetValue(i )
END FOR
CALL h.fetch()
END WHILE
CALL d_list.setCurrentRow("sr_custlist", 1)
...
-- Copy field values from d_list to d_rec dialog
FOR i=1 TO fields.getLength()
    CALL d_rec.setFieldValue( fields[i].name,
        d_list.getFieldValue( fields[i].name )
    )
```
Related concepts

ui.Dialog.getFieldValue on page 2533
Returns the value of a field controlled by a dynamic dialog.

Dynamic Dialogs on page 2004
Dialogs can be created at runtime with the ui.Dialog class.

ui.Dialog.setSelectionMode
Defines the row selection mode for the specified list.

Syntax

```java
setSelectionMode(
    name  STRING,
    mode  INTEGER )
```

1. `name` is the name of the screen record, see Identifying screen-arrays in ui.Dialog methods on page 2557.
2. `mode` defines the selection mode (0, 1).

Usage

In DISPLAY ARRAY instructions, the setSelectionMode() method can be used to enable/disable multi-row selection.

Possible values of the `mode` parameter are 0 (single row selection) or 1 (multi-range selection). Other values are reserved for future use.

If multi-row selection is switched off, selected rows get deselected.

For more details about multi-row selection, see Multiple row selection on page 1851.

ui.Dialog.setSelectionRange
Sets the row selection flags for a range of rows.

Syntax

```java
setSelectionRange(
    name  STRING,
    start INTEGER,
    end   INTEGER,
    value BOOLEAN )
```

1. `name` is the name of the screen record, see Identifying screen-arrays in ui.Dialog methods on page 2557.
2. `start` is the starting row index.
3. `end` is the ending row index.
4. `value` is the selection flag to set.

Usage

If multi-row selection is enabled with `setSelectionMode()`, you can set the selection flags for a range of rows with the `setSelectionRange()` method.

```java
ON ACTION select_all
    CALL DIALOG.setSelectionRange( "sr", 1, -1, TRUE)
```

The start and end index must be in the range of possible row indexes (from 1 to DIALOG.getArrayLength()).

If you specify an end index of -1, it will set the flags from start index to the end of the list.
ui.Dialog.validate
Checks form level validation rules.

Syntax

validate(
  formFieldList )  
RETURNS INTEGER

1. formFieldList is a [ ] variable list of strings defining the fields to be validated. The [ ] square brackets are optional, if only one element is provided. See also Identifying fields in ui.Dialog methods on page 2556.

Usage

Use the validate() method in order to execute NOT NULL, REQUIRED and INCLUDE validation rules defined in the form specification files.

The method takes a variable list parameter with the [ ] square brace notation, containing a comma-separated list of fields or screen records. The [ ] square brackets are optional, if only one element is to be validated:

DIALOG.validate("customer.cust_name")
DIALOG.validate("cust_rec.*")
DIALOG.validate( [ "cust_rec.*", "ord_rec.*" ] )

There are different notations to identify form fields in a dialog. For details, see Identifying fields in ui.Dialog methods on page 2556.

The method returns zero if success, or the input error code of the first field which does not satisfy the validation rules.

Note: The current field is always checked, even if it is not part of the validation field list. This is mandatory, otherwise the current field may be left with invalid data.

If an error occurs, the validate() method automatically displays the corresponding error message, and registers the next field to jump to when the interactive instruction gets the control back.

The validate() method does not stop code execution if an error is detected. You must execute a CONTINUE DIALOG or CONTINUE INPUT instruction to cancel the code execution.

A typical usage is for a "save" action:

ON ACTION save
  IF DIALOG.validate("cust_rec.*") < 0 THEN
    CONTINUE DIALOG
  END IF
  CALL customer_save()

Related concepts

ui.Dialog.accept on page 2520
Validates and terminates the dialog.

ui.Dialog.cancel on page 2527
Cancels a parent dialog from a sub-dialog.

ui.Dialog.visualToArrayIndex
Converts the visual index to the program array index for a given screen array.

Syntax

visualToArrayIndex(
  name STRING,  
  visualIndex INTEGER )
1. *name* is the name of the screen record, see Identifying screen-arrays in ui.Dialog methods on page 2557.
2. *visualIndex* is the index of the row as seen by the end user.

**Usage**

When the end user sorts rows in a table, the visual row index may differ from the program array index.

Use this method to convert a row index as seen by the end user, to the program array index. For example, if the application implements a feature that allows the user to enter a row index to jump to that row, it will be entered as a visual row index. You must convert this index to the program array index, for example to make a `setCurrentRow()`.

```plaintext
CALL DIALOG.setCurrentRow("sr", DIALOG.visualToArrayIndex("sr", user_index))
```

**Related concepts**

- Handling the current row on page 1835
  Query and control the current row in a read-only or editable list of records.
- `ui.Dialog.arrayToVisualIndex` on page 2527
  Converts the program array index to the visual index for a given screen array.

**Usage**

**Referencing the current dialog**

In order to reference the current dialog, you can define a variable with the `ui.Dialog` type, and get the current dialog object with the `ui.Dialog.getCurrent()` method:

```plaintext
DEFINE d ui.Dialog

INPUT BY NAME ...
  BEFORE DIALOG
    LET d = ui.Dialog.getCurrent()
    CALL d.setActionActive("zoom", FALSE)
  ...
```

As an alternative and to simplify programming, it is recommended that you use the `DIALOG` keyword in the context of the interactive instruction block. The `DIALOG` keyword is a predefined object variable referencing the current dialog. The `DIALOG` variable can only be used inside the interactive instruction block:

```plaintext
INPUT BY NAME custid, custname
  ON ACTION disable
    CALL DIALOG.setFieldActive("custid", FALSE)
END INPUT
```

**Related concepts**

- Passing a dialog reference to functions on page 2554

**Passing a dialog reference to functions**

Using the `DIALOG` keyword outside a dialog instruction block results in a compilation error. However, you can pass the object to a function that defines the dialog parameter with the `ui.Dialog` type.

The following example passes the `DIALOG` object reference to the `setupDialog()` function, which implements action activation rules that must be applied after different events, during the dialog execution:

```plaintext
INPUT BY NAME custid, custname, custaddr
BEFORE INPUT
  CALL setupDialog(DIALOG)
  ...
ON ACTION check_address
  ...
```
CALL setupDialog(DIALOG)

...  
END INPUT

FUNCTION setupDialog(d)
  DEFINE d ui.Dialog
  DEFINE isAdmin BOOLEAN
  LET isAdmin = (global_params.user_group == "admin")
  CALL d.setActionActive("delete", isAdmin)
  CALL d.setActionActive("convert", isAdmin)
  CALL d.setActionActive("check_address", isAdmin AND rec.custaddr IS NOT NULL)
END FUNCTION

Related concepts

Referencing the current dialog on page 2554

Identifying actions in ui.Dialog methods

In ui.Dialog methods such as setActionActive(), the first parameter identifies the action object to be modified. This parameter can be fully-qualified or partly-qualified. If you don't specify a fully-qualified name, the action object will be identified based on the focus context.

The action name specification can be any of the following:

- action-name
- dialog-name.action-name
- dialog-name.field-name.action-name
- field-name.action-name (singular dialogs only)

Here action-name identifies the name of the action specified in ON ACTION action-name or COMMAND "action-name" handlers, while dialog-name identifies the singular dialog or sub-dialog and field-name defines the field bound to the action INFIELD clause of ON ACTION.

The runtime system will raise the error -8089 if the action specified by [dialog-name.[field-name.]action-name can not be found within the current dialog.

As a general rule, assign unique action names for each specific dialog action, to avoid the usage of dialog and/or field identifiers.

The name of the action passed as parameter can use the same letter case as the action definition: The lookup is case-insensitive:

ON ACTION PrintReport

...  
CALL DIALOG.setActionActive("PrintReport", FALSE)

In the DIALOG instruction, actions can be prefixed with the sub-dialog identifier. However, if methods like setActionActive() are called in the context of the sub-dialog, the prefix can be omitted. When using a field-specific action defined with the INFIELD clause of ON ACTION, you can identify the action with the fully-qualified name dialog-name.field-name.action-name. Like sub-dialog actions, if you specify only action-name, the runtime system will search for the action object based on the focus context.

Note that an INPUT or CONSTRUCT sub-dialogs have no identifier by default. The dialog name can be defined with the NAME attribute. For more details, see Identifying sub-dialogs in procedural DIALOG on page 1594.

When using a singular dialog like INPUT, you can identify field-specific actions by field-name.action-name, if the dialog was defined without a NAME attribute.

Example

MAIN
  DEFINE cust_rec RECORD
num INT,
name VARCHAR(50),
city INT
END RECORD
DEFINE orders DYNAMIC ARRAY OF RECORD
ord_num INT,
cust_num INT,
ord_date DATE
END RECORD
OPEN FORM f1 FROM "cust_ord"
DISPLAY FORM f1
DIALOG ATTRIBUTES(UNBUFFERED)
INPUT BY NAME cust_rec.* ATTRIBUTES(NAME="cust")
  ON ACTION compare
    CALL compare()
  ON ACTION check INFIELD cust_city
    CALL check_city(cust_rec.city)
END INPUT
DISPLAY ARRAY orders TO sr_ord.*
  ON ACTION archive
    CALL archive()
END DISPLAY
ON ACTION print
    CALL print()
ON ACTION disable_all
    CALL DIALOG.setActionActive("cust.compare", FALSE)
    CALL DIALOG.setActionActive("cust.cust_city.check", FALSE)
    CALL DIALOG.setActionActive("sr_ord.archive", FALSE)
    CALL DIALOG.setActionActive("print", FALSE)
END DIALOG
END MAIN

Related concepts

Enabling and disabling actions on page 1763
By default, dialog actions are enabled. However, it is recommended that an action be disabled when not allowed in the current context.

Binding action views to action handlers on page 1762
How are action views of the forms bound to action handlers in the program code?

Identifying fields in ui.Dialog methods

In ui.Dialog methods such as setFieldActive(), the first parameter identifies the form field (or, for some methods, a list of fields) to be modified. The form field names can be fully-qualified or partly-qualified.

Fields are identified by the form field name specification (in the .per form), not the program variable name used by the dialog. Form fields are bound to program variables with the binding clause of the dialog instruction (INPUT variable-list FROM field-list.ATTRIBUTES(NAME=variable-list).CONSTRUCT BY NAME sql ON column-list.CONSTRUCT sql ON column-list FROM field-list,INPUT ARRAY array-name FROM screen-array.*).

The field name specification can be any of the following:

- field-name
- table-name.field-name
- screen-record-name.field-name
- FORMONLY.field-name

Here are some examples:
- "cust_name",
- "customer.cust_name",
- "cust_screen_record.cust_name",
• "item_screen_array.item_label",
• "formonly.total",
• "customer.*" (only some methods accept the "dot asterisk" notation)

When no field name prefix is used, the first form field matching that field name will be used. If the field specification is invalid (no field in the current dialog matches the field specification), the method will throw the error -1373.

When using a prefix in the field name specification, it must be equal to the field prefix used in the field binding clause of the dialog.

• When no screen-record was specified in the field binding clause (for example, when using INPUT BY NAME variable-list), the field prefix must be the database table name (or FORMONLY) used in the form file, or any valid screen-record using that field.
• When the FROM clause of the dialog specifies an explicit screen-record (for example, in INPUT variable-list FROM screen-record.* / field-list-with-screen-record-prefix or INPUT ARRAY array-name FROM screen-array.*) the field prefix must be the screen-record name used in the FROM clause.

The name of the field passed as parameter can use the same letter case as the field definition in the form file: The lookup is case-insensitive:

```
-- In the form file:
EDIT f01 = Customer.CustAddr, ... ;
-- In the program code:
CALL DIALOG.setFieldActive("Customer.CustAddr", FALSE)
```

The methods validate(), setFieldActive(), setFieldTouched(), getFieldTouched() can take a list of fields as parameter, by using the "dot-asterisk " notation (screen-record.*). This way you can check, query or change a complete list of fields in one method call:

```
ON ACTION save
    CALL save_cust_record()
    CALL DIALOG.setFieldTouched("customer.*", FALSE)
    ...
```

**Identifying screen-arrays in ui.Dialog methods**

In ui.Dialog methods such as setCurrentRow(), the first parameter is the name of the screen array to identify the list container in the form.

Screen arrays are defined in form specification files with the SCREEN RECORD clause, and used in DISPLAY ARRAY and INPUT ARRAY instructions to bind program array variables to the list container.

In the form file:

```
LAYOUT
...
INSTRUCTIONS
SCREEN RECORD custlist ( ... );
END
```

In the program code:

```
DISPLAY ARRAY custarr TO custlist.*
    ...
    ON ACTION set_row
        CALL DIALOG.setCurrentRow("custlist", row_index)
    ...
```
The name of the screen array passed as parameter can use the same letter case as in the SCREEN RECORD definition:

The lookup is case-insensitive:

```plaintext
-- In the form file:
SCREEN RECORD CustList ( ... );
-- In the program code:
CALL DIALOG.setCurrentRow("CustList", row_index)
```

**Related concepts**

- Screen records / arrays on page 1251
- Form fields can be grouped in a screen record or screen array definition.
- Variable binding in DISPLAY ARRAY on page 1504
- Variable binding in INPUT ARRAY on page 1535

**Field definition for Dynamic Dialogs**

Dynamic dialog creation methods require field definitions in a dynamic array with a predefined structure.

Dynamic dialog creation methods such as `ui.Dialog.createInputByName()` require a dynamic array as parameter, to define the list of fields that the dialog will control.

This parameter must be defined as a DYNAMIC ARRAY OF RECORD, with name and type members declared as STRING:

```plaintext
DEFINE fields DYNAMIC ARRAY OF RECORD
    name STRING,
    type STRING
END RECORD
```

The names provided in the field definition list must identify form fields of the current form.

For example, if the current form file defines the following fields:

```plaintext
LAYOUT
...
END
TABLES
    customer
END
ATTRIBUTES
    EDIT f1 = customer.cust_id;
    EDIT f2 = customer.cust_name;
    ...
END
```

The field names provided in the dynamic array must be:

```plaintext
LET fields[1].name = "customer.cust_id"
LET fields[2].name = "customer.cust_name"
...
```

The types provided in the field definition list will identify the data type to be used for data input and display.

Possible values for types are the string equivalents of the Genero BDL built-in types, for example:

- "INTEGER"
- "VARCHAR(50)"
- "DATE"
- "DECIMAL(10,2)"
- "DATETIME YEAR TO FRACTION(5)"
Note: The type used to define form fields can be the returning value of a `base.SqlHandle.getResultType()` method.

For example:

```plaintext
DEFINE fields DYNAMIC ARRAY OF RECORD
    name STRING,
    type STRING
END RECORD
DEFINE d ui.Dialog
OPEN FORM f1 FROM "custform"
DISPLAY FORM f1
LET fields[1].name = "customer.cust_id"
LET fields[1].type = "INTEGER"

LET fields[2].name = "customer.cust_name"
LET fields[2].type = "VARCHAR(50)"
...
LET d = ui.Dialog.createInputByName(fields)
...
```

Related concepts

**Dynamic Dialogs** on page 2004
Dialogs can be created at runtime with the `ui.Dialog` class.

Examples

**ui.Dialog usage examples.**

**Example 1: Disable fields dynamically**

```plaintext
FUNCTION input_customer()
    DEFINE custid INTEGER
    DEFINE custname CHAR(10)
    INPUT BY NAME custid, custname
    ON ACTION enable
        CALL DIALOG.setFieldActive("custid",1)
    ON ACTION disable
        CALL DIALOG.setFieldActive("custid",0)
    END INPUT
END FUNCTION
```

**Example 2: Get the form and hide fields**

```plaintext
FUNCTION input_customer()
    DEFINE f ui.Form
    DEFINE custid INTEGER
    DEFINE custname CHAR(10)
    INPUT BY NAME custid, custname
    BEFORE INPUT
        LET f = DIALOG.getForm()
        CALL f.setElementHidden("customer.custid",1)
    END INPUT
END FUNCTION
```

**Example 3: Pass a dialog object to a function**

```plaintext
DEFINE r_user RECORD
    can_print BOOLEAN,
    can_query BOOLEAN
END RECORD
```
FUNCTION input_customer()
    DEFINE custid INTEGER
    DEFINE custname CHAR(10)
    INPUT BY NAME custid, custname
    BEFORE INPUT
        CALL setup_dialog(DIALOG)
    END INPUT
END FUNCTION

FUNCTION setup_dialog(d)
    DEFINE d ui.Dialog
    CALL d.setActionActive("print", r_user.can_print)
    CALL d.setActionActive("query", r_user.can_query)
END FUNCTION

Example 4: Set display attributes for cells

Note: This code example uses a DYNAMIC ARRAY with the same RECORD structure as the data array. You could also use a two-dimensional dynamic array or a simple flat array of strings. For more details, see `ui.Dialog.setArrayAttributes` on page 2543

FUNCTION display_items()
    DEFINE i INTEGER
    DEFINE items DYNAMIC ARRAY OF RECORD
        key INTEGER,
        name CHAR(10)
    END RECORD
    DEFINE attributes DYNAMIC ARRAY OF RECORD
        key STRING,
        name STRING
    END RECORD
    FOR i=1 TO 10
        CALL items.appendElement()
        LET items[i].key = i
        LET items[i].name = "name " || i
        CALL attributes.appendElement()
        IF i MOD 2 = 0 THEN
            LET attributes[i].key = "red"
            LET attributes[i].name = "blue reverse"
        ELSE
            LET attributes[i].key = "green"
            LET attributes[i].name = "magenta reverse"
        END IF
    END FOR
    DISPLAY ARRAY items TO sr.* ATTRIBUTES(UNBUFFERED)
    BEFORE DISPLAY
        CALL DIALOG.setCellAttributes(attributes)
    ON ACTION att_modify_cell
        LET attributes[2].key = "red reverse"
    ON ACTION att_clear_cell
        LET attributes[2].key = NULL
    END DISPLAY
END FUNCTION

END FUNCTION
**The ComboBox class**
The `ui.ComboBox` class provides an interface to the COMBOBOX form field view in the abstract user interface tree.

In .per form specification files, a COMBOBOX form field defines both a form field and a view for that model. The `ui.ComboBox` class is an interface to the view of a COMBOBOX form field. It is typically used to configure the widget dynamically in programs, for example to create the list of items shown in the drop down box.

**Related concepts**
- Filling a COMBOBOX item list on page 1735
  
The item list of COMBOBOX fields can be initialized at runtime.

**ui.ComboBox methods**
Methods of the `ui.ComboBox` class.

**Table 535: Class methods**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ui.ComboBox.forName(name STRING)</code></td>
<td>Search for a combobox in the current form.</td>
</tr>
<tr>
<td></td>
<td>RETURNS <code>ui.ComboBox</code></td>
</tr>
<tr>
<td><code>ui.ComboBox.setDefaultInitializer(initializer STRING)</code></td>
<td>Define the default initializer for combobox form items.</td>
</tr>
</tbody>
</table>

### Table 536: Object methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addItem (code STRING, text STRING)</code></td>
<td>Add an element to the item list.</td>
</tr>
<tr>
<td><code>clear ()</code></td>
<td>Clear the item list of a combobox.</td>
</tr>
<tr>
<td><code>getColumnName ()</code></td>
<td>Get the column name of the form field.</td>
</tr>
<tr>
<td><code>getIndexOf (code STRING)</code></td>
<td>Get an item position by name.</td>
</tr>
<tr>
<td><code>getItemCount ()</code></td>
<td>Get the number of items.</td>
</tr>
<tr>
<td><code>getItemName (index INTEGER)</code></td>
<td>Get an item name by position.</td>
</tr>
<tr>
<td><code>getItemText (index INTEGER)</code></td>
<td>Get the item text by position.</td>
</tr>
<tr>
<td><code>getTableName ()</code></td>
<td>Get the table prefix of the form field.</td>
</tr>
<tr>
<td><code>getTag ()</code></td>
<td>Get the combobox tag value.</td>
</tr>
<tr>
<td><code>getTextOf (code STRING)</code></td>
<td>Get the item text by name.</td>
</tr>
<tr>
<td><code>removeItem (code STRING)</code></td>
<td>Remove an item by name.</td>
</tr>
</tbody>
</table>

**ui.ComboBox.setDefaultInitializer**

Define the default initializer for combobox form items.

**Syntax**

```java
ui.ComboBox.setDefaultInitializer()
```
initializer STRING )

1. initializer is the name of the initialization function.

Usage

The ui.ComboBox.setDefaultInitializer() class method specifies a default initialization function to be called each time a COMBOBOX form field is created when loading forms.

Use this method to define a global/default initialization function for all comboboxes of the program. For individual comboboxes, consider using the INITIALIZER form field attribute instead.

Important: The initialization function name is case insensitive.

The initialization function is called with the ui.ComboBox object as the parameter.

The combobox initialization functions typically fill the drop down list of COMBOBOX fields with items.

For more details about combobox initializer programming, see Filling a COMBOBOX item list on page 1735.

Example

The form file form1.per:

```
LAYOUT
GRID
{ [cb1] [cb2] [cb3] }
END
END
ATTRIBUTES
COMBOBOX cb1 = FORMONLY.city1, NOT NULL; -- Gets default initializer
COMBOBOX cb2 = FORMONLY.city2, NOT NULL; -- Gets default initializer
COMBOBOX cb3 = FORMONLY.city3, NOT NULL, INITIALIZER=cb_init_2;
END

The main module:

```
IMPORT FGL setup
MAIN
DEFINE rec RECORD city1, city2, city3 STRING END RECORD
CALL setup.init_combo_setup(TRUE)
CALL ui.ComboBox.setDefaultInitializer("cb_init_1")
OPEN FORM f1 FROM "form1"
DISPLAY FORM f1 -- initialization function is called
INPUT BY NAME rec.*
END
```

The imported module setup.4gl:

```
PRIVATE DEFINE with_undef BOOLEAN

PUBLIC FUNCTION init_combo_setup(wu)
DEFINE wu BOOLEAN
LET with_undef = wu
END FUNCTION

PUBLIC FUNCTION cb_init_1(cb)
DEFINE cb ui.ComboBox
CALL cb.clear()
IF with_undef THEN
```
CALL cb.addItem(0,"<undef>")
END IF
CALL cb.addItem(0,"London")
CALL cb.addItem(0,"Paris")
CALL cb.addItem(0,"Rome")
CALL cb.addItem(0,"Copenhague")
END FUNCTION

PUBLIC FUNCTION cb_init_2(cb)
DEFINE cb ui.ComboBox
CALL cb.clear()
IF with_undef THEN
    CALL cb.addItem(0,"<undef>")
END IF
CALL cb.addItem(0,"Berlin")
CALL cb.addItem(0,"Berne")
CALL cb.addItem(0,"Madrid")
CALL cb.addItem(0,"Vienne")
END FUNCTION

**ui.ComboBox.forName**
Search for a combobox in the current form.

**Syntax**

```
ui.ComboBox.forName (    
    name STRING    )    
RETURNS ui.ComboBox
```

1. **name** is the name of COMBOBOX form item.

**Usage**

The `ui.ComboBox.forName()` class method searches for a `ui.ComboBox` object by form field name in the current form.

The name of the combobox passed as parameter can use the same letter case as that in the form definition file: The lookup is case-insensitive.

After loading a form with `OPEN WINDOW WITH FORM`, use the class method to retrieve a `ui.ComboBox` object into a variable defined as a `ui.ComboBox`.

```
DEFINE cb ui.ComboBox
LET cb = ui.ComboBox.forName("formonly.airport")
```

Verify the function has returned an object, as the form field may not exist.

```
IF cb IS NULL THEN
    ERROR "Form field not found in current form"
    EXIT PROGRAM
END IF
```

Once instantiated, the `ui.ComboBox` object can be used, for example to fill the items of the drop down list.

```
CALL cb.clear()
CALL cb.addItem(1,"Paris")
CALL cb.addItem(2,"London")
CALL cb.addItem(3,"Madrid")
```

**Related concepts**

Example Get a ComboBox form field view and fill the item list on page 2569
ui.ComboBox.addItem
Add an element to the item list.

Syntax

```java
addItem(
    code STRING,
    text STRING )
```

1. `code` is the unique key that identifies the item.
2. `text` is the text to be displayed in the drop down list.

Usage

The `addItem()` method adds an item to the end of the drop down list of the COMBOBOX.

The first parameter is the value that can be set in the form field. The second parameter is the label to be displayed in the drop down list. If the second parameter is `NULL`, the runtime system automatically uses the first parameter as the display value.

Uniqueness is not checked by the runtime system. Make sure that the items created are unique, regarding the value key and the display label.

Trailing spaces are not recommended when populating the first parameter because values get truncated when field validation occurs, and the resulting value (without trailing spaces) will no longer match the COMBOBOX item name. Additionally, trailing spaces in the second parameter may cause the COMBOBOX to be much wider than expected. To avoid such problems, use VARCHAR or STRING variables, or use the CLIPPED operator with CHAR variables.

Related concepts

Example Using the INITIALIZER attribute in the form file on page 2569

ui.ComboBox.clear
Clear the item list of a combobox.

Syntax

```java
clear()
```

Usage

The `clear()` method clears the item list of the combobox.

If the item list is empty, the COMBOBOX drop-down button shows an empty list on the client side.

ui.ComboBox.getColumnName
Get the column name of the form field.

Syntax

```java
getColumnName ()
RETURNS STRING
```

Usage

The `getColumnName()` method returns the form field column name. The form field column name can be `NULL` if not defined at the form field level.
Use the `getTableName()` and `getColumnName()` methods together in order to identify the form field associated with the COMBOBOX. This allows for the identification of the combobox field in your program, for example to fill the drop down list with the appropriate items.

```plaintext
IF cb.getTableName() || cb.getColumnName() == "customer.cust_city" THEN
    CALL cb.clear()
    CALL cb.addItem(1, "Paris")
    CALL cb.addItem(2, "London")
    CALL cb.addItem(3, "Madrid")
END IF
```

**Related concepts**

- `ui.ComboBox.getTableName` on page 2567
  Get the table prefix of the form field.

- `ui.ComboBox.getIndexOf`
  Get an item position by name.

**Syntax**

```plaintext
getIndexOf ( 
    code STRING )
RETURNS INTEGER
```

1. `code` is the name of a combobox item.

**Usage**

The `getIndexOf()` method takes an item name as parameter and returns the position of the item in the drop down list.

The first item is at position 1. The method returns 0 (zero) if the item name does not exist.

The following example checks for item existence, before adding the item.

```plaintext
IF cb.getIndexOf("SFO") == 0 THEN
    CALL cb.addItem("SFO", "San Francisco International Airport, CA")
END IF
```

- `ui.ComboBox.getItemCount`
  Get the number of items.

**Syntax**

```plaintext
getItemCount ()
RETURNS INTEGER
```

**Usage**

The `getItemCount()` method returns the current number of items defined for the COMBOBOX form field.

The method returns 0 (zero) if no items are defined.
ui.ComboBox.getItemName
Get an item name by position.

Syntax

得到 itemName ( index INTEGER )
RETURNS STRING

1. index is the index of the combobox item.

Usage

The getItemName() method returns the name of an item at the given position.

The first item starts at position 1.

ui.ComboBox.getItemText
Get the item text by position.

Syntax

得到 itemText ( index INTEGER )
RETURNS STRING

1. index is the index of the combobox item.

Usage

The getItemText() method returns the display label of an item at the given position.

The first item starts at position 1.

ui.ComboBox.getTableName
Get the table prefix of the form field.

Syntax

得到 tableName ()
RETURNS STRING

Usage

The getTableName() method returns the name of the form field table prefix. The form field table prefix can be NULL if not defined at the form field level.

This allows for the identification of a COMBOBOX field in your program, for example to fill the drop down list with the appropriate items.

Related concepts

ui.ComboBox.getColumn_Name on page 2565
Get the column name of the form field.

**ui.ComboBox.getTag**
Get the combobox tag value.

**Syntax**

```plaintext
getTag ()
RETURNS STRING
```

**Usage**
The `getTag()` method returns the value define by the TAG attribute.

Use the tag to mark COMBOBOX form items with your own flags, in order to adapt the configuration of the combobox dynamically by program. For example, if TAG contains the token "short", fill the drop down list with short names, otherwise fill with long names. The same code can then be used for different COMBOBOX form fields.

**ui.ComboBox.getTextOf**
Get the item text by name.

**Syntax**

```plaintext
ggetTextOf (   code STRING   )
RETURNS STRING
```

1. `code` is the name of a combobox item.

**Usage**
The `getTextOf()` method returns the display label of the item identified by the name passed as parameter.

The method returns NULL if the item name does not exist.

**ui.ComboBox.removeItem**
Remove an item by name.

**Syntax**

```plaintext
removeItem (   code STRING   )
```

1. `code` is the name of a combobox item.

**Usage**
The `removeItem()` method deletes an item from the list. The item to be removed is identified by the name passed as a parameter. If the item does not exist, the method returns without error.

The following example fills a combobox with a few items, and then removes one by using the item name:

```plaintext
DEFINE cb ui.ComboBox
...
LET cb = ui.ComboBox.forName("formonly.airport")
CALL cb.addItem("CDG", "Paris-Charles de Gaulle, France")
CALL cb.addItem("LCY", "London-City Airport, UK")
CALL cb.addItem("LHR", "London-Heathrow, UK")
...
```
CALL cb.removeItem("CDG")

**Related concepts**
Example Get a ComboBox form field view and fill the item list on page 2569

**Examples**
ui.ComboBox usage examples.

**Example Get a ComboBox form field view and fill the item list**

Form Specification File:

```
LAYOUT
GRID
{
  Airport: [cb01]
}
END
END
ATTRIBUTES
COMBOBOX cb01 = FORMONLY.airport TYPE CHAR;
END
```

Program File:

```
MAIN
  DEFINE cb ui.ComboBox
  DEFINE airport CHAR(3)
  OPEN FORM f1 FROM "combobox"
  DISPLAY FORM f1
  LET cb = ui.ComboBox.forName("formonly.airport")
  IF cb IS NULL THEN
    ERROR "Form field not found in current form"
    EXIT PROGRAM
  END IF
  CALL cb.clear()
  CALL cb.addItem("CDG", "Paris-Charles de Gaulle, France")
  CALL cb.addItem("LCY", "London-City Airport, UK")
  CALL cb.addItem("LHR", "London-Heathrow, UK")
  CALL cb.addItem("FRA", "Frankfurt Airport, Germany")
  IF cb.getIndexof("SFO") == 0 THEN
    CALL cb.addItem("SFO", "San Francisco International Airport, CA")
  END IF
  END MAIN
```

**Example Using the INITIALIZER attribute in the form file**

Form Specification File:

```
LAYOUT
GRID
{
  Airport: [cb01]
}
END
END
ATTRIBUTES
COMBOBOX cb01 = FORMONLY.airport TYPE CHAR, INITIALIZER=initcombobox;
END
```
Initialization function:

```pascal
FUNCTION initcombobox(cb)
  DEFINE cb ui.ComboBox
  CALL cb.clear()
  CALL cb.addItem("CDG", "Paris-Charles de Gaulle, France")
  CALL cb.addItem("LCY", "London-City Airport, UK")
  CALL cb.addItem("LHR", "London-Heathrow, UK")
  CALL cb.addItem("FRA", "Frankfurt Airport, Germany")
  CALL cb.addItem("SFO", "San Francisco International Airport, CA")
END FUNCTION
```

Related concepts

- **Filling a COMBOBOX item list** on page 1735
  The item list of COMBOBOX fields can be initialized at runtime.

**The DragDrop class**

The `ui.DragDrop` class is used to control the events related to drag & drop events.

When implementing drag & drop in a dialog, the `ON DRAG* / ON DROP` dialog control blocks take an `ui.DragDrop` variable as a parameter to let you configure and control the drag & drop events. The `ui.DragDrop` variable must be declared in the scope of the dialog implementing drag & drop.

**Related concepts**

- **Drag & drop** on page 1917
Explains programming techniques for the drag & drop feature.

**ui.DragDrop methods**
Methods of the `ui.DragDrop` class.

**Table 537: Object methods**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addPossibleOperation(operation STRING)</code></td>
<td>Add a possible operation.</td>
</tr>
<tr>
<td><code>dropInternal()</code></td>
<td>Perform built-in row drop in trees.</td>
</tr>
<tr>
<td><code>getBuffer()</code></td>
<td>Get drag &amp; drop data from the buffer.</td>
</tr>
<tr>
<td><code>getLocationParent()</code></td>
<td>Get the index of the parent node where the object was dropped.</td>
</tr>
<tr>
<td><code>getLocationRow()</code></td>
<td>Get the index of the target row where the object was dropped.</td>
</tr>
<tr>
<td><code>getOperation()</code></td>
<td>Identify the type of operation on drop.</td>
</tr>
<tr>
<td><code>getSelectedMimeType()</code></td>
<td>Get the previously selected MIME type.</td>
</tr>
<tr>
<td><code>selectMimeType(mimeType STRING)</code></td>
<td>Select the MIME type before getting the data.</td>
</tr>
<tr>
<td><code>setBuffer(buffer STRING)</code></td>
<td>Set the text data of the dragged object.</td>
</tr>
<tr>
<td><code>setFeedback(feedback STRING)</code></td>
<td>Define the appearance of the target during Drag &amp; Drop.</td>
</tr>
<tr>
<td><code>setMimeType(mimeType STRING)</code></td>
<td>Define the MIME type of the dragged object.</td>
</tr>
<tr>
<td><code>setOperation(operation STRING)</code></td>
<td>Define the type of Drag &amp; Drop operation.</td>
</tr>
</tbody>
</table>
ui.DragDrop.addPossibleOperation
Add a possible operation.

**Syntax**

```java
addPossibleOperation(
    operation STRING )
```

1. `operation` is the name of a drag & drop operation.

**Usage**

Drag & drop actions can be of different kinds; you can do a copy of the dragged object, or move the dragged object from the source to the destination.

The default drag & drop operation is defined by a call to `setOperation()` method in `ON DRAG_START`. Use the `addPossibleOperation()` method to define additional operations that are allowed.

See `setOperation()` for possible values.

**Related concepts**

*Understanding drag & drop on page 1917*
This is an introduction to drag & drop programming.

ui.DragDrop.dropInternal
Perform built-in row drop in trees.

**Syntax**

```java
dropInternal()
```

**Usage**

In order to simplify drag & drop programming in the same list, the `ui.DragDrop` class provides the `dropInternal()` utility method, to be called in the `ON DROP` block. This method will perform all the row changes in the array and move row selection as well as cell attributes.

When implementing drag & drop on a tree-view, dropping an element on the tree requires complex code in order to handle parent-child relationships. Nodes can be inserted under a parent between two children, appended at the end of the children list, and at different levels in the tree hierarchy. However, the `dropInternal()` method can also be used on simple lists displayed in a regular `TABLE`.

A call to `dropInternal()` will silently be ignored, if the drag source is not the drop target, or if the method is called in a different context as `ON DROP`.

For more details about dropping elements in tree-views, see `Drag & drop` on page 1917.

ui.DragDrop.getBuffer
Get drag & drop data from the buffer.

**Syntax**

```java
getBuffer()
    RETURNS STRING
```

**Usage**

After identifying the MIME type of a dropped object with `getSelectedMimeType()`, you can call the `getBuffer()` method to get text data from the drag & drop buffer.
Drag & drop data is only available at **ON DROP** time, therefore, the `getBuffer()` method must be called in **ON DROP** only.

**Note:** When using URI MIME types (for file paths for example), the string returned from the `getBuffer()` method can contain URL-encoded characters such as `%5E`, which represents the `^` caret. Therefore, you must URL-decode strings returned from `getBuffer()` with `util.Strings.urlDecode()`. When setting the drag & drop buffer content, if required by the front-end platform, the string can be URL-encoded using `util.Strings.urlEncode()` for `setBuffer()`. However, URL-encoding file paths for `setBuffer()` is usually not required.

**Related concepts**
- Handle drag & drop data with MIME types on page 1919
- How to handle MIME types with drag & drop?

`ui.DragDrop.getLocationParent`

Get the index of the parent node where the object was dropped.

**Syntax**

```
getLocationParent()
RETURNS INTEGER
```

**Usage**

When using a tree view, a node can be dropped as a **sibling** or as a **child** node to another node. In order to distinguish between the cases, you must use the `getLocationParent()` method, which returns the index of the parent node of the drop target node returned by `getLocationRow()`.

If both methods return the same row index, you must append the dropped row as a child of the target node. Otherwise, `getLocationParent()` identifies the parent node where the dropped row has to be added as a child, and `getLocationRow()` is the index of a sibling node. In the latter case the dropped node must be inserted before the node identified by `getLocationRow()`.

These methods are typically used in the **ON DROP** block, but can also be used in **ON DRAG_OVER** to deny the drop depending on the indexes returned; for example, the program might only allow the drop of objects as new children for a given parent node.

**Related concepts**
- `ui.DragDrop.getLocationRow` on page 2573
  Get the index of the target row where the object was dropped.

`ui.DragDrop.getLocationRow`

Get the index of the target row where the object was dropped.

**Syntax**

```
getLocationRow()
RETURNS INTEGER
```

**Usage**

The `getLocationRow()` method returns the index of the row in the drop target list pointed to by the mouse cursor.

This method is typically used in the **ON DROP** block to get the index of the target row to be modified or replaced by the dragged object.
In order to deny the drop, the `getLocationRow()` can be used in conjunction with `setOperation(NULL)` in the ON DRAG_ENTER or ON DRAG_OVER, if the current target row returned by `getLocationRow()` is not valid for a drop operation.

**Related concepts**

*ui.DragDrop.getLocationParent* on page 2573
Get the index of the parent node where the object was dropped.

*ui.DragDrop.getSelectedMimeType*
Get the previously selected MIME type.

**Syntax**

```plaintext
getSelectedMimeType()
RETURNS STRING
```

**Usage**

Before retrieving data from the drag & drop buffer with `getBuffer()`, first call the `getSelectedMimeType()` method, to identify the data format that was previously selected by a `selectMimeType()` call.

The `getSelectedMimeType()` method is typically called in ON DROP to identity the format of the dropped object.

**Related concepts**

*Handle drag & drop data with MIME types* on page 1919
How to handle MIME types with drag & drop?

*ui.DragDrop.getOperation*
Identify the type of operation on drop.

**Syntax**

```plaintext
getOperation()
RETURNS STRING
```

**Usage**

The `getOperation()` method returns the type of the current drag & drop operation ("copy", "move", or "none").

Depending on the value returned by this method, the program can make the appropriate changes in the data model. For example, after a row has been dropped into another list, the source list can remove the original row if the operation was a "move", but keeps the original row if the operation was a "copy".

The `getOperation()` method is typically called in the ON DRAG_FINISHED block.

**Related concepts**

*Understanding drag & drop* on page 1917
This is an introduction to drap & drop programming.

*ui.DragDrop.setOperation* on page 2577
Define the type of Drag & Drop operation.

*ui.DragDrop.addPossibleOperation* on page 2572
Add a possible operation.

*ui.DragDrop.selectMimeType*
Select the MIME type before getting the data.

**Syntax**

```java
selectMimeType(
    mimeType STRING )
```

1. *mimeType* defines the MIME type for dragged objects.

**Usage**

Call the `selectMimeType()` method to check that data is available in a format identified by the MIME type passed as parameter.

If this type of data is available in the buffer, the method returns `TRUE` and you can later get the data with `getBuffer()`.

The `selectMimeType()` method is typically used in `ON DRAG_ENTER`, `ON DRAG_OVER` to deny the drag & drop operation if none of the supported MIME types is available in the buffer.

The `getSelectedMimeType()` method can be used to identify which MIME type has been selected.

**Related concepts**

*Handle drag & drop data with MIME types* on page 1919

*How to handle MIME types with drag & drop?*

*ui.DragDrop.setBuffer*
Set the text data of the dragged object.

**Syntax**

```java
setBuffer(
    buffer STRING )
```

1. *buffer* is a string expression containing drag & drop data.

**Usage**

Use the `setBuffer()` method to provide the text data of objects dragged from the program to an external application.

The `setBuffer()` method is typically used in an `ON DRAG_START` block together with `setMimeType()`.

By default, the dialog will serialize the data of the selected rows as a tab-separated list of values.

The text/plain MIME type is the default.

**Note:** When using URI MIME types (for file paths for example), the string returned from the `getBuffer()` method can contain URL-encoded characters such as `%5E`, which represents the `^` caret. Therefore, you must URL-decode strings returned from `getBuffer()` with `util.Strings.urlDecode()`. When setting the drag & drop buffer content, if required by the front-end platform, the string can be URL-encoded using `util.Strings.urlEncode()` for `setBuffer()`. However, URL-encoding file paths for `setBuffer()` is usually not required.

**Related concepts**

*Handle drag & drop data with MIME types* on page 1919
How to handle MIME types with drag & drop?

ui.DragDrop.setFeedback
Define the appearance of the target during Drag & Drop.

Syntax

```plaintext
setFeedback(
    feedback STRING )
```

1. `feedback` is the type of feedback to display during the drag & drop operation.

Usage

The `setFeedback()` method defines the appearance the target object must have during the drag & drop process.

For example, in a table or tree view, when the mouse is flying over rows in the drop target, a different visual indicator will appear depending on the value that was passed to `setFeedback()`.

Possible values for the `setFeedback()` method are:

<table>
<thead>
<tr>
<th>Parameter Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>Dragged object will be dropped somewhere on the target widget, the exact location does not matter.</td>
</tr>
<tr>
<td>insert</td>
<td>In lists, dragged object will be inserted in between existing rows.</td>
</tr>
<tr>
<td>select</td>
<td>In lists, dragged object will replace the current row under the mouse.</td>
</tr>
</tbody>
</table>

Related concepts

Understanding drag & drop on page 1917
This is an introduction to drag & drop programming.

ui.DragDrop.setMimeType
Define the MIME type of the dragged object.

Syntax

```plaintext
setMimeType(
    mimeType STRING )
```

1. `mimeType` defines the MIME type for the drag & drop buffer.

Usage

Objects dragged from the program to an external application need to be identified with a MIME type and the program must provide the data. The MIME type can be specified with the `setMimeType()` method.

The `setMimeType()` method is typically used in an `ON DRAG_START` block along with `setBuffer()`.

By default, the source target will use the text/plain MIME type and copy the data of the selected rows into the Drag & Drop buffer.

Related concepts

Handle drag & drop data with MIME types on page 1919
How to handle MIME types with drag & drop?

`ui.DragDrop.setOperation`  
Define the type of Drag & Drop operation.

**Syntax**

```
setOperation(  
operation STRING )
```

1. `operation` is the name of a drag & drop operation.

**Usage**

Drag & drop actions can be of different kinds; you can do a copy of the dragged object, or move the dragged object from the source to the destination.

Use the `setOperation()` method to define/force the type of drag & drop operation or to deny/cancel the drag & drop process.

**Note:** The `addPossibleOperation()` method can be used to specify additional drag & drop operations that are possible in this dialog.

**Table 539: Parameters for the setOperation() method**

<table>
<thead>
<tr>
<th>Parameter Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>To deny/cancel the drag &amp; drop process.</td>
</tr>
<tr>
<td>copy</td>
<td>To allow drag &amp; drop as a copy of the source object.</td>
</tr>
<tr>
<td>move</td>
<td>To allow drag &amp; drop as a move of the source object.</td>
</tr>
</tbody>
</table>

The `setOperation()` method can be called in different drag & drop triggers.

A common usage is to deny drag & drop by passing `NULL` in the `ON DRAG_ENTER` and/or `ON DRAG_OVER` blocks because the dragged object does not correspond to the type of objects the target can receive.

This method is also used in `ON DRAG_START` to force a specific type of drag & drop operation (copy or move), or to deny drag start if the context does not allow a drag & drop action.

When called in the `ON DRAG_ENTER` block, the method forces a specific drag & drop operation.

**Related concepts**

- [Understanding drag & drop](#) on page 1917  
  This is an introduction to drag & drop programming.

**The om package**

These topics cover the built-in classes of the `om` package

**The DomDocument class**

The `om.DomDocument` class provides methods to manipulate a data tree, following the DOM standards.


A unique root `om.DomNode` object is owned by an `om.DomDocument` object.
### om.DomDocument methods
Methods of the om.DomDocument class.

#### Table 540: Class methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>

#### Table 541: Object methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>createChars(value STRING)</code></td>
<td>Create a new text node in the DOM document.</td>
</tr>
<tr>
<td><code>createElement(tagName STRING)</code></td>
<td>Create a new element node in the DOM document.</td>
</tr>
<tr>
<td><code>createEntity(name STRING)</code></td>
<td>Create a new entity node in the DOM document.</td>
</tr>
<tr>
<td><code>copy(old om.DomNode, deep INTEGER)</code></td>
<td>Create a new element node by copying an existing node.</td>
</tr>
<tr>
<td><code>getDocumentById(id INTEGER)</code></td>
<td>Returns a node element ID based on the internal AUI tree id.</td>
</tr>
<tr>
<td><code>getDocumentElement()</code></td>
<td>Returns the root node element of the DOM document.</td>
</tr>
<tr>
<td><code>removeElement(oldChild om.DomNode)</code></td>
<td>Remove a DomNode object and all its descendants.</td>
</tr>
</tbody>
</table>
**om.DomDocument.create**
Create a new empty om.DomDocument object.

**Syntax**

```javascript
om.DomDocument.create(
    tagName STRING 
) 
RETURNS om.DomDocument
```

1. `tagName` defines the tag name of the root element.

**Usage**
Use the class method `om.DomDocument.create()` to instantiate a new, empty DOM document object.

To hold the reference to a DOM document object, define a variable with the `om.DomDocument` type.

**Example**

```javascript
DEFINE d om.DomDocument
LET d = om.DomDocument.create("Vehicles")
...
```

**Related concepts**
- `om.DomDocument.createFromXmlFile` on page 2580
- `om.DomDocument.createFromString` on page 2579
  Create a new `om.DomDocument` object from an XML string.

**om.DomDocument.createFromString**
Create a new `om.DomDocument` object from an XML string.

**Syntax**

```javascript
om.DomDocument.createFromString(
    s STRING 
) 
RETURNS om.DomDocument
```

1. `s` is the string expression containing XML data.

**Usage**
Use the class method `om.DomDocument.createFromString()` to instantiate a new DomDocument object that is filled with the content of the specified XML formatted string.

To hold the reference to a DOM document object, define a variable with the `om.DomDocument` type.

**Example**

```javascript
DEFINE d om.DomDocument
LET d = om.DomDocument.createFromString("<Vehicles/>")
...
```

**Related concepts**
- `om.DomDocument.create` on page 2579
  Create a new empty `om.DomDocument` object.
- `om.DomDocument.createFromXmlFile` on page 2580

**om.DomDocument.createFromXmlFile**

**Syntax**

```plaintext
om.DomDocument.createFromXmlFile(
    path STRING
) RETURNS om.DomDocument
```

1. `path` is the path to the file containing XML data.

**Usage**
Use the class method `om.DomDocument.createFromXmlFile()` to instantiate a new `DomDocument` object that is filled with the content of the specified XML file.

To hold the reference to a DOM document object, define a variable with the `om.DomDocument` type.

**Example**

```plaintext
DEFINE d om.DomDocument
LET d = om.DomDocument.createFromXmlFile("vehicles.xml")
...
```

**Related concepts**
- `om.DomDocument.create` on page 2579
  Create a new empty `om.DomDocument` object.
- `om.DomDocument.createFromString` on page 2579
  Create a new `om.DomDocument` object from an XML string.

**om.DomDocument.getDocumentElement**
Returns the root node element of the DOM document.

**Syntax**

```plaintext
getDocumentElement ()
RETURNS om.DomNode
```

**Usage**

To hold the reference to the root node, define a variable with the `om.DomNode` type.

**Example**

```plaintext
MAIN
    DEFINE mydoc om.DomDocument
    DEFINE n om.DomNode
    LET mydoc = om.DomDocument.create("Test")
    LET n = mydoc.getDocumentElement()
END MAIN
```

**Related concepts**
- The `DomNode` class  on page 2584
The `om.DomNode` class provides methods to manipulate a DOM node of a data tree.

`om.DomDocument.getElementById`
Returns a node element ID based on the internal AUI tree id.

**Syntax**

```plaintext
getDocumentById(
    id INTEGER )
RETURNS om.DomNode
```

**Usage**
The method `getElementById()` returns the `om.DomNode` element of the DOM document based on the internal id number passed as parameter.

Each DOM node gets an internal integer id when it is created in the abstract user interface tree. It can be referenced by this unique id. The node id is typically used in other nodes, to reference a node in the DOM document.

To hold the reference to the root node, define a variable with the `om.DomNode` type.

**Example**

```plaintext
MAIN
    DEFINE uid om.DomDocument
    DEFINE n om.DomNode
    LET uid = ui.Interface.getDocument()
    MENU "test"
        COMMAND "Get UI node by ID"
            LET n = uid.getElementById(1)
            IF n IS NOT NULL THEN
                DISPLAY n.toString()
            END IF
    COMMAND "Exit"
    EXIT MENU
END MENU
END MAIN
```

**Related concepts**

The `DomNode class` on page 2584
The `om.DomNode` class provides methods to manipulate a DOM node of a data tree.

`om.DomNode.getId` on page 2595
Returns the internal AUI tree id of a DOM node.

`om.DomDocument.createChars`
Create a new text node in the DOM document.

**Syntax**

```plaintext
createChars(
    value STRING )
RETURNS om.DomNode
```

1. `value` defines the content of the text node.

**Usage**
Use the method `createChars()` to create a new `om.DomNode` text node. The content of the text node must be passed as parameter.
The created node will have the reserved tagName "@chars", and a single attribute named "@chars" storing the character data.

To hold the reference to the new node, define a variable with the om.DomNode type.

```
MAIN
  DEFINE mydoc  om.DomDocument
  DEFINE root, text_node om.DomNode
  LET mydoc = om.DomDocument.create("Test")
  LET root = mydoc.getDocumentElement()
  LET text_node = mydoc.createChars("Hello, world!")
  DISPLAY text_node.getTagName() -- shows "@chars"
  DISPLAY text_node.getAttribute("@chars") -- the text
  CALL root.appendChild(text_node)
  CALL root.writeXML("output.xml")
END MAIN
```

Related concepts

The DomNode class on page 2584
The om.DomNode class provides methods to manipulate a DOM node of a data tree.

*om.DomDocument.createElement*
Create a new element node in the DOM document.

Syntax

```
createElement(
  tagName STRING
)
RETURNS om.DomNode
```

1. `tagName` defines the tag name of the node.

Usage

Use the method `createElement()` to create a new om.DomNode element node. The tag name of the element must be passed as parameter.

To hold the reference to the new node, define a variable with the om.DomNode type.

```
MAIN
  DEFINE mydoc om.DomDocument
  DEFINE n om.DomNode
  LET mydoc = om.DomDocument.create("Test")
  LET n = mydoc.createElement("Car")
END MAIN
```

Related concepts

The DomNode class on page 2584
The om.DomNode class provides methods to manipulate a DOM node of a data tree.

*om.DomDocument.createEntity*
Create a new entity node in the DOM document.

Syntax

```
createEntity()
```

name STRING )
RETURNS om.DomNode

1. *name* defines the name of the entity node.

**Usage**

Use the method `createEntity()` to create a new `om.DomNode` entity node. The entity name must be passed as parameter.

The text representation of a entity node is `&name;`.

The created node will have the reserved tagName "@entity", with a single attribute named "@entity" containing the text of the entity.

To hold the reference to the new node, define a variable with the `om.DomNode` type.

**Example**

```om
MAIN
   DEFINE mydoc om.DomDocument
   DEFINE n om.DomNode
   LET mydoc = om.DomDocument.create("Test")
   LET n = mydoc.createEntity("quote")
END MAIN
```

**Related concepts**

The `DomNode` class on page 2584

The `om.DomNode` class provides methods to manipulate a DOM node of a data tree.

`om.DomDocument.copy`

Create a new element node by copying an existing node.

**Syntax**

```
copy(
   old om.DomNode, 
   deep INTEGER )
RETURNS om.DomNode
```

1. *old* references the source node to copy.
2. *deep* is a boolean to control the recursive node copy.

**Usage**

Use the method `copy()` to create a new `om.DomNode` element node from an existing node.

Pass `TRUE` as second parameter to clone a complete tree of nodes.

To hold the reference to the new node, define a variable with the `om.DomNode` type.

**Example**

```om
MAIN
   DEFINE mydoc om.DomDocument
   DEFINE n, s om.DomNode
   LET s = mydoc.createElement("Car")
   LET n = mydoc.copy(s, TRUE)
END MAIN
```

**Related concepts**

The `DomNode` class on page 2584
The `om.DomNode` class provides methods to manipulate a DOM node of a data tree.

**om.DomDocument.removeElement**
Remove a DomNode object and all its descendants.

**Syntax**

```plaintext
removeElement(
  oldChild  om.DomNode  )
```

1. `oldChild` is the DOM node to be removed.

**Usage**
Use the `removeElement()` method to remove an element and all its descendants from DOM document.

Any reference to the removed `om.DomNode` objects becomes invalid.

**Examples**
`om.DomDocument` usage examples.

**Example 1: Creating a DOM document**

```plaintext
MAIN
  DEFINE d  om.DomDocument
  DEFINE r  om.DomNode
  LET d = om.DomDocument.create("MyDocument")
  LET r = d.getDocumentElement()
END MAIN
```

**The DomNode class**
The `om.DomNode` class provides methods to manipulate a DOM node of a data tree.

This class follows the DOM standards.

A DomNode object is a node (or element) of a `DomDocument`.

Tag and attribute names of DOM nodes are case sensitive; "Wheel" is not the same as "wheel".

Text nodes cannot have attributes, but they have plain text. In text nodes, the characters can be accessed with the `@chars` attribute name. In XML representation, a text node is the text itself. Do not confuse it with the parent node. For example, `<Item id="32">Red shoes</Item>` represents 2 nodes: The parent 'Item' node and a text node with string 'Red shoes'.

If you need to identify an element, use a common attribute like "name". If you need to label an element, use a common attribute like "text".
### om.DomNode methods
Methods of the om.DomNode class.

#### Table 542: Object methods: Node creation

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`appendChild(</td>
<td>Adds an existing node at the end of the list of children in the current node.</td>
</tr>
<tr>
<td>newChild om.DomNode</td>
<td>)</td>
</tr>
<tr>
<td>`createChild(</td>
<td>Creates and adds a node at the end of the list of children in the current node.</td>
</tr>
<tr>
<td>tagName STRING )</td>
<td>RETURNS om.DomNode</td>
</tr>
<tr>
<td>`insertBefore(</td>
<td>Inserts an existing node before the existing node specified.</td>
</tr>
<tr>
<td>newChild om.DomNode,</td>
<td>)</td>
</tr>
<tr>
<td>oldChild om.DomNode)</td>
<td>DELETE</td>
</tr>
<tr>
<td>`removeChild(</td>
<td>Deletes the specified child node from the current node.</td>
</tr>
<tr>
<td>node om.DomNode )</td>
<td>)</td>
</tr>
<tr>
<td>`replaceChild(</td>
<td>Replaces a node by another in the child nodes of the current node.</td>
</tr>
<tr>
<td>new om.DomNode,</td>
<td>)</td>
</tr>
<tr>
<td>old om.DomNode)</td>
<td>)</td>
</tr>
</tbody>
</table>

#### Table 543: Object methods: In/Out

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`loadXml(</td>
<td>Load an XML file into the current node.</td>
</tr>
<tr>
<td>path STRING )</td>
<td>RETURNS om.DomNode</td>
</tr>
<tr>
<td>`parse(</td>
<td>Parses an XML formatted string and creates the DOM structure in the current node.</td>
</tr>
<tr>
<td>s STRING )</td>
<td>RETURNS om.DomNode</td>
</tr>
<tr>
<td>`toString()</td>
<td>Serializes the current node into an XML formatted string.</td>
</tr>
<tr>
<td>[ ]</td>
<td>RETURNS STRING</td>
</tr>
<tr>
<td>`write(</td>
<td>Processes a DOM document with a SAX document handler.</td>
</tr>
<tr>
<td>sdh om.SaxDocumentHandler )</td>
<td>)</td>
</tr>
<tr>
<td>`writeXml(</td>
<td>Creates an XML file from the current DOM node.</td>
</tr>
<tr>
<td>path STRING )</td>
<td>)</td>
</tr>
</tbody>
</table>
### Table 544: Object methods: Node identification

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getId()</code></td>
<td>Returns the internal AUI tree id of a DOM node.</td>
</tr>
<tr>
<td><code>getTagName()</code></td>
<td>Returns the XML tag name of a DOM node.</td>
</tr>
</tbody>
</table>

### Table 545: Object methods: Attributes management

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getAttribute(name STRING)</code></td>
<td>Returns the value of a DOM node attribute.</td>
</tr>
<tr>
<td><code>getAttributesCount()</code></td>
<td>Returns the number of attributes in the DOM node.</td>
</tr>
<tr>
<td><code>getAttributeInteger(name STRING, defaultValue STRING)</code></td>
<td>Returns the value of a DOM node attribute, with default integer value.</td>
</tr>
<tr>
<td><code>getAttributeString(name STRING, defaultValue STRING)</code></td>
<td>Returns the value of a DOM node attribute, with default string value.</td>
</tr>
<tr>
<td><code>getAttributeName(index INTEGER)</code></td>
<td>Returns the name of a DOM node attribute by position.</td>
</tr>
<tr>
<td><code>getAttributeValue(index INTEGER)</code></td>
<td>Returns the value of a DOM node attribute by position.</td>
</tr>
<tr>
<td><code>setAttribute(name STRING, value STRING)</code></td>
<td>Sets the value of a DOM node attribute.</td>
</tr>
<tr>
<td><code>removeAttribute(name STRING)</code></td>
<td>Delete the specified attribute from the DOM node.</td>
</tr>
</tbody>
</table>
Table 546: Object methods: Tree navigation

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getChildByIndex(index INTEGER)</td>
<td>Returns a child DOM node by position.</td>
</tr>
<tr>
<td>getFirstChild()</td>
<td>Returns the first child DOM node.</td>
</tr>
<tr>
<td>getLastChild()</td>
<td>Returns the last child DOM node.</td>
</tr>
<tr>
<td>getNext()</td>
<td>Returns the next sibling DOM node of this node.</td>
</tr>
<tr>
<td>getParent()</td>
<td>Returns the parent DOM node.</td>
</tr>
<tr>
<td>getPrevious()</td>
<td>Returns previous sibling DOM node of this node.</td>
</tr>
<tr>
<td>selectByPath(path STRING)</td>
<td>Finds descendant DOM nodes from an XPath-like pattern.</td>
</tr>
<tr>
<td>selectByTagName(tagName STRING)</td>
<td>Finds descendant DOM nodes based on a tag name.</td>
</tr>
</tbody>
</table>

`om.DomNode.appendChild`
Adds an existing node at the end of the list of children in the current node.

**Syntax**

```plaintext
appendChild(
    newChild om.DomNode
)
```

1. `newChild` is a reference to a node.

**Usage**
The `appendChild()` method takes an existing `om.DomNode` element node and adds it at the end of the children of the object node calling the method.
The child node passed to the `appendChild()` method must have been created from the same DOM document object, for example with the `om.DomDocument.createElement()` method.

If the node passed to the `appendChild()` method is already attached to another parent node, it will be detached from that parent node before being attached to the new parent node.

**Example**

```om
DEFINE parent, child om.DomNode
...
LET child = parent.appendChild("Item")
```

For a complete example, see Example 1: Creating a DOM tree on page 2605.

**Related concepts**

- `om.DomNode.replaceChild` on page 2599
  Replaces a node by another in the child nodes of the current node.
- `om.DomNode.removeChild` on page 2598
  Deletes the specified child node from the current node.
- `om.DomDocument.createElement` on page 2582
  Create a new element node in the DOM document.

**om.DomNode.createChild**

Creates and adds a node at the end of the list of children in the current node.

**Syntax**

```om
createChild(
    tagName STRING )
RETURNS om.DomNode
```

1. `tagName` is the tag name of the new node.

**Usage**

The `createChild()` method creates a new `om.DomNode` element with the tag name passed as parameter, and adds it at the end of the children of the object node calling the method.

The method returns the reference to the created object.

**Note:** When using the string "@chars" as tag name, the `createChild()` method creates an XML text node. To set/get the value of a text node, use the "@chars" as attribute name with `getAttribute()`/`setAttribute()`. Alternatively, the `createChars()` method of the `om.DomDocument` class can be used, to create a text node and set the value in a single step.

**Example**

```om
DEFINE parent, child, text_node om.DomNode
...
LET child = parent.createChild("Item")
...
LET text_node = parent.createChild("@chars")
CALL text_node.setAttribute("@chars", "my text goes here")
```

For a complete example, see Example 1: Creating a DOM tree on page 2605.

**Related concepts**

- `om.DomNode.appendChild` on page 2587
Adds an existing node at the end of the list of children in the current node.

*om.DomNode.insertBefore*

Inserts an existing node before the existing node specified.

**Syntax**

```javascript
insertBefore(
    newChild om.DomNode,
    oldChild om.DomNode)
```

1. *newChild* is a reference to a new created node.
2. *oldChild* is a reference to a child node existing in the current node.

**Usage**

The `insertBefore()` method takes an existing `om.DomNode` element node and inserts it before the child node passed as second parameter, in the list of nodes of the object node calling the method.

The child node passed to the `insertBefore()` method must have been created from the same DOM document object, for example with the `om.DomDocument.createElement()` method.

**Important:** When using the `om` API to update the AUI tree, it is not possible to insert nodes, if the parent node has already been synchronized with the front-end. Child nodes can be inserted, only if the parent node is not yet synchronized with the front-end. For more details, see Manipulating the abstract user interface tree on page 1107.

**Example**

```javascript
DEFINE parent, other, child om.DomNode
...
LET child = mydoc.createElement("Item")
CALL parent.insertBefore(child, other)
```

**Related concepts**

*om.DomDocument.createElement* on page 2582

Create a new element node in the DOM document.

*om.DomNode.loadXml*

Load an XML file into the current node.

**Syntax**

```javascript
loadXml(
    path STRING )
RETURNS om.DomNode
```

1. *path* is the path to the XML file.

**Usage**

The `loadXml()` method takes a file path as parameter and loads the XML content into the current node, by creating a new DOM structure in memory. The method then returns the created child DOM node.

To hold the reference to the new node, define a variable with the `om.DomNode` type.

**Note:** Files encoded in UTF-8 can start with the UTF-8 Byte Order Mark (BOM), a sequence of 0xEF 0xBB 0xBF bytes, also known as UNICODE U+FEFF. When reading files, Genero BDL will ignore the UTF-8 BOM, if it is present at the beginning of the file. This applies to instructions such as `LOAD`, as well as I/O APIs such as `base.Channel.read()` and `readLine()`.
**Example**

```javascript
DEFINE parent, new om.DomNode
...
LET new = parent.loadXml("myfile.xml")
```

**Related concepts**

- `om.DomNode.writeXml` on page 2604
  Creates an XML file from the current DOM node.

- `om.DomNode.parse`
  Parses an XML formatted string and creates the DOM structure in the current node.

**Syntax**

```
parse(  
   s STRING  
)  
RETURNS om.DomNode
```

1. `s` is an XML formatted string.

**Usage**

The `parse()` method scans the XML formatted string passed as parameter and creates the corresponding DOM nodes in the current node. The method then returns the created child DOM node.

The node must be created before it is passed as parameter to this method, typically, with `om.DomDocument.createElement()`.

**Example**

```javascript
DEFINE parent, child om.DomNode
...
LET child = parent.parse("<Item/>")
```

**Related concepts**

- `om.DomNode.toString` on page 2604
  Serializes the current node into an XML formatted string.

- `om.DomNode.getAttribute`
  Returns the value of a DOM node attribute.

**Syntax**

```
getAttribute(  
   name STRING  
)  
RETURNS STRING
```

1. `name` is the name of the attribute.

**Usage**

The `getAttribute()` method returns the value of the attribute passed as parameter, as defined in the current node.

DOM node attribute names are case-sensitive.

If the attribute does not exist for this node type, or if the attribute is not set, the method returns `NULL`. 
For character nodes (created for example with the `createChars()` of a `DomDocument` object), you can get the text value by passing the `@chars` attribute name to the `getAttribute()` method.

**Example**

```om
DEFINE node om.DomNode
DEFINE text_node om.DomNode
...
DISPLAY node.getAttribute("color")
DISPLAY text_node.getAttribute("@chars")
```

**Related concepts**

- `om.DomNode.getAttributeString` on page 2592
  Returns the value of a DOM node attribute, with default string value.
- `om.DomNode.getAttributeInteger` on page 2591
  Returns the value of a DOM node attribute, with default integer value.
- `om.DomNode.setAttribute` on page 2603
  Sets the value of a DOM node attribute.
- `om.DomDocument.createChars` on page 2581
  Create a new text node in the DOM document.

**Syntax**

```om
def getAttributeInteger ( 
    name STRING, 
    defaultValue STRING ) 
RETURNS INTEGER
```

1. `name` is the name of the attribute.
2. `defaultValue` is the default value.

**Usage**

The `getAttributeInteger()` method returns the value of the attribute passed as parameter, as defined in the current node.

DOM node attribute names are case-sensitive.

If the attribute is not defined, the method returns the default value passed as second parameter.

**Related concepts**

- `om.DomNode.getAttribute` on page 2590
  Returns the value of a DOM node attribute.
- `om.DomNode.setAttribute` on page 2603
  Sets the value of a DOM node attribute.

- `om.DomNode.getAttributeCount`
  Returns the number of attributes in the DOM node.

**Syntax**

```om
getAttributeCount ()
RETURNS INTEGER
```
Usage

The `getAttributesCount()` method returns the number of attributes defined in the current node.

This method is typically used to scan all the attributes of a node by position, with the `getAttributeName()` and `getAttributeValue()` methods.

Example

```plaintext
FUNCTION display_attribute_names(node om.DomNode)
    DEFINE index, count INTEGER
    LET count = node.getAttributesCount()
    FOR index = 1 TO count
        DISPLAY node.getAttributeName(index)
    END FOR
END FUNCTION
```

Related concepts

- `om.DomNode.getAttributeName` on page 2592
  Returns the name of a DOM node attribute by position.
- `om.DomNode.getAttributeValue` on page 2593
  Returns the value of a DOM node attribute by position.
- `om.DomNode.getAttributeString`
  Returns the value of a DOM node attribute, with default string value.

Syntax

```plaintext
getAttributeString (name STRING, defaultValue STRING )
RETURNS STRING
```

1. `name` is the name of the attribute.
2. `defaultValue` is the default value.

Usage

The `getAttributeString()` method returns the value of the attribute passed as parameter, as defined in the current node.

DOM node attribute names are case-sensitive.

If the attribute is not defined, the method returns the default value passed as second parameter.

Related concepts

- `om.DomNode.getAttribute` on page 2590
  Returns the value of a DOM node attribute.
- `om.DomNode.setAttribute` on page 2603
  Sets the value of a DOM node attribute.

- `om.DomNode.getAttributeName`
  Returns the name of a DOM node attribute by position.

Syntax

```plaintext
getAttributeName (index INTEGER )
RETURNS STRING
```
1. *index* is the index of the attribute, starts at 1.

**Usage**
The `getAttributeName()` method returns the name of an attribute by position in the current node. DOM node attribute names are case-sensitive.
If the attribute does not exist at the given position, the method returns `NULL`.

**Example**
```java
DEFINE node om.DomNode
...
DISPLAY node.getAttributeName(12)
```
For a complete example, see Example 2: Displaying a DOM tree recursively on page 2606.

**Related concepts**
- `om.DomNode.getAttributeValue` on page 2593
  Returns the value of a DOM node attribute by position.
- `om.DomNode.getAttributesCount` on page 2591
  Returns the number of attributes in the DOM node.

**Syntax**
```java
getAttributeValue(
    index INTEGER )
RETURNS STRING
```

1. *index* is the index of the attribute, starts at 1.

**Usage**
The `getAttributeValue()` method returns the value of an attribute by position in the current node.
DOM node attribute names are case-sensitive.
If the attribute does not exist at the given position, the method returns `NULL`.

**Example**
```java
DEFINE node om.DomNode
...
DISPLAY node.getAttributeValue(12)
```
For a complete example, see Example 2: Displaying a DOM tree recursively on page 2606.

**Related concepts**
- `om.DomNode.getAttributeName` on page 2592
  Returns the name of a DOM node attribute by position.
- `om.DomNode.getAttributesCount` on page 2591
Returns the number of attributes in the DOM node.

*om.DomNode.getChildByIndex*

Returns a child DOM node by position.

**Syntax**

```plaintext
getChildByIndex(
    index INTEGER
) RETURNS om.DomNode
```

1. *index* is the index of the child node, starts at 1.

**Usage**

The `getChildByIndex()` method returns the child DOM node by position in the current node.

If there is no child node at the given position, the method returns `NULL`.

**Related concepts**

*om.DomNode.getChildCount* on page 2594

Returns the number of children nodes.

*om.DomNode.getFirstChild*

Returns the first child DOM node.

**Syntax**

```plaintext
getFirstChild()
RETURNS INTEGER
```

**Usage**

The `getFirstChild()` method returns the number of child nodes in the current node.

This method is typically used to scan the child nodes of a DOM node, with the `getChildByIndex()` method.

**Example**

```plaintext
FUNCTION display_children(node om.DomNode)
    DEFINE index, count INTEGER
    DEFINE child om.DomNode
    LET count = node.getChildCount()
    FOR index = 1 TO count
        LET child = node.getChildByIndex(index)
        DISPLAY child.toString()
    END FOR
END FUNCTION
```

**Related concepts**

*om.DomNode.getChildByIndex* on page 2594

Returns a child DOM node by position.

*om.DomNode.getFirstChild*

Returns the first child DOM node.
**Usage**

The `getFirstChild()` method returns the first child DOM node in the current node.

This method is typically used to scan child nodes with the `getNext()` method, until `getNext()` returns `NULL`.

**Example**

```plaintext
FUNCTION display_children(node om.DomNode)
    DEFINE child om.DomNode
    LET child = node.getFirstChild()
    WHILE child IS NOT NULL
        DISPLAY child.toString()
        LET child = child.getNext()
    END WHILE
END FUNCTION
```

**Related concepts**

- `om.DomNode.getNext` on page 2596
  - Returns the next sibling DOM node of this node.
- `om.DomNode.getPrevious` on page 2597
  - Returns previous sibling DOM node of this node.
- `om.DomNode.getId`
  - Returns the internal AUI tree id of a DOM node.

**Syntax**

```plaintext
getId()
RETURNS INTEGER
```

**Usage**

The `getId()` method returns an internal integer identifier generated automatically for any `om.DomNode` object created in the abstract user interface tree.

The internal id is typically used to reference a DOM node in an attribute of another node, to link nodes logically together.

If the DOM node does not belong to the AUI tree, the method returns zero.

**Related concepts**

- `om.DomDocument.getElementById` on page 2581
  - Returns a node element ID based on the internal AUI tree id.
- `om.DomNode.getLastChild`
  - Returns the last child DOM node.

**Syntax**

```plaintext
getLastChild()
RETURNS om.DomNode
```

**Usage**

The `getLastChild()` method returns the last child DOM node in the current node.
This method is typically used to scan child nodes with the `getPrevious()` method, until `getPrevious()` returns `NULL`.

**Example**

```java
FUNCTION display_children(node om.DomNode)
    DEFINE child om.DomNode
    LET child = node.getLastChild()
    WHILE child IS NOT NULL
        DISPLAY child.toString()
        LET child = child.getPrevious()
    END WHILE
END FUNCTION
```

**Related concepts**

- `om.DomNode.getPrevious` on page 2597
  
  Returns previous sibling DOM node of this node.

- `om.DomNode.getNext` on page 2596
  
  Returns the next sibling DOM node of this node.

**Syntax**

```
getNext ()
RETURNS om.DomNode
```

**Usage**

The `getNext()` method returns the next sibling DOM node following the current node, within the children list of the parent node.

**Related concepts**

- `om.DomNode.getFirstChild` on page 2594
  
  Returns the first child DOM node.

- `om.DomNode.getPrevious` on page 2597
  
  Returns previous sibling DOM node of this node.

- `om.DomNode.getParent`
  
  Returns the parent DOM node.

**Syntax**

```
getParent ()
RETURNS om.DomNode
```

**Usage**

The `getParent()` method returns the parent DOM node of the current node.

If the current node is the root node, the method returns `NULL`.

**Example**

```java
DEFINE parent, current om.DomNode
...
```
LET parent = current.getParent()

Related concepts

- `om.DomNode.getFirstChild` on page 2594
  Returns the first child DOM node.
- `om.DomNode.getLastChild` on page 2595
  Returns the last child DOM node.

- `om.DomNode.getPrevious`
  Returns previous sibling DOM node of this node.

Syntax

```
getPrevious()
  RETURNS om.DomNode
```

Usage

The `getPrevious()` method returns the previous sibling DOM node preceding the current node, within the children list of the parent node.

Related concepts

- `om.DomNode.getLastChild` on page 2595
  Returns the last child DOM node.
- `om.DomNode.getNext` on page 2596
  Returns the next sibling DOM node of this node.

- `om.DomNode.getTagName`
  Returns the XML tag name of a DOM node.

Syntax

```
getTagName()
  RETURNS STRING
```

Usage

The `getTagName()` method returns the XML tag name of the node.

Use this method to identify the type of the node.

For character nodes (created for example with the `createChars()` of a `DomDocument` object), the `getTagName()` method returns "@chars".

Example

```
DEFINE node om.DomNode
...
DISPLAY node.getTagName()
```

- `om.DomNode.removeAttribute`
  Delete the specified attribute from the DOM node.

Syntax

```
removeAttribute(
    name STRING
  )
```
1. *name* is the name of the attribute.

**Usage**

The `removeAttribute()` method deletes the attribute identified by the name passed as parameter.

DOM node attribute names are case-sensitive.

If the attribute does not exist for this node the method returns silently.

**Example**

```javascript
DEFINE node om.DomNode
...
CALL node.removeAttribute("comments")
```

**Related concepts**

*om.DomNode.setAttribute* on page 2603
Sets the value of a DOM node attribute.

*om.DomNode.removeChild*

Deletes the specified child node from the current node.

**Syntax**

```
removeChild(
    node om.DomNode )
```

1. *node* is a reference to a node.

**Usage**

The `removeChild()` method detaches an `om.DomNode` element node from the current node.

The removed node is not destroyed, if it is still referenced by a variable. The `removeChild()` method will only break the link between the parent node and the child node. The child node still exists in the DOM document, but it is an orphan node, that can be attached to another parent node in the document.

**Example**

```
MAIN
    DEFINE doc  om.DomDocument,
        r om.DomNode,
        p om.DomNode,
        c om.DomNode

    LET doc = om.DomDocument.create("Items")
    LET r = doc.createElement("Zoo")
    LET p = doc.createElement("DodoList")
    CALL r.appendChild(p)
    LET c = doc.createElement("Dodo")
    CALL c.setAttribute("name", "momo")
    CALL c.setAttribute("gender", "male")
    CALL p.appendChild(c)
    CALL r.writeXml("file1.xml")
    CALL p.removeChild(c)
```
The above program will produce the following files:

**file1.xml**

```xml
<?xml version='1.0' encoding='ASCII'?>
<Zoo>
    <DodoList>
        <Dodo name="momo" gender="male"/>
    </DodoList>
</Zoo>
```

**file2.xml**

```xml
<?xml version='1.0' encoding='ASCII'?>
<Zoo>
    <DodoList>
        <Dodo name="momo" gender="male"/>
    </DodoList>
</Zoo>
```

**file3.xml**

```xml
<?xml version='1.0' encoding='ASCII'?>
<Zoo>
    <DodoList/>
</Zoo>
```

**Related concepts**

- **om.DomNode.appendChild** on page 2587
  Adds an existing node at the end of the list of children in the current node.

- **om.DomNode.replaceChild** on page 2599
  Replaces a node by another in the child nodes of the current node.

**Syntax**

```plaintext
replaceChild(
    new om.DomNode,
    old om.DomNode)
```

1. *new* is a reference to the new node.
2. *old* is the node to be replaced.

**Usage**

The replaceChild() method puts the om.DomNode element passed as first parameter at the place of the node referenced by the second parameter, in the children list of the object node calling the method.

The new child node passed to the replaceChild() method must have been created from the same DOM document object, for example with the om.DomDocument.createElement() method.

The old node is not destroyed, if it is still referenced by a variable. The old node still exists in the DOM document, but it is an orphan node, that can be attached to another parent node in the document.
**Example**

```
MAIN
  DEFINE doc om.DomDocument,
         r om.DomNode,
         p om.DomNode,
         o om.DomNode,
         n om.DomNode

  LET doc = om.DomDocument.create("Items")
  LET r = doc.createElement("Zoo")
  LET p = doc.createElement("DodoList")
  CALL r.appendChild(p)
  LET o = doc.createElement("Dodo")
  CALL o.setAttribute("name", "momo")
  CALL o.setAttribute("gender", "male")
  CALL p.appendChild(o)
  CALL r.writeXml("file1.xml")
  LET n = doc.createElement("Dodo")
  CALL n.setAttribute("name", "kiki")
  CALL n.setAttribute("gender", "female")
  CALL p.replaceChild(n, o)
  -- o is orphan but still exists
  CALL o.writeXml("file2.xml")
  LET o = NULL -- unref/destroy the node
  CALL r.writeXml("file3.xml")
END MAIN
```

The above program will produce following files:

**file1.xml**

```xml
<?xml version='1.0' encoding='ASCII'?>
<Zoo>
  <DodoList>
    <Dodo name="momo" gender="male"/>
  </DodoList>
</Zoo>
```

**file2.xml**

```xml
<?xml version='1.0' encoding='ASCII'?>
<Dodo name="momo" gender="male"/>
```

**file3.xml**

```xml
<?xml version='1.0' encoding='ASCII'?>
<Zoo>
  <DodoList>
    <Dodo name="kiki" gender="female"/>
  </DodoList>
</Zoo>
```
Related concepts

**om.DomNode.appendChild** on page 2587
Adds an existing node at the end of the list of children in the current node.

**om.DomNode.removeChild** on page 2598
Deletes the specified child node from the current node.

**om.DomDocument.createElement** on page 2582
Create a new element node in the DOM document.

**om.DomNode.selectByPath**
Finds descendant DOM nodes from an XPath-like pattern.

Syntax

```java
selectByPath ( 
    path STRING )
RETURNS om.NodeList
```

1. *path* is an XPath-like pattern.

Usage

The `selectByPath()` method scans the DOM tree for descendant nodes from the specified XPath-like pattern.

**Important:** The `selectByPath()` method supports a limited XPath syntax.

The search pattern must always start with `/` or `//`, must contain at least one tag name, and the attributes expression allows only equality comparison.

The pattern supported by `selectByPath()` is limited to the following syntax:

```
<i> /</i> 
// TagName [ [@AttributeName="Value"] ] [...]
```

DOM node tag names and attributes names are case-sensitive.

The method creates a list of nodes as an `om.NodeList` object. This list object is then used to process the nodes found.
### Table 547: XPath examples

<table>
<thead>
<tr>
<th>XPath expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>/Vehicle</code></td>
<td>The document element <code>&lt;Vehicle/&gt;</code> of this document.</td>
</tr>
<tr>
<td><code>//Vehicle</code></td>
<td>All <code>&lt;Vehicle/&gt;</code> elements in the document.</td>
</tr>
<tr>
<td><code>//Transport/Vehicle</code></td>
<td>All <code>&lt;Vehicle/&gt;</code> elements that are children of <code>&lt;Transport/&gt;</code> elements.</td>
</tr>
<tr>
<td><code>//Transport//Engine</code></td>
<td>All <code>&lt;Engine/&gt;</code> elements that are arbitrary descendants of <code>&lt;Transport/&gt;</code> elements.</td>
</tr>
<tr>
<td><code>//Transport/*/Engine</code></td>
<td>All <code>&lt;Engine/&gt;</code> elements that are grandchildren of <code>&lt;Transport/&gt;</code> elements.</td>
</tr>
<tr>
<td><code>//Transport/*</code></td>
<td>All elements that are children of <code>&lt;Transport/&gt;</code> elements.</td>
</tr>
<tr>
<td><code>//Engine[@power=&quot;185&quot;]</code></td>
<td>All <code>&lt;Engine/&gt;</code> elements where the power attribute equals 185.</td>
</tr>
<tr>
<td><code>//Vehicle[@type=&quot;car&quot;]//Engine[@power=&quot;185&quot;]</code></td>
<td>All <code>&lt;Engine/&gt;</code> elements where the power attribute equals 185, children of <code>&lt;Vehicle/&gt;</code> elements where the type attribute is &quot;car&quot;.</td>
</tr>
</tbody>
</table>

### Example

```oml
DEFINE node om.DomNode,
    nodelist om.NodeList

LET nodelist = node.selectByPath("//Grid/Table[@tabName="t1"]")
```

For a complete example, see Example 2: Search nodes by XPath on page 2609.

### Related concepts

- **The NodeList class** on page 2607
- A `om.NodeList` object holds a list of DOM nodes.

- **om.DomNode.selectByTagName**
  Finds descendant DOM nodes based on a tag name.

### Syntax

```oml
selectByTagName( 
    tagName STRING ) 
RETURNS om.NodeList
```

1. `tagName` is a tag name for the search.

### Usage

The `selectByTagName()` method scans the DOM tree for descendant nodes defined with the tag name specified as parameter.
DOM node tag names are case-sensitive.

The method creates a list of nodes as a `om.NodeList` object. This list object is then used to process the nodes found.

**Example**

```oml
DEFINE node om.DomNode,
    nodelist om.NodeList
...
LET nodelist = node.selectByTagName("Car")
```

For a complete example, see Example 1: Search nodes by tag name on page 2608.

**Related concepts**
- The `NodeList` class  on page 2607
- A `om.NodeList` object hold a list of DOM nodes.

**om.DomNode.setAttribute**
Sets the value of a DOM node attribute.

**Syntax**

```oml
setAttribute (  
    name STRING,
    value STRING )
```

1. `name` is the name of the attribute.
2. `value` is the attribute value.

**Usage**

The `setAttribute()` method sets the value of an attribute in the current node.

DOM node attribute names are case-sensitive.

**Note:** Make sure that the strings passed to the method do not contain illegal XML characters. Illegal XML characters will be silently ignored. Illegal XML characters are any character below space (ASCII 32), except \r (ASCII 13), \n (ASCII 10) and \t (ASCII 9).

For character nodes (created for example with the `createChars()` of a `DomDocument` object), you can set the text value by passing the `@chars` attribute name to the method.

**Example**

```oml
DEFINE doc om.DomDocument  
DEFINE node om.DomNode     
DEFINE text_node om.DomNode
...
CALL node.setAttribute("name", "tiger")
...
LET text_node = doc.createChars("Initial text")
CALL text_node.setAttribute("@chars", "New text...")
CALL node.appendChild(text_node)
```

For a complete example, see Example 1: Creating a DOM tree on page 2605.

**Related concepts**
- `om.DomNode.getAttribute` on page 2590
  Returns the value of a DOM node attribute.
- `om.DomNode.getAttributeString` on page 2592
Returns the value of a DOM node attribute, with default string value.

`om.DomNode.getAttributeInteger` on page 2591

Returns the value of a DOM node attribute, with default integer value.

`om.DomNode.toString`

Serializes the current node into an XML formatted string.

**Syntax**

```
toString()  
RETURNS STRING
```

**Usage**

The `toString()` method builds an XML formatted string with the DOM structure of the current node and returns the string.

**Example**

```
DEFINE node om.DomNode, s STRING
...
LET s = node.toString()
```

**Related concepts**

- `om.DomNode.parse` on page 2590
  Parses an XML formatted string and creates the DOM structure in the current node.

- `om.DomNode.write`
  Processes a DOM document with a SAX document handler.

**Syntax**

```
write(  
  sdh om.SaxDocumentHandler )
```

1. `sdh` references a SAX document handler.

**Usage**

The `write()` method processes the current DOM node content with the SAX document handler passed as parameter.

See the SAX document handler class for more details.

**Related concepts**

- The `SaxDocumentHandler class` on page 2617
  The `om.SaxDocumentHandler` class provides an interface to write an XML filter with events.

- `om.DomNode.writeXml`
  Creates an XML file from the current DOM node.

**Syntax**

```
writeXml(  
  path STRING )
```

1. `path` is the path to the XML file.
**Usage**

The `writeXml()` method writes the content of the current DOM node to the file passed as parameter.

**Example**

```
DEFINE node om.DomNode
...
CALL noe.writeXml("output.xml")
```

**Related concepts**

- `om.DomNode.loadXml` on page 2589
  Load an XML file into the current node.

**Examples**

- `om.DomNode` usage examples.

**Example 1: Creating a DOM tree**

To create a DOM tree with the following structure (represented in XML format):

```
<Vehicles>
  <Car name="Corolla" color="Blue" weight="1546">Nice car!</Car>
  <Bus name="Maxibus" color="Yellow" weight="5278">
    <Wheel width="315" diameter="925" />
    <Wheel width="315" diameter="925" />
    <Wheel width="315" diameter="925" />
    <Wheel width="315" diameter="925" />
  </Bus>
</Vehicles>
```

You write the following:

```
MAIN
DEFINE d  om.DomDocument
DEFINE r, n, t, w om.DomNode
DEFINE i INTEGER

LET d = om.DomDocument.create("Vehicles")
LET r = d.getDocumentElement()

LET n = r.createChild("Car")
CALL n.setAttribute("name","Corolla")
CALL n.setAttribute("color","Blue")
CALL n.setAttribute("weight","1546")

LET t = d.createChars("Nice car!")
CALL n.appendChild(t)
LET t = d.createEntity("nbsp")
CALL n.appendChild(t)
LET t = d.createChars("Yes, very nice!")
CALL n.appendChild(t)

LET n = r.createChild("Bus")
CALL n.setAttribute("name","Maxibus")
CALL n.setAttribute("color","Yellow")
CALL n.setAttribute("weight","5278")
FOR i=1 TO 4
  LET w = n.createChild("Wheel")
  CALL w.setAttribute("width","315")
  CALL w.setAttribute("diameter","925")
END FOR

CALL r.writeXml("Vehicles.xml")
```
**Example 2: Displaying a DOM tree recursively**

The following example displays a DOM tree content recursively:

```plaintext
FUNCTION displayDomNode(n, e)
    DEFINE n om.DomNode
    DEFINE e, i, s INTEGER

    LET s = e*2
    DISPLAY s SPACES || "Tag: " || n.getTagName()

    DISPLAY s SPACES || "Attributes:"
    FOR i=1 TO n.getAttributesCount()
        DISPLAY s SPACES || "  " || n.getAttributeName(i) || 
        
        "=" || n.getAttributeValue(i) || ""
    END FOR

    LET n = n.getFirstChild()

    DISPLAY s SPACES || "Child Nodes:"
    WHILE n IS NOT NULL
        CALL displayDomNode(n, e+1)
        LET n = n.getNext()
    END WHILE

END FUNCTION
```

**Example 3: Writing a DOM tree to a SAX handler**

The following example outputs a DOM tree without indentation.

```plaintext
MAIN
    DEFINE d om.DomDocument
    DEFINE r, n, t, w om.DomNode
    DEFINE dh om.SaxDocumentHandler
    DEFINE i INTEGER

    LET dh = om.XmlWriter.createPipeWriter("cat")
    CALL dh.setIndent(FALSE)

    LET d = om.DomDocument.create("Vehicles")
    LET r = d.getDocumentElement()

    LET n = r.createChild("Car")
    CALL n.setAttribute("name","Corolla")
    CALL n.setAttribute("color","Blue")
    CALL n.setAttribute("weight","1546")

    LET t = d.createChars("Nice car!")
    CALL n.appendChild(t)

    LET n = r.createChild("Bus")
    CALL n.setAttribute("name","Maxibus")
    CALL n.setAttribute("color","yellow")
    CALL n.setAttribute("weight","5278")
    FOR i=1 TO 4
        LET w = n.createChild("Wheel")
        CALL w.setAttribute("width","315")
        CALL w.setAttribute("diameter","925")
    END FOR
```
CALL r.write(dh)
END MAIN

The NodeList class
An om.NodeList object holds a list of DOM nodes.
The list is created from an om.DomNode.selectByTagName() or om.DomNode.selectByPath() method.
After creating the node list, you can process the nodes with the getLength() and item() methods of the om.NodeList object.

om.NodeList methods
Methods of the om.NodeList class.

Table 548: Object methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getLength()</td>
<td>RETURNS INTEGER Returns the number of elements in the node list.</td>
</tr>
<tr>
<td>item(index INTEGER)</td>
<td>RETURNS om.DomNode Returns a DOM node element by position in the node list.</td>
</tr>
</tbody>
</table>

om.NodeList.getLength
Returns the number of elements in the node list.

Syntax

gLength()

RETURNS INTEGER

1. node is a reference to a node.

Usage

The getLength() method returns the size of the node list.
Query the node list for elements with the item() method, in the range 1 to getLength().

Example

DEFINE list om.NodeList
... DISPLAY list.getLength()

For a complete example, see Example 1: Search nodes by tag name on page 2608.

Related concepts
om.DomNode.selectByTagName on page 2602
Finds descendant DOM nodes based on a tag name.
om.DomNode.selectByPath on page 2601
Finds descendant DOM nodes from an XPath-like pattern.

**om.NodeList.item**

Returns a DOM node element by position in the node list.

### Syntax

```plaintext
item( index INTEGER )
RETURNS om.DomNode
```

1. `index` is the ordinal position of the node in the list.

### Usage

The `item()` method returns the `om.DomNode` object at the position specified.

First element is at position 1.

If there is no element at the specified index, the method returns `NULL`.

### Example

```plaintext
DEFINE list om.NodeList, 
      node om.DomNode 
...
LET node = list.item(12)
```

For a complete example, see Example 1: Search nodes by tag name on page 2608.

### Related concepts

- `om.DomNode.selectByTagName` on page 2602
  Finds descendant DOM nodes based on a tag name.
- `om.DomNode.selectByPath` on page 2601
  Finds descendant DOM nodes from an XPath-like pattern.

### Examples

- `om.NodeList` usage examples.

**Example 1: Search nodes by tag name**

XML file: `Vehicles.xml`

```xml
<?xml version='1.0' encoding='ASCII'?>
<Vehicles>
  <Vehicle type="car" name="DeLorean DMC-12" year="1983" color="Gray" weight="1230">
    <Engine energy="petrol" size="2849" power="185" />
    <Wheels count="4" width="280" diameter="550" />
  </Vehicle>
  <Vehicle type="car" name="Corolla" year="2003" color="Blue" weight="1146">
    <Engine energy="gasoline" size="1200" power="75" />
    <Wheels count="4" width="220" diameter="525" />
  </Vehicle>
  <Vehicle type="bus" name="Maxibus" year="1998" color="Yellow" weight="5278">
    <Engine energy="diesel" size="4100" power="445" />
    <Wheels count="6" width="315" diameter="925" />
  </Vehicle>
</Vehicles>
```
Program file:

```plaintext
MAIN
DEFINE d om.DomDocument
DEFINE nl om.NodeList
DEFINE r, n om.DomNode
DEFINE i INTEGER
LET d = om.DomDocument.createFromXmlFile("Vehicles.xml")
LET r = d.getDocumentElement()
LET nl = r.selectByTagName("Vehicle")
FOR i = 1 TO nl.getLength()
    LET n = nl.item(i)
    DISPLAY n.getAttribute("name")
END FOR
END MAIN
```

Output:

DeLorean DMC-12
Corolla
Maxibus

Related concepts

`om.DomNode.selectByTagName` on page 2602

Finds descendant DOM nodes based on a tag name.

*Example 2: Search nodes by XPath*

**XML file: Vehicles.xml**

```
<?xml version='1.0' encoding='ASCII'?>
<Vehicles>
    <Vehicle type="car" name="DeLorean DMC-12" year="1983" color="Gray"
        weight="1230">
        <Engine energy="petrol" size="2849" power="185" />
        <Wheels count="4" width="280" diameter="550" />
    </Vehicle>
    <Vehicle type="car" name="Corolla" year="2003" color="Blue"
        weight="1146">
        <Engine energy="gasoline" size="1200" power="75" />
        <Wheels count="4" width="220" diameter="525" />
    </Vehicle>
    <Vehicle type="bus" name="Maxibus" year="1998" color="Yellow"
        weight="5278">
        <Engine energy="diesel" size="4100" power="445" />
        <Wheels count="6" width="315" diameter="925" />
    </Vehicle>
</Vehicles>
```

Program file:

```plaintext
MAIN
DEFINE d om.DomDocument
DEFINE nl om.NodeList
DEFINE r, n om.DomNode
DEFINE i INTEGER
LET d = om.DomDocument.createFromXmlFile("Vehicles.xml")
LET r = d.getDocumentElement()
LET nl = r.selectByPath("//Vehicle//Wheels[@count="4"]")
```
FOR i = 1 TO nl.getLength()
    LET n = nl.item(i)
    DISPLAY n.getParent().getAttribute("name")
END FOR
END MAIN

Output:
DeLorean DMC-12
Corolla

Related concepts
om.DomNode.selectByPath on page 2601
Finds descendant DOM nodes from an XPath-like pattern.

The SaxAttributes class
The om.SaxAttributes class holds a set of attributes to process with a SAX reader or writer.
To process SAX attributes, create a om.SaxAttributes object with a SAX reader or SAX writer object.
Get an instance of SaxAttributes with the om.XmlReader.getAttributes() method.

om.SaxAttributes methods
Methods of the om.SaxAttributes class.

Table 549: Class methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy(attr om.SaxAttributes)</td>
<td>Clones an existing SAX attributes object.</td>
</tr>
<tr>
<td>create()</td>
<td>Create a new SAX attributes object.</td>
</tr>
<tr>
<td></td>
<td>RETURNS om.SaxAttributes</td>
</tr>
<tr>
<td></td>
<td>RETURNS om.SaxAttributes</td>
</tr>
</tbody>
</table>
### Table 550: Object methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addAttribute(name STRING, value STRING)</code></td>
<td>Appends a new attribute to the end of the list.</td>
</tr>
<tr>
<td><code>clear()</code></td>
<td>Clears the SAX attribute list.</td>
</tr>
<tr>
<td><code>getLength()</code></td>
<td>Returns the number of attributes in the list.</td>
</tr>
<tr>
<td><code>getName(index INTEGER)</code></td>
<td>Returns the name of an attribute by position.</td>
</tr>
<tr>
<td><code>getValue(name STRING)</code></td>
<td>Returns the value of an attribute by name.</td>
</tr>
<tr>
<td><code>getValueByIndex(index INTEGER)</code></td>
<td>Returns an attribute value by position.</td>
</tr>
<tr>
<td><code>removeAttribute(index INTEGER)</code></td>
<td>Delete an attribute by position.</td>
</tr>
<tr>
<td><code>setAttributes(attr om.SaxAttributes)</code></td>
<td>Clears the list and copies the attributes passed.</td>
</tr>
</tbody>
</table>

**om.SaxAttributes.addAttribute**  
Appends a new attribute to the end of the list.

**Syntax**

```java
addAttribute(
    name STRING,
    value STRING )
```

1. `name` is the name of the attribute.
2. `value` is the value of the attribute.

**Usage**

The `addAttribute()` method appends a new attribute with name and value at the end of the list.  
Attribute names are case-sensitive.
Note: Make sure that the strings passed to the method do not contain illegal XML characters. Illegal XML characters will be silently ignored. Illegal XML characters are any character below space (ASCII 32), except \r (ASCII 13), \n (ASCII 10) and \t (ASCII 9).

Example

```
DEFINE attrs om.SaxAttributes
   ...
   CALL attrs.addAttribute("name","jo")
```

For a complete example, see Example 2: Creating a SAX attributes object on page 2616.

Related concepts

om.SaxAttributes.setAttributes on page 2616
Clears the list and copies the attributes passed.

om.SaxAttributes.copy
Clones an existing SAX attributes object.

Syntax

```
copy ( attr om.SaxAttributes )
RETURNS om.SaxAttributes
```

1. `attr` is a set of SAX attributes to clone.

Usage

The `om.SaxAttributes.copy()` class method makes a clone of the `om.SaxAttributes` object passed as reference and returns the created object.

Example

```
DEFINE copy, orig om.SaxAttributes
   ...
   LET copy = om.SaxAttributes.copy(orig)
```

For a complete example, see Example 2: Creating a SAX attributes object on page 2616.

Related concepts

om.SaxAttributes.create on page 2612
Create a new SAX attributes object.

om.SaxAttributes.create
Create a new SAX attributes object.

Syntax

```
create ()
RETURNS om.SaxAttributes
```

Usage

The `om.SaxAttributes.create()` class method creates a new `om.SaxAttributes` object and returns it.

To hold the reference to a SAX attributes object, define a variable with the type `om.SaxAttributes` type.
Example

DEFINE attrs om.SaxAttributes

... LET attrs = om.SaxAttributes.create()

For a complete example, see Example 2: Creating a SAX attributes object on page 2616.

Related concepts

- **om.SaxAttributes.copy** on page 2612
  Clones an existing SAX attributes object.

- **om.SaxAttributes.clear**
  Clears the SAX attribute list.

Syntax

```
clear()
```

Usage

Use the `clear()` method to clean the SAX attribute list.

Related concepts

- **om.SaxAttributes.addAttribute** on page 2611
  Appends a new attribute to the end of the list.

- **om.SaxAttributes.getLength**
  Returns the number of attributes in the list.

Syntax

```
getLength()
   RETURNS INTEGER
```

Usage

The `getLength()` method returns the number of attributes in the current SAX attribute list.

Use this method with `getName()` and `getValueByIndex()` to retrieve attributes by position.

Example

```
DEFINE attrs om.SaxAttributes,
    index INTEGER

... FOR index = 1 TO attrs.getLength()
    DISPLAY attrs.getName(index), " = ", attrs.getValueByIndex(index)
END FOR
```

Related concepts

- **om.SaxAttributes.getName** on page 2614
  Returns the name of an attribute by position.

- **om.SaxAttributes.getValueByIndex** on page 2615
Returns an attribute value by position.

`om.SaxAttributes.getName`

Returns the name of an attribute by position.

**Syntax**

```plaintext
getName(
   index  INTEGER  )
RETURNS  STRING
```

1. `index` is the position of the attribute in the list.

**Usage**

The `getName()` method returns the name of the attribute at the specified ordinal position in the list.

If the attribute does not exist at the given position, the method returns `NULL`.

**Example**

```plaintext
DEFINE attrs om.SaxAttributes
...
DISPLAY attrs.getName(3)
```

For a complete example, see Example 2: Creating a SAX attributes object on page 2616.

**Related concepts**

- `om.SaxAttributes.getLength` on page 2613
- `om.SaxAttributes.getValueByIndex` on page 2615

`om.SaxAttributes.getValue`

Returns the value of an attribute by name.

**Syntax**

```plaintext
getValue(
   name  STRING  )
RETURNS  STRING
```

1. `name` is the name of an attribute.

**Usage**

The `getValue()` method returns the value of the attribute identified by the name passed as parameter.

If the attribute does not exist, the method returns `NULL`.

**Example**

```plaintext
DEFINE attrs om.SaxAttributes
...
DISPLAY attrs.getValue("name")
```

For a complete example, see Example 2: Creating a SAX attributes object on page 2616.

**Related concepts**

- `om.SaxAttributes.getValueByIndex` on page 2615
Returns an attribute value by position.

\textit{om.SaxAttributes.getValueByIndex}

Returns an attribute value by position.

**Syntax**

```
getValueByIndex(
    index INTEGER
) RETURNS STRING
```

1. \textit{index} is the position of the attribute in the list.

**Usage**

The \texttt{getValueByIndex()} method returns the value of the attribute at the specified ordinal position in the list.

If the attribute does not exist at the given position, the method returns \texttt{NULL}.

**Example**

```
DEFINE attrs om.SaxAttributes
...
DISPLAY attrs.getValueByIndex(3)
```

For a complete example, see Example 2: Creating a SAX attributes object on page 2616.

**Related concepts**

\textit{om.SaxAttributes.getLength} on page 2613

Returns the number of attributes in the list.

\textit{om.SaxAttributes.getName} on page 2614

Returns the name of an attribute by position.

\textit{om.SaxAttributes.removeAttribute}

Delete an attribute by position.

**Syntax**

```
removeAttribute(
    index INTEGER
)
```

1. \textit{index} is the position of the attribute in the list.

**Usage**

The \texttt{removeAttribute()} method removes the attribute at the given ordinal position.

If the attribute does not exist at the given position, the method returns silently.

**Example**

```
DEFINE attrs om.SaxAttributes
...
CALL attrs.removeAttribute( attrs.getLength() )
```

For a complete example, see Example 2: Creating a SAX attributes object on page 2616.

**Related concepts**

\textit{om.SaxAttributes.addAttribute} on page 2611
Appends a new attribute to the end of the list.

**om.SaxAttributes.setAttributes**

Clears the list and copies the attributes passed.

### Syntax

```plaintext
setAttributes(
    attr om.SaxAttributes
)
```

1. `attr` is a reference to a list of attributes.

### Usage

The `setAttributes()` method takes an existing `om.SaxAttributes` object reference and makes a copy of all attributes into the current attribute list.

### Example

```plaintext
DEFINE curr, orig om.SaxAttributes
...
CALL curr.setAttributes(orig)
```

For a complete example, see [Example 2: Creating a SAX attributes object](#) on page 2616.

### Related concepts

- **om.SaxAttributes.addAttribute** on page 2611
  
Appends a new attribute to the end of the list.

### Examples

- **om.SaxAttributes usage examples.**
  
#### Example 1: Displaying SAX attributes of an XML node

```plaintext
FUNCTION displayAttributes( a )
    DEFINE a om.SaxAttributes
    DEFINE i, m INTEGER
    LET m = a.getLength()
    FOR i=1 to m
        DISPLAY a.getName(i) || "=" || a.getValueByIndex(i) || "]"
    END FOR
END FUNCTION
```

#### Example 2: Creating a SAX attributes object

```plaintext
MAIN
    DEFINE a, c om.SaxAttributes,
        i, m INTEGER
    LET a = om.SaxAttributes.create()
    CALL a.addAttribute("name","Jo")
    CALL a.addAttribute("birth","2008-12-20")
    DISPLAY a.getValue("birth")
    LET c = om.SaxAttributes.copy( a )
    LET m = a.getLength()
    DISPLAY "Len = ", m
    FOR i=1 TO m
        DISPLAY i, ": ", a.getName(i),
        " = ", a.getValueByIndex(i)
    END FOR
    CALL a.removeAttribute("birth")
    CALL c.setAttributes(a)
    CALL a.clear()
```
The SaxDocumentHandler class
The om.SaxDocumentHandler class provides an interface to write an XML filter with events.
This class follows the SAX standards.

A om.SaxDocumentHandler object can be used in two different ways:

1. To implement an XML SAX filter, based on functions defined in a .4gl module, by using the
   createForName() class method.
2. To write an XML document to a file, process or socket output, by using om.XmlWriter creation methods, and
   the om.SaxDocumentHandler processing methods.

The om.SaxDocumentHandler class also provides methods to process all SAX events by hand. This is useful if
you want to chain SAX filters.

om.SaxDocumentHandler methods
Methods of the om.SaxDocumentHandler class.

Table 551: Class methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>om.SaxDocumentHandler.createForName</td>
<td>Creates a new SAX document handler object for the given .4gl module.</td>
</tr>
<tr>
<td>moduleName STRING</td>
<td>RETURNS om.SaxDocumentHandler</td>
</tr>
</tbody>
</table>
### Table 552: Object methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>characters</strong></td>
<td>Processes a text node.</td>
</tr>
<tr>
<td>(chars STRING)</td>
<td></td>
</tr>
<tr>
<td><strong>endDocument</strong></td>
<td>Processes the end of the document.</td>
</tr>
<tr>
<td>()</td>
<td></td>
</tr>
<tr>
<td><strong>endElement</strong></td>
<td>Processes the end of an element.</td>
</tr>
<tr>
<td>(name STRING)</td>
<td></td>
</tr>
<tr>
<td><strong>processingInstruction</strong></td>
<td>Processes a processing instruction.</td>
</tr>
<tr>
<td>(name STRING, data STRING)</td>
<td></td>
</tr>
<tr>
<td><strong>readXmlFile</strong></td>
<td>Reads and processes an XML file with the SAX document handler.</td>
</tr>
<tr>
<td>(path STRING)</td>
<td></td>
</tr>
<tr>
<td><strong>setIndent</strong></td>
<td>Controls indentation in XML output.</td>
</tr>
<tr>
<td>(indenting BOOLEAN)</td>
<td></td>
</tr>
<tr>
<td><strong>startDocument</strong></td>
<td>Processes the beginning of the document.</td>
</tr>
<tr>
<td>()</td>
<td></td>
</tr>
<tr>
<td><strong>startElement</strong></td>
<td>Processes the beginning of an element.</td>
</tr>
<tr>
<td>(name STRING, atts om.SaxAttributes)</td>
<td></td>
</tr>
<tr>
<td><strong>skippedEntity</strong></td>
<td>Processes an unresolved entity.</td>
</tr>
<tr>
<td>(name STRING)</td>
<td></td>
</tr>
</tbody>
</table>

---

**om.SaxDocumentHandler.createForName**

Creates a new SAX document handler object for the given .4gl module.

#### Syntax

```
om.SaxDocumentHandler.createForName(
    moduleName STRING)
RETURNS om.SaxDocumentHandler
```

1. `moduleName` is the name of the .4gl module defining the document handler events.

#### Usage

The `om.SaxDocumentHandler.createForName()` method creates an `om.SaxDocumentHandler` instance and binds the .42m module passed as argument to the object.

To hold the reference to a SAX document handler object, define a variable with the `om.SaxDocumentHandler` type.
The module must be available as a compiled .42m file, which is loadable based on the environment settings (FGLLDPATH).

The module must implement the following functions to process the SAX filter events:

**Table 553: Functions of the SAX document handler module**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>startDocument()</code></td>
<td>Called once at the beginning of the document processing.</td>
</tr>
<tr>
<td><code>endDocument()</code></td>
<td>Called once at the end of the document processing.</td>
</tr>
<tr>
<td><code>startElement( tagname STRING, attrs om.SaxAttributes )</code></td>
<td>Called when an XML element is reached. Use the om.SaxAttributes methods to handle the attributes of the processed element.</td>
</tr>
<tr>
<td>1. <em>tagname</em> is the tag name of element.</td>
<td></td>
</tr>
<tr>
<td>2. <em>attrs</em> is list of attributes.</td>
<td></td>
</tr>
<tr>
<td><code>endElement( tagname STRING )</code></td>
<td>Called when the end of an XML element is reached.</td>
</tr>
<tr>
<td>1. <em>tagname</em> is the tag name of element.</td>
<td></td>
</tr>
<tr>
<td><code>processingInstruction( piname STRING, data STRING )</code></td>
<td>Called when a processing instruction is reached.</td>
</tr>
<tr>
<td>1. <em>piname</em> is the name of the processing instruction.</td>
<td></td>
</tr>
<tr>
<td>2. <em>data</em> is the content of the processing instruction.</td>
<td></td>
</tr>
<tr>
<td><code>characters( data STRING )</code></td>
<td>Called when a text node is reached.</td>
</tr>
<tr>
<td>1. <em>data</em> is the text data.</td>
<td></td>
</tr>
<tr>
<td><code>skippedEntity( name STRING )</code></td>
<td>Called when an unknown entity node is reached (like &amp;xxx; for example).</td>
</tr>
<tr>
<td>1. <em>name</em> is the name of the unknown entity.</td>
<td></td>
</tr>
</tbody>
</table>

**Example**

```plaintext
DEFINE f om.SaxDocumentHandler
LET f = om.SaxDocumentHandler.createForName("mysaxmod")
...
```

For a complete example, see Example 1: Extracting phone numbers from a directory on page 2623.

**Related concepts**
The SaxAttributes class on page 2610

The om.SaxAttributes class holds a set of attributes to process with a SAX reader or writer.

FGLLDPATH on page 278
Defines the search paths to load program modules.

*om.SaxDocumentHandler.characters*

Processes a text node.

**Syntax**

```
class om.SaxDocumentHandler

characters(
    chars STRING )
```

1. `chars` is the content of the text node.

**Usage**

The `characters()` method processes a text node with the SAX interface.

**Note:** Make sure that the strings passed to the method do not contain illegal XML characters. Illegal XML characters will be silently ignored. Illegal XML characters are any character below space (ASCII 32), except \r (ASCII 13), \n (ASCII 10) and \t (ASCII 9).

**Related concepts**

*om.SaxDocumentHandler.skippedEntity* on page 2623

Processes an unresolved entity.

*om.SaxDocumentHandler.endDocument*

Processes the end of the document.

**Syntax**

```
class om.SaxDocumentHandler

endDocument()
```

**Usage**

The `endDocument()` method ends the document processing with the SAX interface.

**Related concepts**

*om.SaxDocumentHandler.startDocument* on page 2622

Processes the beginning of the document.

*om.SaxDocumentHandler.endElement*

Processes the end of an element.

**Syntax**

```
class om.SaxDocumentHandler

endElement(
    name STRING )
```

1. `name` is the tag name of the element.

**Usage**

The `endElement()` method processes the end of an element with the SAX interface.

**Related concepts**

*om.SaxDocumentHandler.startElement* on page 2622
Processes the beginning of an element.

`om.SaxDocumentHandler.processingInstruction`
Processes a processing instruction.

**Syntax**

```java
processingInstruction(
    name STRING,
    data STRING )
```

1. *name* is the name of the processing instruction (token after `<?`).
2. *data* is the string in the processing instruction tag.

**Usage**

The `processingInstruction()` method processes a processing instruction with the SAX interface.

A processing instruction appears in an XML formatted text as:

```xml
<?name data ?>
```

**Related concepts**

- The `SaxAttributes` class on page 2610
  The `om.SaxAttributes` class holds a set of attributes to process with a SAX reader or writer.

- `om.SaxDocumentHandler.readXmlFile`
  Reads and processes an XML file with the SAX document handler.

**Syntax**

```java
readXmlFile(
    path STRING )
```

1. *path* is the path to an XML formatted file.

**Usage**

Use the `readXmlFile()` method after creating the `om.SaxDocumentHandler` object, to process the XML data from a file input stream.

**Example**

```java
DEFINE f om.SaxDocumentHandler
LET f = om.SaxDocumentHandler.createForName("mysaxmod")
CALL f.readXmlFile("cars.xml")
...
```

For a complete example, see Example 1: Extracting phone numbers from a directory on page 2623.

**Related concepts**

- The `SaxAttributes` class on page 2610
  The `om.SaxAttributes` class holds a set of attributes to process with a SAX reader or writer.

- `FGLLLDPATH` on page 278
Defines the search paths to load program modules.

*om.SaxDocumentHandler.setIndent*
Controls indentation in XML output.

**Syntax**

```java
setIndent (indenting BOOLEAN)
```

1. `indenting`: TRUE enables indentation; FALSE disables indentation.

**Usage**

By default, the *om.SaxDocumentHandler* object outputs XML with indentation.

In order to disable indentation, use the `setIndent(FALSE)` method.

*om.SaxDocumentHandler.startDocument*
Processes the beginning of the document.

**Syntax**

```java
startDocument()
```

**Usage**

The `startDocument()` method begins the document processing with the SAX interface.

**Related concepts**

*om.SaxDocumentHandler.endDocument* on page 2620
Processes the end of the document.

*om.SaxDocumentHandler.startElement*
Processes the beginning of an element.

**Syntax**

```java
startElement (name STRING, atts om.SaxAttributes)
```

1. `name` is the tag name of the element.
2. `atts` is the list of attributes of the element.

**Usage**

The `startElement()` method processes the beginning of an element with the SAX interface.

Use the `om.SaxAttributes` methods to handle the attributes of an element.

**Example**

```java
DEFINE out om.SaxDocumentHandler
node om.DomNode,
atts om.SaxAttributes,
x, c INTEGER
...
CALL atts.clear()
LET c = node.getChildCount()
```
FOR x=1 TO c
    CALL attrs.addAttribute( node.getAttributeName(x), node.getAttributeValue(x) )
END FOR
CALL out.startElement( node.getTag_Name(), attrs )

Related concepts
om.SaxDocumentHandler.endElement on page 2620
Processes the end of an element.

The SaxAttributes class on page 2610
The om.SaxAttributes class holds a set of attributes to process with a SAX reader or writer.

om.SaxDocumentHandler.skippedEntity
Processes an unresolved entity.

Syntax

skippedEntity( name STRING )

1. name is the name of the unknown entity.

Usage
The skippedEntity() method processes an entity that is not known by the XML parser.

Related concepts
om.SaxDocumentHandler.characters on page 2620
Processes a text node.

Examples
om.SaxDocumentHandler usage examples.
Example 1: Extracting phone numbers from a directory
This example shows how to write a SAX filter to extract phone numbers from a directory file written in XML.

MAIN
    DEFINE f om.SaxDocumentHandler
    LET f = om.SaxDocumentHandler.createForName("module1")
    CALL f.readXmlFile("customers.xml")
END MAIN

Note: The parameter of the createForName() method specifies the name of a source file that has been compiled into a .42m file ("module1.42m" in our example).

The module module1.4gl:

FUNCTION startDocument ()
END FUNCTION

FUNCTION processingInstruction(name,data)
    DEFINE name,data STRING
END FUNCTION

FUNCTION startElement(name,attr)
    DEFINE name STRING
    DEFINE attr om.SaxAttributes
    DEFINE i INTEGER
    CASE name
        WHEN "Customer"
The XML file customers.xml:

```xml
<Customers>
  <Customer customer_num="101" fname="Ludwig" lname="Pauli"
    company="All Sports Supplies" address1="213 Erstwild Court"
    address2="" city="Sunnyvale" state="CA" zip-code="94086">
    <CellPhone number="408-789-8075" />
    <WorkPhone number="873-123-4543" />
  </Customer>
  <Customer customer_num="102" fname="Carole" lname="Sadler"
    company="Sports Spot" address1="785 Geary St"
    address2="" city="San Francisco" state="CA" zip-code="94117">
    <CellPhone number="415-822-1289" />
    <WorkPhone number="834-842-8373" />
  </Customer>
  <Customer customer_num="103" fname="Philip" lname="Currie"
    company="Phil's Sports" address1="654 Poplar"
    address2="P. O. Box 3498" city="Palo Alto" state="CA"
    zip-code="94303">
    <CellPhone number="415-328-4543" />
    <WorkPhone number="932-118-4824" />
  </Customer>
</Customers>
```

Output:

Pauli Ludwig
   Cell phone: 408-789-8075
   Work phone: 873-123-4543
Sadler Carole
   Cell phone: 415-822-1289
   Work phone: 834-842-8373
Currie Philip
   Cell phone: 415-328-4543
   Work phone: 932-118-4824
Example 2: Producing an XML file with om.SaxDocumentHandler

This example shows how to write an XML file by using a SAX document handler object created from the om.XmlWriter.createFileWriter() method:

```
MAIN
    DEFINE w om.SaxDocumentHandler
    DEFINE a om.SaxAttributes

    LET a = om.SaxAttributes.create()

    LET w = om.XmlWriter.createFileWriter("output.xml")
    CALL w.startDocument()
    CALL w.startElement("Foo", NULL)
    CALL w.processingInstruction("PI2", "val")

    CALL w.startElement("Foo1", NULL)
    CALL w.characters("foo bar")
    CALL w.skippedEntity("nbsp")
    CALL w.EndElement("Foo1")

    CALL w.endElement("Foo")
    CALL w.endDocument()
END MAIN
```

The XmlReader class

The om.XmlReader class provides methods to read and process a file written in XML format.

The processing of the XML file is streamed-data based; the file is loaded and processed sequentially with events. To process XML element attributes, an om.XmlReader object must cooperate with a om.SaxAttributes object. The XmlReader class can only read from a file. To write to a file, use the om.XmlWriter class.

Steps to use a XML reader:

1. Declare a variable with the om.XmlReader type.
2. Create the reader object with the createFileReader() method and assign the reference to the variable.
3. Process SAX events in a WHILE loop, by reading document fragments with the read() method.
4. Inside the loop, depending on the SAX event, process element attributes with getAttributes() or get the element data with the getCharacters() methods.

om.XmlReader methods

Methods of the om.XmlReader class.

Table 554: Class methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>om.XmlReader.createFileReader(path STRING) RETURNS om.XmlReader</td>
<td>Creates an XML reader object from a file.</td>
</tr>
</tbody>
</table>
Table 555: Object methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| getCharacters()     | RETURNS STRING  
 Returns the character data of the current processed element. |
| getAttributes()     | RETURNS om.SaxAttributes  
 Builds an attribute list for the current processed element. |
| getTagName()        | RETURNS STRING  
 Returns the tag name of the current processed element. |
| read()              | RETURNS STRING  
 Read the next SAX event to process. |
| skippedEntity()     | RETURNS STRING  
 Returns the name of an unresolved entity. |

*om.XmlReader.createFileReader*

Creates an XML reader object from a file.

**Syntax**

```plaintext
om.XmlReader.createFileReader(path STRING )  
RETURNS om.XmlReader
```

1. `path` is the path to an XML formatted file.

**Usage**

Use the `om.XmlReader.createFileReader()` method to create a new `om.XmlReader` object, to process the XML data from a file input stream.

To hold the reference to an XmlReader object, define a variable with the `om.XmlReader` type.

**Note:** Files encoded in UTF-8 can start with the UTF-8 Byte Order Mark (BOM), a sequence of `0xEF 0xBB 0xBF` bytes, also known as UNICODE U+FEFF. When reading files, Genero BDL will ignore the UTF-8 BOM, if it is present at the beginning of the file. This applies to instructions such as `LOAD`, as well as I/O APIs such as `base.Channel.read()` and `readLine()`.

**Example**

```plaintext
DEFINE r om.XmlReader  
LET r = om.XmlReader.createFileReader("cars.xml")  
...  
```

For a complete example, see Example 1: Parsing an XML file on page 2630.

**Related concepts**

*om.XmlReader.read* on page 2628
Reads the next SAX event to process.

*om.XmlReader.getAttributes*

Builds an attribute list for the current processed element.

**Syntax**

```plaintext
getAttributes()
RETURNS om.SaxAttributes
```

**Usage**

Use the `getAttributes()` method to create a list of attributes of an `om.SaxAttributes` object, from the current processed element, in the `StartElement` or `EndElement` event context.

Declare a variable with the `om.SaxAttributes` type to reference the attribute list.

Note that once created with the `getAttributes()` method, the `om.SaxAttributes` object is automatically updated based on the element currently processed by the `om.XmlReader`.

**Example**

```plaintext
DEFINE r om.XmlReader,
    e STRING, i INT
    a om.SaxAttributes
...
LET e = r.read()
WHILE e IS NOT NULL
    CASE e
        ...
        WHEN "StartElement"
            LET a = r.getAttributes()
            FOR i=1 to a.getLength()
                ...
    ...
```

For a complete example, see Example 1: Parsing an XML file on page 2630.

**Related concepts**

The `SaxAttributes` class on page 2610

The `om.SaxAttributes` class holds a set of attributes to process with a SAX reader or writer.

*om.XmlReader.getCharacters*

Returns the character data of the current processed element.

**Syntax**

```plaintext
getCharacters()
RETURNS STRING
```

**Usage**

Use the `getCharacters()` method to get the character data of the current processed element, in the Characters event context.

**Example**

```plaintext
DEFINE r om.XmlReader,
    e STRING
...
LET e = r.read()
```
WHILE e IS NOT NULL
  CASE e
    ...
    WHEN "Characters"
      DISPLAY "Characters: ", r.getCharacters(), ""
    ...
  END

For a complete example, see Example 1: Parsing an XML file on page 2630.

`om.XmlReader.getTagName`

Returns the tag name of the current processed element.

**Syntax**

```plaintext```
getTagName()
  RETURNS STRING
```

**Usage**

Use the `readXmlFile()` method to get the tag name of the current processed element, in the `StartElement` or `EndElement` event context.

**Example**

```plaintext```
DEFINE r om.XmlReader,
  e STRING
  ...
  LET e = r.read()
  WHILE e IS NOT NULL
    CASE e
      ...
      WHEN "StartElement"
        DISPLAY "TagName = ", r.getTagName()
    ...
  END
```

For a complete example, see Example 1: Parsing an XML file on page 2630.

`om.XmlReader.read`

Reads the next SAX event to process.

**Syntax**

```plaintext```
read()
  RETURNS STRING
```

**Usage**

The `read()` method reads the next XML fragment and returns the name of the SAX event to process.
### Table 556: Events that can be returned by the `read()` method

<table>
<thead>
<tr>
<th>Event name</th>
<th>Description</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>StartDocument</td>
<td>Beginning of the document</td>
<td>Prepare processing (allocate resources)</td>
</tr>
<tr>
<td>StartElement</td>
<td>Beginning of a node</td>
<td>Get current element’s tag name or attributes with <code>getTagName()</code> <code>getAttributes()</code></td>
</tr>
<tr>
<td>Characters</td>
<td>Value of the current element</td>
<td>Get current text element’s value with <code>getCharacters()</code></td>
</tr>
<tr>
<td>SkippedEntity</td>
<td>Reached skipped entity</td>
<td>Get current skipped entity element’s value with <code>skippedEntity()</code></td>
</tr>
<tr>
<td>EndElement</td>
<td>Ending of a node</td>
<td>Get current element’s tagname with <code>getTagName()</code></td>
</tr>
<tr>
<td>EndDocument</td>
<td>Ending of the document</td>
<td>Finish processing (release resources)</td>
</tr>
</tbody>
</table>

**Example**

```plain
DEFINE r om.XmlReader,
    e STRING
...
LET e = r.read()
WHILE e IS NOT NULL
    CASE e
        ...
    END CASE
LET e = r.read()
END WHILE
```

For a complete example, see [Example 1: Parsing an XML file](#) on page 2630.

`om.XmlReader.skippedEntity`

Returns the name of an unresolved entity.

**Syntax**

```plain
skippedEntity()
    RETURNS STRING
```

**Usage**

The `skippedEntity()` method returns the name of the unresolved entity, in the SkippedEntity event context.

The parser identifies well know character entities such as `&amp;` `/&apos;` `/&lt;` `/&gt;` `/&quot;`, other character entities are treated as skipped entities and can be processed in the SkippedEntity event.

**Example**

```plain
DEFINE r om.XmlReader,
    e STRING
...
LET e = r.read()
WHILE e IS NOT NULL
```
CASE e
...
WHEN "SkippedEntity"
    DISPLAY "Entity:'", r.skippedEntity(), "'"
...

For a complete example, see Example 1: Parsing an XML file on page 2630.

**Related concepts**

The `SaxAttributes` class on page 2610
The `SaxAttributes` class holds a set of attributes to process with a SAX reader or writer.

FGLLDPATH on page 278
Defines the search paths to load program modules.

**Examples**

`om.XmlReader` usage examples.

*Example 1: Parsing an XML file*

```plaintext
MAIN
    DEFINE i, l INTEGER
    DEFINE r om.XmlReader
    DEFINE e String
    DEFINE a om.SaxAttributes
    LET r = om.XmlReader.createFileReader("myfile.xml")
    LET l = 0
    LET e = r.read()
    WHILE e IS NOT NULL
        CASE e
            WHEN "StartDocument"
                DISPLAY "StartDocument:"
            WHEN "StartElement"
                LET l = l+1
                DISPLAY l SPACES, "StartElement:", r.getTagName()
                LET a = r.getAttributes()
                FOR i=1 to a.getLength()
                    DISPLAY l SPACES, "  ", a.getName(i), " = ", a.getValueByIndex(i)
                END FOR
            WHEN "Characters"
                DISPLAY l SPACES, "  Characters:'", r.getCharacters(), "'"
            WHEN "SkippedEntity"
                DISPLAY "Entity:'", r.skippedEntity(), "'"
            WHEN "EndElement"
                DISPLAY l SPACES, "EndElement:", r.getTagName()
                LET l = l-1
            WHEN "EndDocument"
                DISPLAY "EndDocument:"
            OTHERWISE
                DISPLAY "Invalid event: ", e
        END CASE
        LET e = r.read()
    END WHILE
END MAIN
```

The `XmlWriter` class

The `XmlWriter` class implements methods to write XML to a stream.

The `XmlWriter` class implements methods to create a `SaxDocumentHandler` object.

Steps to use a XML writer:
1. Declare a variable with the `om.SaxDocumentHandler` type.
2. Create the writer object with one of the class methods of `om.XmlWriter` and assign the reference to the variable.
   - `om.XmlWriter.createFileWriter(filename)` creates an object writing to a file.
   - `om.XmlWriter.createPipeWriter(command)` creates an object writing to a pipe opened by a subprocess.
   - `om.XmlWriter.createSocketWriter(hostname, portnum)` creates an object writing to the TCP socket.
3. Output XML data with the methods of the `om.SaxDocumentHandler` object:
   a. Use the method `startDocument()` to start writing to the output.
   b. From this point, the order of method calls defines the structure of the XML document. To write an element, fill an `om.SaxAttributes` object with attributes.
   c. Then, initiate the element output with the method `startElement()`.
   d. Write element data with the `characters()` method.
   e. Entity nodes are created with the `skippedEntity()` method.
   f. Finish element output with a call to the `endElement()` method.
   g. Repeat these steps as many times as you have elements to write.
   h. Instead of using the `startElement()` method, you can generate processing instruction elements with `processingInstruction()`.
   i. Finally, you must finish the document output with a `endDocument()` call.

### om.XmlWriter methods
Methods of the `om.XmlWriter` class.

#### Table 557: Class methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>om.XmlWriter.createChannelWriter(channel base.Channel)</code></td>
<td>Creates an <code>om.SaxDocumentHandler</code> object writing to a channel object.</td>
</tr>
<tr>
<td><code>om.XmlWriter.createPipeWriter(command STRING)</code></td>
<td>Creates an <code>om.SaxDocumentHandler</code> object writing to a pipe created for a process.</td>
</tr>
<tr>
<td><code>om.XmlWriter.createSocketWriter(host STRING, port INTEGER)</code></td>
<td>Creates an <code>om.SaxDocumentHandler</code> object writing to a socket.</td>
</tr>
</tbody>
</table>

**om.XmlWriter.createChannelWriter**
Creates an `om.SaxDocumentHandler` object writing to a channel object.

**Syntax**

```c
om.XmlWriter.createChannelWriter(
```

**channel** base.Channel )  
RETURNS om.SaxDocumentHandler

1. *channel* is a base.Channel object reference.

**Usage**

The om.XmlWriter.createChannelWriter() class method creates an om.SaxDocumentHandler object that will write to the specified channel object, when using the om.SaxDocumentHandler methods.

The base.Channel object must exist and be open to receive data from the SAX document handler.

**Example**

This example uses the channel to write to stdout, by passing NULL as file name to the base.Channel.openFile() method:

```
DEFINE w om.SaxDocumentHandler  
DEFINE ch base.Channel  
...  
LET ch = base.Channel.create()  
CALL ch.openFile(NULL, "w")  
LET w = om.XmlWriter.createChannelWriter(ch)  
...  
```

**Related concepts**

The **Channel class** on page 2412  
The base.Channel class is a built-in class providing basic input/output functions.

The **SaxDocumentHandler class** on page 2617  
The om.SaxDocumentHandler class provides an interface to write an XML filter with events.

---

**om.XmlWriter.createFileWriter**  
Creates an om.SaxDocumentHandler object writing to a file.

**Syntax**

```
om.XmlWriter.createFileWriter(  
   path STRING  
)  
RETURNS om.SaxDocumentHandler
```

1. *path* is the path to the file.

**Usage**

The om.XmlWriter.createFileWriter() class method creates an om.SaxDocumentHandler object that will write to the specified file when using the om.SaxDocumentHandler methods.

The file is created if it does not exist. If the file cannot be created, the method returns NULL.

When passing NULL as file name, the XmlWriter can be used to write to stdout.

**Example**

```
DEFINE w om.SaxDocumentHandler  
...  
LET w = om.XmlWriter.createFileWriter("mydata.xml")  
IF w IS NULL THEN  
    ERROR "Could not create file."  
    EXIT PROGRAM 1  
END IF  
...  
```
-- Create an XmlWriter object to write to stdout:
LET w = om.XmlWriter.createFileWriter(NULL)
...

Related concepts
The SaxDocumentHandler class on page 2617
The om.SaxDocumentHandler class provides an interface to write an XML filter with events.

`om.XmlWriter.createPipeWriter`
Creates an om.SaxDocumentHandler object writing to a pipe created for a process.

Syntax

```
om.XmlWriter.createPipeWriter(
    command STRING )
RETURNS om.SaxDocumentHandler
```

1. `command` is the command to be executed.

Usage

The `om.XmlWriter.createPipeWriter()` class method creates an `om.SaxDocumentHandler` object that will write to a pipe created for the specified command. XML data will be send through the pipe when using the `om.SaxDocumentHandler` methods.

If the process or pipe cannot be created, the method returns NULL.

Example

```
DEFINE w om.SaxDocumentHandler
...
LET w = om.XmlWriter.createPipeWriter("sort -u")
IF w IS NULL THEN
    ERROR "Could not create process."
    EXIT PROGRAM 1
END IF
...
```

Related concepts
The SaxDocumentHandler class on page 2617
The om.SaxDocumentHandler class provides an interface to write an XML filter with events.

`om.XmlWriter.createSocketWriter`
Creates an om.SaxDocumentHandler object writing to a socket.

Syntax

```
om.XmlWriter.createSocketWriter(
    host STRING,
    port INTEGER )
RETURNS om.SaxDocumentHandler
```

1. `host` is the name of the host computer listening to the TCP port.
2. `port` is the port number to connect to.
Usage

The `om.XmlWriter.createSocketWriter()` class method creates an `om.SaxDocumentHandler` object that will write to a socket identified by the host and port number passed as parameters. XML data will be sent through the socket when using the `om.SaxDocumentHandler` methods.

If the socket cannot be opened, the method returns `NULL`. No timeout is used.

Example

```om
DEFINE w om.SaxDocumentHandler
...
LET w = om.XmlWriter.createSocketWriter("myhost",8012)
IF w IS NULL THEN
    ERROR "Could not open socket."
    EXIT PROGRAM 1
END IF
...
```

Related concepts

The `SaxDocumentHandler` class on page 2617

The `om.SaxDocumentHandler` class provides an interface to write an XML filter with events.

Examples

`om.XmlWriter` usage examples.

Example 1: Writing XML to a file

```om
MAIN
    DEFINE w om.SaxDocumentHandler
    DEFINE a,n om.SaxAttributes

    LET w = om.XmlWriter.createFileWriter("sample.html")
    LET a = om.SaxAttributes.create()
    LET n = om.SaxAttributes.create()

    CALL n.clear()
    CALL w.startDocument()
    CALL w.startElement("HTML",n)
    CALL w.startElement("HEAD",n)
    CALL w.startElement("TITLE",n)
    CALL w.characters("HTML page generated with XmlWriter")
    CALL w.endElement("TITLE")
    CALL a.clear()
    CALL a.addAttribute("type","text/css")
    CALL w.startElement("STYLE",a)
    CALL w.characters("\nBODY { background-color:#c0c0c0; }\n")
    CALL w.endElement("STYLE")
    CALL w.endElement("HEAD")
    CALL w.startElement("BODY",n)
      CALL addHLine(w)
      CALL addTitle(w,"What is XML?",1,"55ff55")
      CALL addParagraph(w,"XML = eXtensible Markup Language ...")
      CALL addHLine(w)
      CALL addTitle(w,"What is SAX?",1,"55ff55")
    CALL w.endElement("BODY")
```
CALL addParagraph(w,"SAX = Simple Api for XML ...")

CALL w.endElement("BODY")

CALL w.endElement("HTML")

CALL w.endDocument()

END MAIN

FUNCTION addHLine(w)
    DEFINE w om.SaxDocumentHandler
    DEFINE a om.SaxAttributes
    LET a = om.SaxAttributes.create()
    CALL a.clear()
    CALL a.addAttribute("width","100%")
    CALL w.startElement("HR",a)
    CALL w.endElement("HR")
END FUNCTION

FUNCTION addTitle(w,t,x,c)
    DEFINE w om.SaxDocumentHandler
    DEFINE t VARCHAR(100)
    DEFINE x INTEGER
    DEFINE a om.SaxAttributes
    DEFINE n varchar(10)
    LET a = om.SaxAttributes.create()
    LET n = "h" || x
    CALL a.clear()
    CALL w.startElement(n,a)
    IF c IS NOT NULL THEN CALL a.addAttribute("color",c)
    END IF
    CALL w.startElement("FONT",a)
    CALL w.characters(t)
    CALL w.endElement("FONT")
    CALL w.endElement(n)
END FUNCTION

FUNCTION addParagraph(w,t)
    DEFINE w om.SaxDocumentHandler
    DEFINE t VARCHAR(2000)
    DEFINE a om.SaxAttributes
    LET a = om.SaxAttributes.create()
    CALL a.clear()
    CALL w.startElement("P",a)
    CALL w.characters("Text is:")
    CALL w.skippedEntity("nbsp") # Add a non breaking space: &nbsp;
    CALL w.characters("is")
    CALL w.characters(t)
    CALL w.endElement("P")
END FUNCTION

Built-in front calls

This section contains the description of all built-in front calls.

Related concepts

User-defined front calls on page 2249
Front-ends can be extended with custom functions to access specific features.

**Built-in front calls summary**

Various front-end functions are implemented within Genero front-ends.

This section describes the front-end functions available for all types of front-ends. Note that several front-end functions are specific to the type of front-end.
### Table 558: Standard front-end functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ui.Interface.frontCall(&quot;standard&quot;, &quot;cbAdd&quot;, [text], [result])</code></td>
<td>Adds to the content of the clipboard.</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;standard&quot;, &quot;cbClear&quot;, [], [result])</code></td>
<td>Clears the content of the clipboard.</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;standard&quot;, &quot;cbGet&quot;, [], [text])</code></td>
<td>Gets the content of the clipboard.</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;standard&quot;, &quot;cbPaste&quot;, [], [result])</code></td>
<td>Pastes the content of the clipboard to the current field.</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;standard&quot;, &quot;cbSet&quot;, [text], [result])</code></td>
<td>Set the content of the clipboard.</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;standard&quot;, &quot;clearFileCache&quot;, [], [result])</code></td>
<td>Clears the local file cache.</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;standard&quot;, &quot;execute&quot;, [cmd,wait], [result])</code></td>
<td>Executes a command on the front-end platform, with or without waiting.</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;standard&quot;, &quot;feInfo&quot;, [name], [result])</code></td>
<td>Queries general front-end properties.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;standard&quot;, &quot;getEnv&quot;, [name], [value])</code></td>
<td>Returns an environment variable set in the user session on the front-end platform.</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;standard&quot;, &quot;getWindowId&quot;, [aui-win-id], [loc-win-id])</code></td>
<td>Returns the local window manager identifier of the window corresponding to the AUI window id passed as parameter.</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;standard&quot;, &quot;hardCopy&quot;, [pgsize], [result])</code></td>
<td>Prints a screenshot of the current window.</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;standard&quot;, &quot;launchURL&quot;, [url [, mode ] ], [])</code></td>
<td>Opens an URL with the default URL handler of the front-end.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;standard&quot;, &quot;mdClose&quot;, [name], [result])</code></td>
<td>Unloads a DLL or shared library front call module.</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;standard&quot;, &quot;openDir&quot;, [path], [caption])</code></td>
<td>Displays a file dialog window to get a directory path on the local file system.</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;standard&quot;, &quot;openFile&quot;, [path], [name], [wildcards], [caption])</code></td>
<td>Displays a file dialog window to let the user select a single file path on the local file system.</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;standard&quot;, &quot;openFiles&quot;, [path], [name], [wildcards], [caption])</code></td>
<td>Displays a file dialog window to let the user select a list of file paths on the local file system.</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;standard&quot;, &quot;playSound&quot;, [resource])</code></td>
<td>Plays the sound file passed as parameter on the front-end platform.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;standard&quot;, &quot;saveFile&quot;, [path], [name], [filetype], [caption])</code></td>
<td>Displays a file dialog window to get a path to save a file on the local file system.</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;standard&quot;, &quot;setReportFont&quot;, [font])</code></td>
<td>Override the font used for GDC report generation for the current application (DBPRINT=FGLSERVER).</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;standard&quot;, &quot;setReportPrinter&quot;, [printer])</code></td>
<td>Override the GDC printer configuration used for report generation for the current application (DBPRINT=FGLSERVER).</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
## Webcomponent module front-end functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Important: This feature is deprecated, its use is discouraged although not prohibited.</td>
<td>Defines the base path where web components are located.</td>
<td>Yes</td>
<td>N/A</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;standard&quot;, &quot;setWebComponentPath&quot;, [path], [])</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;standard&quot;, &quot;shellExec&quot;, [document, action], [result])</td>
<td>Opens a file on the front-end platform with the program associated to the file extension.</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

### Table 559: Webcomponent module front-end functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ui.Interface.frontCall(&quot;webcomponent&quot;, &quot;call&quot;, [aui-name, functionName [...], param1, param2, ...], [result])</td>
<td>Calls a JavaScript function through the web component.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;webcomponent&quot;, &quot;frontCallAPIVersion&quot;, [], [result])</td>
<td>Returns the API version of web component front-end calls.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;webcomponent&quot;, &quot;getTitle&quot;, [aui-name], [result])</td>
<td>Returns the title of the HTML doc rendered by a web component.</td>
</tr>
</tbody>
</table>

### Table 560: Monitor module front-end functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ui.Interface.frontCall(&quot;monitor&quot;, &quot;update&quot;, [path-to-update-file, warning-text [...], elevation-prompt], [result])</td>
<td>Starts the GDC update.</td>
</tr>
</tbody>
</table>
### Table 561: Windows DDE front-end functions

<table>
<thead>
<tr>
<th>Function name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Important:</strong> This feature is deprecated, its use is discouraged although not prohibited.</td>
<td></td>
</tr>
<tr>
<td>CALL [ program, document, encoding ], [ result ]</td>
<td>DDEConnect opens a DDE connection.</td>
</tr>
<tr>
<td><strong>Important:</strong> This feature is deprecated, its use is discouraged although not prohibited.</td>
<td></td>
</tr>
<tr>
<td>CALL [ program, document, command, encoding ], [ result ]</td>
<td>DDEExecute executes a DDE command.</td>
</tr>
<tr>
<td><strong>Important:</strong> This feature is deprecated, its use is discouraged although not prohibited.</td>
<td></td>
</tr>
<tr>
<td>CALL [ program, document ], [ result ]</td>
<td>DDEFinish closes a DDE connection.</td>
</tr>
<tr>
<td><strong>Important:</strong> This feature is deprecated, its use is discouraged although not prohibited.</td>
<td></td>
</tr>
<tr>
<td>CALL [ result ]</td>
<td>DDEFinishAll closes all DDE connections.</td>
</tr>
<tr>
<td><strong>Important:</strong> This feature is deprecated, its use is discouraged although not prohibited.</td>
<td></td>
</tr>
<tr>
<td>CALL [ errmsg ]</td>
<td>DDEError returns error information about the last DDE operation.</td>
</tr>
<tr>
<td><strong>Important:</strong> This feature is deprecated, its use is discouraged although not prohibited.</td>
<td></td>
</tr>
<tr>
<td>CALL [ program, container, cells, encoding ], [ result, value ]</td>
<td>DDEPeek retrieves data from the specified program and document using the DDE channel.</td>
</tr>
<tr>
<td>Function name</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>DDEPoke</strong></td>
<td>Sends data to the specified program and document using the DDE channel.</td>
</tr>
</tbody>
</table>

**Important:** This feature is deprecated, its use is discouraged although not prohibited.

```
CALL
ui.Interface.frontCall("WINDE", "DDEPoke",
[ program, container, cells, values, encoding ], [result])
```

## Table 562: Windows® COM front-end functions

<table>
<thead>
<tr>
<th>Function name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CreateInstance</strong></td>
<td>The CreateInstance function creates an instance of a registered COM object.</td>
</tr>
<tr>
<td><strong>CallMethod</strong></td>
<td>The CallMethod function calls a method on a specified object.</td>
</tr>
<tr>
<td><strong>GetProperty</strong></td>
<td>The GetProperty function gets a property of an object.</td>
</tr>
<tr>
<td><strong>SetProperty</strong></td>
<td>The SetProperty function sets a property of an object.</td>
</tr>
<tr>
<td><strong>GetError</strong></td>
<td>The GetError function gets a description of the last error which occurred.</td>
</tr>
</tbody>
</table>

**Important:** This feature is deprecated, its use is discouraged although not prohibited.

```
CALL
ui.Interface.frontCall("WINCOM", "CreateInstance",
[program], [handle])
```

```
CALL
ui.Interface.frontCall("WINCOM", "CallMethod",
[handle, method, arg1, ... ], [result])
```

```
CALL
ui.Interface.frontCall("WINCOM", "GetProperty",
[handle, member], [result])
```

```
CALL
ui.Interface.frontCall("WINCOM", "SetProperty",
[handle, member, value ], [result])
```

```
CALL
ui.Interface.frontCall("WINCOM", "GetError",
[], [result])
```
<table>
<thead>
<tr>
<th>Function name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Important</strong>: This feature is deprecated, its use is discouraged although not prohibited.</td>
<td></td>
</tr>
</tbody>
</table>

**CALL**

ui.Interface.frontCall("WINCOM","ReleaseInstance",
[handle], [result])

The ReleaseInstance function releases an instance of a COM object.

**CALL**

ui.Interface.frontCall("WinMail","Init",
[], [id])

The Init function initializes the module.

- _ret_ is the identifier of the message initialized.
- For each Init function, a **Close** must be called.

**Important**: This feature is deprecated, its use is discouraged although not prohibited.

**CALL**

ui.Interface.frontCall("WinMail","Close",
[id], [result])

The Close function clears all information corresponding to a message, and frees the memory occupied by the message.

**Important**: This feature is deprecated, its use is discouraged although not prohibited.

**CALL**

ui.Interface.frontCall("WinMail","SetBody",
[id, body], [result])

The SetBody function sets the body of the mail.

**Important**: This feature is deprecated, its use is discouraged although not prohibited.

**CALL**

ui.Interface.frontCall("WinMail","SetSubject",
[id, subject], [result])

The SetSubject function sets the subject of the mail.

**Important**: This feature is deprecated, its use is discouraged although not prohibited.

**CALL**

ui.Interface.frontCall("WinMail","AddTo",
[id, name, address], [result])

The AddTo function adds a "To" addressee to the mail.

---

**Table 563: WinMail front-end functions: General**

<table>
<thead>
<tr>
<th>Function name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Important</strong>: This feature is deprecated, its use is discouraged although not prohibited.</td>
<td></td>
</tr>
</tbody>
</table>

**CALL**

ui.Interface.frontCall("WinMail","Init",
[], [id])

The Init function initializes the module.

- _ret_ is the identifier of the message initialized.
- For each Init function, a **Close** must be called.

**Important**: This feature is deprecated, its use is discouraged although not prohibited.

**CALL**

ui.Interface.frontCall("WinMail","Close",
[id], [result])

The Close function clears all information corresponding to a message, and frees the memory occupied by the message.

**Important**: This feature is deprecated, its use is discouraged although not prohibited.

**CALL**

ui.Interface.frontCall("WinMail","SetBody",
[id, body], [result])

The SetBody function sets the body of the mail.

**Important**: This feature is deprecated, its use is discouraged although not prohibited.

**CALL**

ui.Interface.frontCall("WinMail","SetSubject",
[id, subject], [result])

The SetSubject function sets the subject of the mail.

**Important**: This feature is deprecated, its use is discouraged although not prohibited.

**CALL**

ui.Interface.frontCall("WinMail","AddTo",
[id, name, address], [result])

The AddTo function adds a "To" addressee to the mail.
<table>
<thead>
<tr>
<th>Function name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Important:</strong> This feature is deprecated, its use is discouraged although not prohibited.</td>
<td></td>
</tr>
<tr>
<td>CALL</td>
<td>The AddCC function adds a &quot;CC&quot; addressee to the mail.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;WinMail&quot;,&quot;AddCC&quot;, [ id, name, address ], [ result ] )</td>
<td></td>
</tr>
<tr>
<td><strong>Important:</strong> This feature is deprecated, its use is discouraged although not prohibited.</td>
<td></td>
</tr>
<tr>
<td>CALL</td>
<td>The AddBCC function adds a &quot;BCC&quot; addressee to the mail.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;WinMail&quot;,&quot;AddBCC&quot;, [ id, name, address ], [ result ] )</td>
<td></td>
</tr>
<tr>
<td><strong>Important:</strong> This feature is deprecated, its use is discouraged although not prohibited.</td>
<td></td>
</tr>
<tr>
<td>CALL</td>
<td>The AddAttachment function adds a file as an attachment to the mail.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;WinMail&quot;,&quot;AddAttachment&quot;, [ id, fileName], [ result ] )</td>
<td></td>
</tr>
<tr>
<td><strong>Important:</strong> This feature is deprecated, its use is discouraged although not prohibited.</td>
<td></td>
</tr>
<tr>
<td>CALL</td>
<td>The SendMailSMTP function sends the mail with the SMTP protocol.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;WinMail&quot;,&quot;SendMailSMTP&quot;, [ id ], [ result ] )</td>
<td></td>
</tr>
<tr>
<td><strong>Important:</strong> This feature is deprecated, its use is discouraged although not prohibited.</td>
<td></td>
</tr>
<tr>
<td>CALL</td>
<td>The SendMailMAPI function sends the mail with the MAPI protocol.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;WinMail&quot;,&quot;SendMailMAPI&quot;, [ id ], [ result ] )</td>
<td></td>
</tr>
<tr>
<td><strong>Important:</strong> This feature is deprecated, its use is discouraged although not prohibited.</td>
<td></td>
</tr>
<tr>
<td>CALL</td>
<td>The GetError function gets a description of the last error that occurred.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;WinMail&quot;,&quot;GetError&quot;, [ id ], [ result ] )</td>
<td></td>
</tr>
</tbody>
</table>
Table 564: WinMail front-end functions: SMTP-specific

<table>
<thead>
<tr>
<th>Function name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Important:</strong> This feature is deprecated, its use is discouraged although not prohibited.</td>
<td>The SetSmtp function sets the SMTP server to be used.</td>
</tr>
<tr>
<td>CALL</td>
<td>ui.Interface.frontCall(&quot;WinMail&quot;,&quot;SetSmtp&quot;, [ id, smtp:port ], [ result ] )</td>
</tr>
</tbody>
</table>

**Important:** This feature is deprecated, its use is discouraged although not prohibited.

CALL
ui.Interface.frontCall("WinMail","SetFrom", [ id, name, address ], [ result ] )

The SetFrom function sets sender information.

Table 565: theme module front-end functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ui.Interface.frontCall(&quot;theme&quot;, &quot;setTheme&quot;, [name], [])</td>
<td>Activates a specific GBC theme.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;theme&quot;, &quot;getCurrentTheme&quot;, [], [result])</td>
<td>Gets the active GBC theme.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;theme&quot;, &quot;listThemes&quot;, [], [result])</td>
<td>Lists all available GBC themes.</td>
</tr>
</tbody>
</table>
Table 566: localStorage module front-end functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ui.Interface.frontCall(&quot;localStorage&quot;, &quot;clear&quot;, [], [])</code></td>
<td>Removes all local storage key/value pairs.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;localStorage&quot;, &quot;getItem&quot;, [key], [value])</code></td>
<td>Returns the current value of local storage key.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;localStorage&quot;, &quot;keys&quot;, [], [key-list])</code></td>
<td>Returns the list of defined local storage keys.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;localStorage&quot;, &quot;removeItem&quot;, [key], [])</code></td>
<td>Deletes the specified local storage key.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;localStorage&quot;, &quot;setItem&quot;, [key,value], [])</code></td>
<td>Sets a value for local storage key.</td>
</tr>
<tr>
<td>Function Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;mobile&quot;, &quot;chooseContact&quot;, [], [result])</code></td>
<td>Lets the user choose a contact from the mobile device contact list and returns the vCard.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;mobile&quot;, &quot;choosePhoto&quot;, [], [path])</code></td>
<td>Lets the user select a picture from the mobile device's photo gallery and returns a picture identifier.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;mobile&quot;, &quot;chooseVideo&quot;, [], [path])</code></td>
<td>Lets the user select a video from the mobile device's video gallery and returns a video identifier.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;mobile&quot;, &quot;composeMail&quot;, [to, subject, content, cc, bcc, attachments ...], [result])</code></td>
<td>Invokes the user's default mail application for a new mail to send.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;mobile&quot;, &quot;composeSMS&quot;, [recipients, content], [result])</code></td>
<td>Sends an SMS text to one or more phone numbers.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;mobile&quot;, &quot;connectivity&quot;, [], [result])</code></td>
<td>Returns the type of network available for the mobile device.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;mobile&quot;, &quot;getGeolocation&quot;, [], [status, latitude, longitude])</code></td>
<td>Returns the Global Positioning System (GPS) location of a mobile device.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;mobile&quot;, &quot;getRemoteNotifications&quot;, [], [data])</code></td>
<td>This front call retrieves push notification messages.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;mobile&quot;, &quot;importContact&quot;, [vcard], [result])</code></td>
<td>Creates a new contact, or merges to an existing entry, the contact details passed in a vCard string.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;mobile&quot;, &quot;isForeground&quot;, [], [result])</code></td>
<td>Indicates if the mobile app is in foreground mode.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;mobile&quot;, &quot;newContact&quot;, [defaults], [vcard])</code></td>
<td>Lets the user input contact information to create a new entry in the contact database of the mobile device.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;mobile&quot;, &quot;registerForRemoteNotifications&quot;, [], [registration-token])</code></td>
<td>This front call registers a mobile device for push notifications.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;mobile&quot;, &quot;runOnServer&quot;, [appurl, timeout], [])</code></td>
<td>Run an application from the Genero Application Server using the specified URL.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;mobile&quot;, &quot;scanBarCode&quot;, [], [code, type])</code></td>
<td>Allows the user to scan a barcode with a mobile device.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;mobile&quot;, &quot;takePhoto&quot;, [], [path])</code></td>
<td>Lets the user take a picture with the mobile device and returns the corresponding picture identifier.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;mobile&quot;, &quot;takeVideo&quot;, [], [path])</code></td>
<td>Lets the user take a video with the mobile device and returns the corresponding video identifier.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;mobile&quot;, &quot;unregisterFromRemoteNotifications&quot;, [], [])</code></td>
<td>This front call unregisters the mobile device from push notifications.</td>
</tr>
</tbody>
</table>
### Table 568: Android™ module front-end functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ui.Interface.frontCall(&quot;android&quot;, &quot;askForPermission&quot;, [permission], [result])</code></td>
<td>Ask the user to enable a dangerous feature on the Android device.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;android&quot;, &quot;showAbout&quot;, [])</code></td>
<td>Shows the GMA about box displaying version information.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;android&quot;, &quot;showSettings&quot;, [])</code></td>
<td>Shows the GMA settings box controlling debug options.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;android&quot;, &quot;startActivity&quot;, [action, data, category, type, component, extras], [])</code></td>
<td>Starts an external Android™ application (activity), and returns to the GMA application immediately.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;android&quot;, &quot;startActivityForResult&quot;, [action, data, category, type, component, extras], [outdata, outextras])</code></td>
<td>Starts an external application (Android™ activity) and waits until the activity is closed.</td>
</tr>
</tbody>
</table>

### Table 569: iOS module front-end functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ui.Interface.frontCall(&quot;ios&quot;, &quot;getBadgeNumber&quot;, [])</code></td>
<td>Returns the current badge number associated to the app.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;ios&quot;, &quot;setBadgeNumber&quot;, [value], [])</code></td>
<td>Sets the current badge number associated to the app.</td>
</tr>
</tbody>
</table>
## Table 570: Cordova front-end functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ui.Interface.frontCall(&quot;cordova&quot;, &quot;call&quot;, [plugin-name, function-name [, param1, param2, ... ]], [result])</code></td>
<td>Calls a function in a Cordova plugin and returns a result.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;cordova&quot;, &quot;callWithoutWaiting&quot;, [plugin-name, function-name [, param1, param2, ... ]], [callback-id])</code></td>
<td>Calls a function asynchronously in a Cordova plugin, without waiting for a result.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;cordova&quot;, &quot;getAllCallbackData&quot;, [callback-id-filter], [results])</code></td>
<td>Returns all results for asynchronous Cordova plugin front calls, based on a callback ID filter.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;cordova&quot;, &quot;getCallbackDataCount&quot;, []), [count]</code></td>
<td>Returns the number of pending Cordova plugin results.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;cordova&quot;, &quot;getFirstCallbackData&quot;, [], [result, callback-id])</code></td>
<td>Returns the first Cordova plugin result from the result queue of all asynchronous Cordova plugin front calls, and removes it from the queue.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;cordova&quot;, &quot;getPluginInfo&quot;, [plugin-name], [result])</code></td>
<td>Returns details about a specific Cordova plugin.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;cordova&quot;, &quot;listPlugins&quot;, []), [plugins]</code></td>
<td>Returns the list of available Cordova plugins.</td>
</tr>
</tbody>
</table>

### Standard front calls

Standard front call functions provide common utility APIs to control the front-end.

This table shows the functions implemented by the front-ends in the "standard" module, available on all front-ends.
<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>ui.Interface.frontCall(&quot;standard&quot;, &quot;cbAdd&quot;, [text], [result])</td>
<td>Adds to the content of the clipboard.</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;standard&quot;, &quot;cbClear&quot;, [], [result])</td>
<td>Clears the content of the clipboard.</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;standard&quot;, &quot;cbGet&quot;, [], [text])</td>
<td>Gets the content of the clipboard.</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;standard&quot;, &quot;cbPaste&quot;, [], [result])</td>
<td>Pastes the content of the clipboard to the current field.</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;standard&quot;, &quot;cbSet&quot;, [text], [result])</td>
<td>Set the content of the clipboard.</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;standard&quot;, &quot;clearFileCache&quot;, [], [result])</td>
<td>Clears the local file cache.</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;standard&quot;, &quot;execute&quot;, [cmd, wait], [result])</td>
<td>Executes a command on the front-end platform, with or without waiting.</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;standard&quot;, &quot;feInfo&quot;, [name], [result])</td>
<td>Queries general front-end properties.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;standard&quot;, &quot;getEnv&quot;, [name], [value])</td>
<td>Returns an environment variable set in the user session on the front-end platform.</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;standard&quot;, &quot;getWindowId&quot;, [auui-win-id], [loc-win-id])</td>
<td>Returns the local window manager identifier of the window corresponding to the AUI window id passed as parameter.</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;standard&quot;, &quot;hardCopy&quot;, [pgsize], [result])</td>
<td>Prints a screenshot of the current window</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;standard&quot;, &quot;launchURL&quot;, [url [, mode ]], [])</td>
<td>Opens an URL with the default URL handler of the front-end.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;standard&quot;, &quot;mdClose&quot;, [name], [result])</td>
<td>Unloads a DLL or shared library front call module.</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;standard&quot;, &quot;openDir&quot;, [path], [caption])</td>
<td>Displays a file dialog window to get a directory path on the local file system.</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;standard&quot;, &quot;openFile&quot;, [path], [name], [wildcards], [caption])</td>
<td>Displays a file dialog window to let the user select a single file path on the local file system.</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;standard&quot;, &quot;openFiles&quot;, [path], [name], [wildcards], [caption])</td>
<td>Displays a file dialog window to let the user select a list of file paths on the local file system.</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;standard&quot;, &quot;playSound&quot;, [resource], [])</td>
<td>Plays the sound file passed as parameter on the front-end platform.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;standard&quot;, &quot;saveFile&quot;, [path], [name], [filetype], [caption])</td>
<td>Displays a file dialog window to get a path to save a file on the local file system.</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;standard&quot;, &quot;setReportFont&quot;, [font], [result])</td>
<td>Override the font used for GDC report generation for the current application (DBPRINT=FGLSERVER).</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;standard&quot;, &quot;setReportPrinter&quot;, [printer], [result])</td>
<td>Override the GDC printer configuration used for report generation for the current application (DBPRINT=FGLSERVER).</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Function Name</td>
<td>Description</td>
<td>GDC</td>
<td>GBC</td>
<td>GMA</td>
<td>GMI</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td><strong>Important:</strong> This feature is deprecated, its use is discouraged although not prohibited.</td>
<td>Defines the base path where web components are located.</td>
<td>Yes</td>
<td>N/A</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;standard&quot;, &quot;setWebComponentPath&quot;, [path], [])</td>
<td>Opens a file on the front-end platform with the program associated to the file extension.</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**cbAdd**

Adds to the content of the clipboard.

**Syntax**

```javascript
ui.Interface.frontCall("standard", "cbAdd", [text], [result])
```

1. **text** - The text to be added.
2. **result** - The execution status (TRUE=sucess, FALSE=error).

**Usage**

The "cbAdd" front call adds the text passed as parameter to the content of the clipboard of the front-end platform.

**cbClear**

Clears the content of the clipboard.

**Syntax**

```javascript
ui.Interface.frontCall("standard", "cbClear", [], [result])
```

1. **result** - The execution status (TRUE=sucess, FALSE=error).

**Usage**

The "cbClear" front call clears the content of the clipboard. This front call takes no input parameters.

**cbGet**

Gets the content of the clipboard.

**Syntax**

```javascript
ui.Interface.frontCall("standard", "cbGet", [], [text])
```

1. **text** - The text found in the clipboard.
**Usage**
The "cbGet" front call returns the current content of the clipboard.
This front call takes no input parameters.

**cbPaste**
Pastes the content of the clipboard to the current field.

**Syntax**
```
ui.Interface.frontCall("standard", "cbPaste",
 [], [result])
```

1. **result** - The execution status (TRUE = success, FALSE = error).

**Usage**
The "cbPaste" front call pastes the content of the clipboard to the current field.
This front call takes no input parameters.

**cbSet**
Set the content of the clipboard.

**Syntax**
```
ui.Interface.frontCall("standard", "cbSet",
 [text], [result])
```

1. **text** - The text to be set.
2. **result** - The execution status (TRUE = success, FALSE = error).

**Usage**
The "cbSet" front call sets the content of the clipboard with the text passed as parameter.

**Note:** GBC specific: due to browsers security policies, GBC cannot paste content in the clipboard without user permissions. When using the cbSet call, GBC will present data in a text box, that you can copy and paste elsewhere.

**clearFileCache**
Clears the local file cache.

**Important:** This feature is provided for development and debug purpose, and must not be used in a production environment.

**Syntax**
```
ui.Interface.frontCall("standard", "clearFileCache",
 [], [result])
```

1. **result** - The execution status (TRUE = success, FALSE = error).

**Usage**
The "clearFileCache" front call removes all the local resource files cached by the front-end.

The resource files (like images, font files) transmitted by the runtime system to the front-end are automatically cached on the front-end workstation.

If needed, it is possible to clear the cache with the clearFileCache front call.
Note: This front call is only supported by the GDC front-end.

Related information
The resource file cache of the front-end on page 1152

execute
Executes a command on the front-end platform, with or without waiting.

Syntax

```
ui.Interface.frontCall("standard", "execute", 
[cmd, wait], [result])
```

1. `command` - The command to be executed.
2. `wait` - The wait option (TRUE=wait, FALSE=do not wait).
3. `result` - The execution status (TRUE=succes, FALSE=error).

Usage
The "execute" front call runs a command on the front-end platform, with or without waiting option.

Tip: When specifying a file path, pay attention to platform specific rules regarding directory separators and space characters in file names. When the front-end executes on a recent Microsoft™ Windows™ system, you can use the / slash character as directory separator, like on Unix systems. A directory or file name can contain spaces, and there is no need to surround the path with double quotes in such case. When using backslash directory separators, make sure to escape backslash characters in string literals with \\.

If the second parameter is set to TRUE, the runtime system will wait until the front-end gives the control back after the local command was executed.

Tip: With the front-end running on Microsoft™ Windows™ systems platforms, depending on the program to be executed, you may need to add the "cmd /C" command before the executable name and program arguments.

Example
For Microsoft™ Windows™ system:

```
-- Using backslash as directory separator:
CALL ui.Interface.frontCall("standard", "execute", 
["C:\\Program Files\\FourJS\\gdc\\3.10\\bin\\gdc.exe --help",FALSE],
[res] )

-- Using slash as directory separator:
CALL ui.Interface.frontCall("standard", "execute", 
["C:/Program Files/FourJS/gdc/3.10/bin/gdc.exe --help",FALSE], [res] )
```

Related concepts
startActivity (Android) on page 2716
Starts an external Android™ application (activity), and returns to the GMA application immediately.

startActivityResult (Android) on page 2717
Starts an external application (Android™ activity) and waits until the activity is closed.

feInfo
Queries general front-end properties.

Syntax

```
ui.Interface.frontCall("standard", "feInfo", 
[name], [result])
```
1. **name** - The name of the property.
2. **result** - The value of the property.

**Usage**

The `feInfo` front call returns a front-end property value depending on the property name passed in as the parameter. Some `feInfo` options take an optional parameter, such as `screenResolution`:

```javascript
CALL ui.Interface.frontCall("standard", "feInfo", ["screenResolution", 2], [resolution])
```

**Table 572: Property names and descriptions for the standard.feInfo front call**

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
</table>
| dataDirectory      | Returns the directory name that can be used for temporary files on the front-end side. This directory is cleaned at front-end start-up and end, and is common to all front-end instances, except for the Genero Browser Client (GBC). The possible values returned are:  
  • With Genero Browser Client, this is not applicable.  
  • With Genero Desktop Client, the local cache directory. For example, "/home/username/.cache/Four Js/Genero Desktop".  
  • With Genero Mobile for Android™, this is the GMA application cache directory. Content may be erased, once the app is closed.  
  • With Genero Mobile for iOS, this is the temporary directory in the application sandbox (iOS NSTemporaryDirectory() system call). Content may be erased, once the app is closed. | Yes | No  | Yes | Yes |
<p>| dictionariesDirectory | Returns the directory name where spell checking dictionary files are located. This parameter is only supported by GDC, for the spellCheck style attribute of TextEdit elements. A program can query the dictionariesDirectory info in order to send dictionary files to the GDC with an fgl_putfile() call. | Yes | N/A | N/A | N/A |
| deviceModel        | Returns the name of the device, for example &quot;iPad4,5&quot;.                      | Yes | No  | Yes | Yes |</p>
<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
</table>
| deviceId      | • With Genero Mobile for iOS, returns the identifierForVendor.  
• With Genero Mobile for Android™, returns the IMEI, otherwise the Android™ id (but may change after device re-installation)  
**Important:** For GMA / Android™, accessing this device information requires the READ_PHONE_STATE. Starting with Android 10, the READ_PRIVILEGED_PHONE_STATE privileged permission is required, but this privilege can only be set for fully managed devices. See [Android permissions](#) on page 3595 for more details. | Yes | No | Yes | Yes |
| feName        | The code identifying the type of front-end component.  
**Tip:** To save a network round trip, it is recommended to use `ui.Interface.getFrontEndName()` instead. | Yes | Yes | Yes | Yes |
| fePath        | The installation directory of the front-end executable.  
• With Genero Desktop Client, it returns the path to the installation directory of the GDC.  
• With Genero Browser Client, it returns the path to the installation directory of the GAS.  
• With Genero Mobile for Android™, it returns the installation directory. For example, `/data/data/com.fourjs.gma/fgl`.  
• With Genero Mobile for iOS, it returns the installation directory. For example: `/private/var/mobile/Applications/B3E6-C48A-ED4EFA`. Below the installation directory are the "Documents" (which is by default `pwd`), "GMI.app" (deployed p-code resides in `GMI.app/app/`) and "tmp" directories.  
**Important:** The installation path returned by this front call may change in future versions, do not base application code on this. On mobile devices, consider using the `os.Path.pwd` on page 2806 utility function to get the application working directory when executing programs. | Yes | No | Yes | Yes |
<p>| freeStorageSpace | Returns the number of bytes available on the device. | Yes | No | Yes | Yes |</p>
<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>iccid</td>
<td>With Genero Mobile for iOS, returns an error (not allowed).</td>
<td>N/A</td>
<td>N/A</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>With Genero Mobile for Android™, returns the ICCID if available, otherwise raise an error.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Important:</strong> For GMA / Android™, accessing this device information requires the READ_PHONE_STATE. Starting with Android 10, the READ_PRIVILEGED_PHONE_STATE privileged permission is required, but this privilege can only be set for fully managed devices. See Android permissions on page 3595 for more details.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>imei</td>
<td>With Genero Mobile for iOS, returns an error (not allowed).</td>
<td>N/A</td>
<td>N/A</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>With Genero Mobile for Android™, returns the IMEI if available, otherwise raises an error.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Important:</strong> For GMA / Android™, accessing this device information requires the READ_PHONE_STATE. Starting with Android 10, the READ_PRIVILEGED_PHONE_STATE privileged permission is required, but this privilege can only be set for fully managed devices. See Android permissions on page 3595 for more details.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ip</td>
<td>Returns the IP address of the network interface used for the GUI connection. For mobile platforms, this is the preferred IP address of the device. If there is WIFI, either the IPv4 address is returned (for example: 192.168.0.12) or if there is no IPv4 address, the IPv6 address is returned (for example: 2a02:810a:82c0:478:d462:e334:6a1d:fb78). If there is no WIFI, either the cellular IPv4 or IPv6 address is returned. If there is no network, NULL is returned.</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>numScreens</td>
<td>Number of screens available on the front-end platform. On typical front-end platforms and devices, the number of screens is 1. In some rare cases, a desktop computer can be configured with more than one screen.</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>osType</td>
<td>The operating system type where the front-end is running. Possible return values include &quot;WINDOWS&quot;, &quot;LINUX&quot;, &quot;macOS&quot;, &quot;ANDROID&quot;, &quot;IOS&quot;.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Property name</td>
<td>Description</td>
<td>GDC</td>
<td>GBC</td>
<td>GMA</td>
<td>GMI</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>osVersion</td>
<td>The version of the operating system.</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Example of returned values: &quot;4.3&quot;, &quot;5.10.15&quot;.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ppi</td>
<td>Returns the screen pixel density of the front-end platform (Pixels Per Inch). This front call takes an optional screen number as parameter (1 is the default).</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>• With Genero Mobile for iOS, it returns the PixelsPerInch of an iOS device.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• With Genero Mobile for Android™, it returns the DPI (ppi == dpi)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Returns the screen resolution of the front-end platform.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>This front call takes an optional screen number as parameter (1 is the default).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Example of returned values: &quot;1200x1824&quot;, &quot;1920x1104&quot;.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> For mobile devices, the value can change depending on the device orientation.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Returns the build platform target code name, identifying the operating system the front-end binary was compiled. This front call is provided for debugging purpose, do not base code on the returned value, it can change if the target OS version is upgraded for example. Use the osType property instead.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Example of returned values:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;w32v100&quot; = Windows® 32 bits, Visual C++ 10.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;w64v110&quot; = Windows® 64 bits, Visual C++ 11.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;d32a040&quot; = Android™ 4.0 ARM 32 bits.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;d32x040&quot; = Android™ 4.0 x86 32 bits.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;i32a070&quot; = iOS 7.0 ARM 32 bits.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;i32x070&quot; = iOS 7.0 x86 32 bits.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> For GBC, it returns the same value as osType.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Returns the current size of the front-end view-port.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>• For mobile front-ends, this is the size of the mobile screen.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• For Genero Desktop Client, this is the size of the current window.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• For Genero Browser Client, this is the size of the browser webview.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Example of returned values: &quot;1200x1824&quot;, &quot;1920x1104&quot;.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### userPreferredLang

Returns the language and territory of the locale defined on the front-end platform, in the `language_territory` format.

- For Genero Desktop Client and mobile front-ends, the front-end locale is defined by the operating system.
- For Genero Browser Client, the front-end locale is defined in the web browser preferences.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
<th>GDC</th>
<th>GBC</th>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>userPreferredLang</td>
<td>Returns the language and territory of the locale defined on the front-end platform, in the <code>language_territory</code> format.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### getEnv

Returns an environment variable set in the user session on the front end platform.

**Syntax**

```java
ui.Interface.frontCall("standard", "getEnv", [name], [value])
```

1. `name` - The name of the environment variable.
2. `value` - The value of the environment variable.

**Usage**
The "getEnv" front call returns an environment variable set in the user session on the front-end platform.

**Related concepts**
- [Environment variables](#) on page 262
- Genero BDL related environment variables.

### getWindowId

Returns the local window manager identifier of the window corresponding to the AUI window id passed as parameter.

**Syntax**

```java
ui.Interface.frontCall("standard", "getWindowId", [aui-win-id], [loc-win-id])
```

1. `aui-win-id` - The id of the window node in the AUI tree.
2. `loc-win-id` - The id of the window in the window manager where the front-end is running.

**Usage**
The `getWindowId` front call returns the local identifier that corresponds to the AUI window id passed as parameter, in the window manager where the front-end is displaying the application forms.

The node id must reference a `Window` node, otherwise "0" is returned. In traditional mode, window widgets are simple frames. Use "0" as `aui-win-id` parameter to get the top level window id in the local window system.

### hardCopy

Prints a screenshot of the current window.

**Syntax**

```java
ui.Interface.frontCall("standard", "hardCopy",)
```
1. **pgsize** - Pass "1" to adapt the screenshot to the page size.
2. **result** - The execution status (TRUE=success, FALSE=error).

### Usage

The "hardCopy" front call allows you to print a screenshot of the current window.

The `pgsize` parameter is optional; either leave out, or enter "1" to indicate that the screenshot must be adapted to the page size.

### launchURL

Opens an URL with the default URL handler of the front-end.

### Syntax

```java
ui.Interface.frontCall("standard", "launchURL", [
  url ↓, mode ↓ ], [] )
```

1. **url** - The URL / URI to invoke.
2. **mode** (optional) - This parameter is ignored by GMA/GMI/GDC. With GBC, by default, a new browser window/tab is opened. Use "replace" mode, to reuse the current web browser window/tab for the specified URL.

### Usage

The "launchURL" front call opens an URL with the default URL handler available on the front-end platform. This is typically the web browser for "HTTP:" URLs, or the mailer for "mailto:" URLs, but the corresponding application may also be dedicated to the type of object specified by the URL (for example, a mapping service or to initiate a phone call).

**Important:** Some types of URLs are not supported by all front-end platforms. Make sure that you test all target front-ends when using a launchURL front call. For example, when the GBC front-end is running from HTTP/HTTPS (through the GAS), the web browser will block file://host/path URIs from opening, as this would create a security hole.

This front call is a powerful feature: front-end applications can register themselves as URL handlers, so you can start applications on the front-end through the launchURL front call.

Supported schemes depend on your system configuration.

**Tip:** It is possible to produce an URI with the `ui.Interface.filenameToURI()` method, from the file located on the application server. This URI can then be used with the launchURL front call for example, to show PDF files.

The `mode` parameter is optional and is interpreted differently depending on the front-end type:

- With the Genero Browser Client (GBC), use "replace" for the `mode` parameter, if you want the current application in the browser window or tab to be replaced with the new URL, instead of launching a new browser window or tab. If it is not present, or if a value other than "replace" is specified, the Genero Browser Client behaves like the Genero Desktop Client, opening the URL in a new browser window.
- With Genero Mobile for Android (GMA), Genero Mobile for iOS (GMI), and Genero Desktop Client (GDC) front-ends, the `mode` parameter is ignored if specified.

### Example

To invoke Google Play Store:

```java
CALL ui.Interface.frontCall("standard", "launchURL",
```
To open Google Maps:

```javascript
CALL ui.Interface.frontCall("standard", "launchURL", [
"geo:48.613363,7.711083?z=17"], [])
```

To open Google Street View:

```javascript
CALL ui.Interface.frontCall("standard", "launchURL", [
"google.streetview:cbll=48.613363,7.711083&cbp=1,0,,0,1.0&mz=17"], [])
```

To initiate a phone call:

```javascript
CALL ui.Interface.frontCall("standard", "launchURL", [
"tel:+336717623"], [])
```

**mdClose**

Unloads a DLL or shared library front call module.

**Syntax**

```javascript
ui.Interface.frontCall("standard", "mdClose", [name], [result])
```

1. **name** - The name of the module to be closed.
2. **result** - The result status (0 = success, -1 = module not found, -2 = cannot unload (busy)).

**Usage**

Front call modules are loaded on demand. After calling a function of a specific module, you can use the "mdClose" front call to unload the shared library and save resources.

**openDir**

Displays a file dialog window to get a directory path on the local file system.

**Syntax**

```javascript
ui.Interface.frontCall("standard", "openDir", [path, caption], [result])
```

1. **path** - The default path of the directory like "/tmp".
2. **caption** - The caption to be displayed.
3. **result** - The name of the selected directory (or NULL if canceled).

**Usage**

When invoking the "openDir" front call, the front-end displays the file dialog window using the local file system, to let the end user enter a directory path.

**Note**: The file dialog window rendering and features depend on the type of front end and the type of the front end platform (desktop OS, web browser).

If the user cancels the dialog, the front call returns NULL in the result variable.
Tip: When specifying a file path, pay attention to platform specific rules regarding directory separators and space characters in file names. When the front-end executes on a recent Microsoft™ Windows™ system, you can use the / slash character as directory separator, like on Unix systems. A directory or file name can contain spaces, and there is no need to surround the path with double quotes in such case. When using backslash directory separators, make sure to escape backslash characters in string literals with \\.

**Example**

```main
DEFINE rec RECORD
   path STRING,
   caption STRING
END RECORD
DEFINE result STRING

LET rec.path = "/tmp"
LET rec.caption = "Select directory"
CALL ui.Interface.frontCall("standard","openDir", [rec.*],[result])

IF result IS NULL THEN
   DISPLAY "No directory was selected."
ELSE
   DISPLAY "Directory ": result
END IF

END MAIN
```

**Related concepts**

- openFile on page 2659
  Displays a file dialog window to let the user select a single file path on the local file system.

- openFiles on page 2660
  Displays a file dialog window to let the user select a list of file paths on the local file system.

- saveFile on page 2662
  Displays a file dialog window to get a path to save a file on the local file system.

- openFile
  Displays a file dialog window to let the user select a single file path on the local file system.

**Syntax**

```ui.Interface.frontCall("standard", "openFile",
   [path, name, wildcards, caption],
   [result])```

1. **path** - The initial directory path to look for files. On some platforms, when path is a filename like "/tmp/document.txt", the name of the file is used as default file to be selected.
2. **name** - The label to be displayed for the file types / wildcards.
3. **wildcards** - A space-separated list of wildcards (for example: "*.pdf" or "README* test*.txt")
4. **caption** - The caption of the file dialog window / frame.
5. **result** - The name of the selected file (or NULL if canceled).

**Usage**

When invoking the "openFile" front call, the front-end displays a file dialog window using the local file system, to let the end user select an existing file.

**Note**: The file dialog window rendering and features depend on the type of front end and the type of the front end platform (desktop OS, web browser).
If the user cancels the dialog, the front call returns **NULL** in the result variable.

**Important:** With the GBC front-end in a web browser, the *path* parameter is ignored, and *wildcards* can only hold one type of file extension.

**Tip:** When specifying a file path, pay attention to platform specific rules regarding directory separators and space characters in file names. When the front-end executes on a recent Microsoft® Windows™ system, you can use the / slash character as directory separator, like on Unix systems. A directory or file name can contain spaces, and there is no need to surround the path with double quotes in such case. When using backslash directory separators, make sure to escape backslash characters in string literals with `\`.

**Example**

```plaintext
MAIN
    DEFINE rec RECORD
        path STRING,
        name STRING,
        wildcards STRING,
        caption STRING
    END RECORD
    DEFINE result STRING
    LET rec.path = "/tmp"
    LET rec.name = "Image files"
    LET rec.wildcards = "*.jpg *.png"
    LET rec.caption = "Open file"
    CALL ui.Interface.frontCall("standard","openFile",[rec.*],[result])
    IF result IS NULL THEN
        DISPLAY "No file was selected."
    ELSE
        DISPLAY "File : ", result
    END IF
END MAIN
```

**Related concepts**

- **openFiles** on page 2660
  Displays a file dialog window to let the user select a list of file paths on the local file system.
- **openDir** on page 2658
  Displays a file dialog window to get a directory path on the local file system.
- **saveFile** on page 2662
  Displays a file dialog window to get a path to save a file on the local file system.

**openFiles**
Displays a file dialog window to let the user select a list of file paths on the local file system.

**Syntax**

```plaintext
ui.Interface.frontCall("standard", "openFiles",
    [path,name,wildcards,caption],
    [result])
```

1. *path* - The default path of a directory like "/tmp".
2. *name* - The label to be displayed for the file types / wildcards.
3. *wildcards* - A space-separated list of wildcards (for example: "*.pdf" or "README* test*.txt")
4. *caption* - The caption to be displayed.
5. *result* - The list of selected file paths as a JSON array (or NULL if canceled).
Usage

When invoking the "openFiles" front call, the front-end displays a file dialog window, to let the end user select several file paths from the local file system.

**Note:** The file dialog window rendering and features depend on the type of front end and the type of the front end platform (desktop OS, web browser).

This front call is typically used to let the end user select several files that will be processed by the application.

**Important:** With the GBC front-end in a web browser, the *path* parameter is ignored, and *wildcards* can only hold one type of file extension.

When the file dialog is validated, the result variable contains a JSON formatted string representing an array of file paths:

```
["/my/first/path", "/my/second/path", "/my/third/path"]
```

The resulting string can then be converted to a DYNAMIC ARRAY OF STRING with the `util.JSON.parse()` method.

**Note:** The order of the paths in the result variable can differ from the selection order of the user.

If the user cancels the dialog, the front call returns an empty JSON array ([]) in the result variable.

**Tip:** When specifying a file path, pay attention to platform specific rules regarding directory separators and space characters in file names. When the front-end executes on a recent Microsoft™ Windows™ system, you can use the / slash character as directory separator, like on Unix systems. A directory or file name can contain spaces, and there is no need to surround the path with double quotes in such case. When using backslash directory separators, make sure to escape backslash characters in string literals with \\.

Example

```plaintext
IMPORT util
MAIN
  DEFINE rec RECORD
    path STRING,
    name STRING,
    wildcards STRING,
    caption STRING
  END RECORD
  DEFINE result STRING
  DEFINE files DYNAMIC ARRAY OF STRING
  DEFINE x INTEGER

  LET rec.path = "/tmp"
  LET rec.name = "Image files"
  LET rec.wildcards = "*.jpg *.png"
  LET rec.caption = "Select files"
  CALL ui.Interface.frontCall("standard","openFiles",[rec.*],[result])

  CALL util.JSON.parse( result, files )

  FOR x=1 TO files.getLength()
    DISPLAY SFMT("File %1: %2 ", x, files[x])
  END FOR
END MAIN
```

Related concepts

- [openDir](#) on page 2658
  Displays a file dialog window to get a directory path on the local file system.
- [openFile](#) on page 2659
Displays a file dialog window to let the user select a single file path on the local file system.

**saveFile** on page 2662
Displays a file dialog window to get a path to save a file on the local file system.

**playSound**
Plays the sound file passed as parameter on the front-end platform.

**Syntax**

```javascript
ui.Interface.frontCall("standard", "playSound", [resource], [])
```

1. *resource* - The sound file to play.

**Usage**

The "playSound" front call opens the sound file/resource passed as parameter and plays the sound on the front-end. Supported sound file format depends on the front-end infrastructure (platform, technology, web browser, etc.) The sound file must be located on the front-end platform, or be an URI that can be loaded by the front-end.

**Tip:** It is possible to produce an URI with the `ui.Interface.filenameToURI()` method, from the file located on the application server. This URI can then be used with the `playSound` front call.

When using a file name, it can be an absolute or relative path. In a client/server front-end configuration, if the sound file is located on the application server, it is in charge of the program to transfer the file on the front-end platform.

When executing an app on a mobile device, if it is not an absolute path, the sound file path is relative to the `appdir`.

**Tip:** When specifying a file path, pay attention to platform specific rules regarding directory separators and space characters in file names. When the front-end executes on a recent Microsoft™ Windows™ system, you can use the / slash character as directory separator, like on Unix systems. A directory or file name can contain spaces, and there is no need to surround the path with double quotes in such case. When using backslash directory separators, make sure to escape backslash characters in string literals with `\`.

**Example**

```javascript
CALL ui.Interface.frontCall("standard", "playSound", ["beep.mp3"], [])
```

**Related concepts**

- [Directory structure for GMA apps](#)
- [Directory structure for GMI apps](#)

**saveFile**
Displays a file dialog window to get a path to save a file on the local file system.

**Syntax**

```javascript
ui.Interface.frontCall("standard", "saveFile", [path, name, filetype, caption], [result])
```

1. *path* - The default path to the file like "/tmp/document.txt".
2. *name* - The label to be displayed for the file types / wildcards.
3. *filetype* - The file types (as a blank space-separated list of extensions).
4. *caption* - The caption to be displayed.
5. `result` - The name of the selected file (or NULL if canceled).

**Usage**

When invoking the "saveFile" front call, the front-end displays the file dialog window using the local file system, to let the end user enter a file path.

**Note:** The file dialog window rendering and features depend on the type of front end and the type of the front end platform (desktop OS, web browser).

If the user cancels the dialog, the front call returns NULL in the result variable.

**Tip:** When specifying a file path, pay attention to platform specific rules regarding directory separators and space characters in file names. When the front-end executes on a recent Microsoft™ Windows™ system, you can use the / slash character as directory separator, like on Unix systems. A directory or file name can contain spaces, and there is no need to surround the path with double quotes in such case. When using backslash directory separators, make sure to escape backslash characters in string literals with \\.

### Example

```main
MAIN
    DEFINE rec RECORD
        path STRING,
        name STRING,
        wildcards STRING,
        caption STRING
    END RECORD
    DEFINE result STRING
    LET rec.path = "/tmp/new_file.jpg"
    LET rec.name = "Image files"
    LET rec.wildcards = "*.jpg *.png"
    LET rec.caption = "Save file"
    CALL ui.Interface.frontCall("standard","saveFile",[rec.*],[result])

    IF result IS NULL THEN
        DISPLAY "No file name was entered."
    ELSE
        DISPLAY "File :", result
    END IF
END MAIN
```

**Related concepts**

- openDir on page 2658
  Displays a file dialog window to get a directory path on the local file system.

- openFile on page 2659
  Displays a file dialog window to let the user select a single file path on the local file system.

- openFiles on page 2660
  Displays a file dialog window to let the user select a list of file paths on the local file system.

- setReportFont
  Override the font used for GDC report generation for the current application (DBPRINT=FGLSERVER).

  **Syntax**

  ```
  ui.Interface.frontCall("standard", "setReportFont", [font], [result])
  ```

  1. `font` - A string describing the font to use for report generation.
2. **result** - The execution status (TRUE=success, FALSE=error).

**Usage**

The "setReportFont" front call overrides the default font used by GDC for report generation for the current application, when using **DBPRINT=FGLSERVER**.

An empty or null string resets it to the default behavior.

The **font** parameter is a string that describe the font to use for report generation. For example: "Helvetica, Bold, Italic, 13".

**Tip:** Copy/paste the font string from the "Report To Printer" font panel from GDC Monitor.

Alternatively, you can specify "<ASK_ONCE>" , "<ASK_ALWAYS>" , "<USER_DEFINED>" or "<USE_DEFAULT>" which will perform the corresponding actions.

**Example**

```plaintext
DEFINE result BOOLEAN
CALL ui.interface.frontCall("standard","setReportFont",
    ["Helvetica, Bold, Italic, 13"],
    [result])
```

**setReportPrinter**

Override the GDC printer configuration used for report generation for the current application (DBPRINT=FGLSERVER).

**Syntax**

```plaintext
ui.Interface.frontCall("standard", "setReportPrinter",
    [printer], [result])
```

1. **printer** - A string describing the printer to use for report generation.

2. **result** - The execution status (TRUE=success, FALSE=error).

**Usage**

The "setReportPrinter" front call overrides the printer configuration used for report generation for the current application, when using **DBPRINT=FGLSERVER**.

An empty or null string resets it to the default behavior.

The **printer** parameter is a string that describes the printer to use for report generation. For example: "moliere, Portrait, A4, 96 dpi, 1 copy, Ascendent, Color, Auto".

**Tip:** Copy/paste the printer string from the "Report To Printer" printer panel from GDC Monitor.

Alternatively, you can specify "<ASK_ONCE>" , "<ASK_ALWAYS>" , "<USER_DEFINED>" or "<USE_DEFAULT>" which will perform the corresponding actions.

**Example**

```plaintext
DEFINE result BOOLEAN
CALL ui.interface.frontCall("standard","setReportPrinter",
    ["moliere, Portrait, A4, 96 dpi, 1 copy, Ascendent, Color, Auto"],
    [result])
```
**setWebComponentPath**
Defines the base path where web components are located.

**Syntax**

```
ui.Interface.frontCall("standard", "setWebComponentPath", [path], [])
```

1. `path` - The base URL. For example, "http://myserver/components" or "file:///c:/components".

**Usage**
This front call defines the base path to find gICAPI web components files.

**Important**: The `standard.setWebComponentPath` front call is deprecated. Consider using deployment solutions described in Deploying the gICAPI web component files on page 1947.

For the Genero Desktop Client, it defines the base path where web components are located, when GDC is directly connected to the runtime system. This is ignored when GDC is connected to the GAS.

For Genero Mobile, it sets the main web component lookup path. An URI is expected. For example, "file:///data/data/com.fourjs.gma/cache/appdata/mywebcomponents" or "http://mygas/mywebcomponents".

**shellExec**
Opens a file on the front-end platform with the program associated to the file extension.

**Syntax**

```
ui.Interface.frontCall("standard", "shellExec", [document, action], [result])
```

1. `document` - The document file to be opened.
2. `action` - (optional, Windows® Only!) The action to perform, related to the way the file type is registered in Windows® Registry.
3. `result` - The execution status (TRUE = success, FALSE = error).

**Usage**
The "shellExec" front call opens a file on the front-end platform with the program associated to the file extension.

**Tip**: When specifying a file path, pay attention to platform specific rules regarding directory separators and space characters in file names. When the front-end executes on a recent Microsoft® Windows™ system, you can use the / slash character as directory separator, like on Unix systems. A directory or file name can contain spaces, and there is no need to surround the path with double quotes in such case. When using backslash directory separators, make sure to escape backslash characters in string literals with \.  

This front call is mainly designed for the Genero Desktop Client on Windows® platforms.

**Important**: Under X11 Systems, this uses `xdg-open`, which needs to be installed and configured on your system. `Kfmclient` will be used as a workaround when `xdg-open` is not available.

**Tip**: In order to view a document (like a PDF for example), if that document can be displayed by web browsers, use the `launchURL` on page 2657 front call instead, especially if you want to use both the Genero Desktop Client (GDC) and the Genero Browser Client (GBC) front-ends.
Web component front calls

This section describes web component specific front calls.

This table shows the functions provided by the "webcomponent" module.

Table 573: Webcomponent module front-end functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ui.Interface.frontCall(&quot;webcomponent&quot;, &quot;call&quot;, [&quot;aui-name&quot;, function-name, param1, param2, ...], [result])</td>
<td>Calls a JavaScript function through the web component.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;webcomponent&quot;, &quot;frontCallAPIVersion&quot;, [], [result])</td>
<td>Returns the API version of web component front-end calls.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;webcomponent&quot;, &quot;getTitle&quot;, [&quot;aui-name&quot;], [result])</td>
<td>Returns the title of the HTML doc rendered by a web component.</td>
</tr>
</tbody>
</table>

Related concepts

Web components on page 1922
This section describes how to use web components in your application.

call
Calls a JavaScript function through the web component.

Syntax

ui.Interface.frontCall("webcomponent", "call", ["aui-name", function-name, param1, param2, ...], [result])

1. aui-name - This is the name of the web component name in the AUI tree.
2. function-name - This is the name of the web component JavaScript function to be called.
3. param1, param2, ... - Optional parameters to be passed to the web component JavaScript function.
4. result - Holds the JavaScript function return value.

Usage

This front call executes a JavaScript function of the WEBCOMPONENT form field identified by aui-name.

Important: Use the webcomponent.call front call only for specific needs: With desktop and web front ends, a front call implies a network roundtrip and abstract user interface update that may cause unwanted delays. Prefer using gICAPI interface triggers such as onData / onProperty instead.

The JavaScript function identified by function-name must be implemented in the HTML content pointed by the URL-based web component, or in the user-defined JavaScript of a gICAPI-based web component.

The arguments following the function-name argument will be passed to the JavaScript function.

The result variable will contain the value returned by the JavaScript function.
When using simple data types in arguments, values are passed as is to the JavaScript function of the web component. When using RECORD or DYNAMIC ARRAY types, the runtime system converts the structured data to a JSON string. Similarly, if the JavaScript function returns a complex data structure in JSON notation, it must be used to assign a RECORD or DYNAMIC ARRAY.

For more details about JSON notation usage for complex data types, see the `ui.Interface.frontCall()` method.

**Example 1: Using a simple string parameter and return value:**

```plaintext
DEFINE result STRING
CALL ui.Interface.frontCall("webcomponent","call",
    ["formonly.mywebcomp","echoString","abcdef"],result)
```

**Example 2: Using structured variables to be converted to JSON for the JavaScript code:**

```plaintext
DEFINE options RECORD
    filter STRING,
    creation DATE
END RECORD,
LET options.filter = "abc*"
LET options.creation = TODAY
CALL ui.Interface.frontCall("webcomponent","call",
    ["formonly.mywebcomp","getItems", options ],
    [ items ])
```

For a complete example, see Example 1: Calling a JavaScript function on page 1954.

**frontCallAPIVersion**

Returns the API version of web component front-end calls.

**Syntax**

```plaintext
ui.Interface.frontCall("webcomponent", "frontCallAPIVersion",
    [],[result])
```

1. `result` - Holds the API version for web component front calls.

**Usage**

This front call can be used to check the API version for the web component front calls.

If the API version changes, you must adapt the code to the expected front call API implemented for the web components.

The value returned by this front call is typically a version number, such as 1.0, 1.1, etc.

**Example**

```plaintext
FUNCTION wc_api_version()
    DEFINE vers STRING
    TRY
        CALL ui.Interface.frontCall("webcomponent","frontCallAPIVersion", [],[vers])
        -- we can safely call "webcomponent" "call" in the code
        RETURN vers
    CATCH
        -- we cannot call the "webcomponent" functions...
        RETURN 0
    END TRY
```
getTitle
Returns the title of the HTML doc rendered by a web component.

Syntax

ui.Interface.frontCall("webcomponent", "getTitle", [aui-name], [result])

1. aui-name - This is the name of the web component in the AUI tree.
2. result - Holds the title of the HTML document.

Usage

This front call can be used to get the title of the HTML document that is rendered by the web component identified by the aui-name. For more details refer to http://www.w3schools.com/tags/tag_title.asp.

A typical usage of this front call is when implementing a web component based on the O-Auth mechanism to identify the current user. For example, with the Google accounts authentication service, after the login and password have been validated by Google, the authentication token is returned in the title of the HTML document. This token is typically used by the application to identify the user in remote API calls.

Genero Desktop Client front calls

This section describes GDC-specific front calls.

GDC Monitor Front Calls

This section describes front calls specific to the GDC monitor.

The GDC monitor is the administration component of the Genero Desktop Front-End. This component allows you to configure the GDC and do setup tasks.

The following table shows the functions implemented by all front-ends in the "monitor" module.

Table 574: monitor module front-end functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ui.Interface.frontCall(&quot;monitor&quot;, &quot;update&quot;, [path-to-update-file [,warning-text [,elevation-prompt]]], [result])</code></td>
<td>Starts the GDC update.</td>
</tr>
</tbody>
</table>

update

Starts the GDC update.

Syntax

ui.Interface.frontCall("monitor", "update", [path-to-update-file [,warning-text [,elevation-prompt]]], [result])

1. path-to-update-file - Defines the path to the zip archive containing the update material.
2. warning-text - The warning to be displayed to the user before updating the GDC.
3. **elevation-prompt** - A boolean to indicate if the MS Windows elevation prompt must be displayed, when the GDC installation requires administrator privileges. When set to true and the update requires permissions elevation, the elevation prompt is displayed. When set to false and the update requires permission elevation, the update will fail. The parameter is ignored, if the update does not require permission elevation. Default is false.

4. **result** - The execution status (TRUE=success, FALSE=error).

**Usage**

The "update" front call will start the update process based on the specified file. This file is expected to have been pushed previously on the GDC (generally using the fgl_putfile() built-in function).

For more details about this feature, see the Genero Desktop Client User Guide, in the Auto-Update section.

**Example**

```plaintext
MAIN
  DEFINE res BOOLEAN
  MENU "Update GDC"
    COMMAND "Do Update"
      CALL ui.Interface.frontCall( "monitor", "update",
        [ "C:\\tmp\\gdc-package.zip",
          "GDC update is required",
          TRUE ], [ res ]
      )
    COMMAND "Quit"
  EXIT MENU
END MAIN
```

**Related concepts**

- fgl_putfile() on page 2291
  Transfers a file from the virtual machine context to the front-end context.

**Windows DDE Support**

Description of Windows® DDE support.

**Important**: This feature is deprecated, its use is discouraged although not prohibited.

Dynamic Data Exchange (DDE) is a form of inter-process communication implemented by Microsoft™ for Windows® platforms. DDE uses shared memory to exchange data between applications. Applications can use DDE for one-time data transfers and for ongoing exchanges in applications that send updates to one another as new data becomes available.

Please refer to your Microsoft™ documentation for DDE compatibility between existing versions. As an example, DDE commands were changed between Office 97 and Office 98.

We provide a DDE interface as a Front-End Extension: WinDDE.DLL

**Using the WinDDE API**

With WinDDE support, you can invoke a Windows® application and send or receive data to or from it. To use this functionality, the program must use the Windows® front-end.

Before using the DDE functions, the TCP communication channel between the application and the front-end must be established with a display (OPEN WINDOW, MENU, DISPLAY TO).
The DDE API is used in a four-part procedure, as described in the following steps:

1. The application sends to the front-end the DDE order using the TCP/IP channel.
2. The front-end executes the DDE order and sends the data to the Windows® application through the DDE API.
3. The Windows® application executes the command and sends the result, which can be data or an error code, to the front-end.
4. The Windows® front-end sends back the result to the application using the TCP/IP channel.

A DDE connection is uniquely identified by two values: the name of the DDE Application, and the document. Most DDE functions require these two values to identify the DDE source or target.

**WINDDE API front calls**

The DDE API is based on the front call technique.

The DDE API is based on the front call technology. All DDE functions are grouped in the WINDDE front-end function module.

**Table 575: Windows DDE front-end functions**

<table>
<thead>
<tr>
<th>Function name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Important: This feature is deprecated, its use is discouraged although not prohibited.</td>
<td>DDEConnect opens a DDE connection. DDEExecute executes a DDE command.</td>
</tr>
<tr>
<td>CALL</td>
<td>ui.Interface.frontCall(&quot;WINDDE&quot;, &quot;DDEConnect&quot;, [program, document, encoding], [result])</td>
</tr>
<tr>
<td>Important: This feature is deprecated, its use is discouraged although not prohibited.</td>
<td>CALL</td>
</tr>
</tbody>
</table>

**Figure 120: The four-part procedure of the DDE API**

The four parts of the DDE API are illustrated in the figure. The application sends a DDE order to the front-end using the TCP/IP channel. The front-end executes the order, sends data to the Windows® application, and sends the result back to the application. A DDE connection is uniquely identified by the name of the DDE Application and the document.
<table>
<thead>
<tr>
<th>Function name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Important:</strong> This feature is deprecated, its use is discouraged although not prohibited.</td>
<td></td>
</tr>
<tr>
<td>CALL</td>
<td>DDEFinish closes a DDE connection.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;WINDDE&quot;,&quot;DDEFinish&quot;, [ program, document ], [result] )</td>
<td></td>
</tr>
<tr>
<td><strong>Important:</strong> This feature is deprecated, its use is discouraged although not prohibited.</td>
<td></td>
</tr>
<tr>
<td>CALL</td>
<td>DDEFinishAll closes all DDE connections.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;WINDDE&quot;,&quot;DDEFinishAll&quot;, [], [result] )</td>
<td></td>
</tr>
<tr>
<td><strong>Important:</strong> This feature is deprecated, its use is discouraged although not prohibited.</td>
<td></td>
</tr>
<tr>
<td>CALL</td>
<td>DDEError returns error information about the last DDE operation.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;WINDDE&quot;,&quot;DDEError&quot;, [], [errmsg] )</td>
<td></td>
</tr>
<tr>
<td><strong>Important:</strong> This feature is deprecated, its use is discouraged although not prohibited.</td>
<td></td>
</tr>
<tr>
<td>CALL</td>
<td>DDEPeek retrieves data from the specified program and document using the DDE channel.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;WINDDE&quot;,&quot;DDEPeek&quot;, [ program, container, cells, encoding ], [ result, value ] )</td>
<td></td>
</tr>
<tr>
<td><strong>Important:</strong> This feature is deprecated, its use is discouraged although not prohibited.</td>
<td></td>
</tr>
<tr>
<td>CALL</td>
<td>DDEPoke sends data to the specified program and document using the DDE channel.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;WINDDE&quot;,&quot;DDEPoke&quot;, [ program, container, cells, values, encoding ], [result] )</td>
<td></td>
</tr>
</tbody>
</table>

**DDEConnect**

*DDEConnect* opens a DDE connection.

**Syntax**

**Important:** This feature is deprecated, its use is discouraged although not prohibited.

CALL ui.Interface.frontCall("WINDDE","DDEConnect", [ program, document, encoding ], [result] )

- `program` is the name of the DDE application.
- `document` is the document that is to be opened.
- `encoding` is an optional parameter. It allows to force the encoding to use between ASCII and wide char/unicode. When not specified, WinDDE will try to retrieve the correct encoding by itself. Possible values are:
• UNICODE
• ASCII

• result is an integer variable receiving the status.
• result is TRUE if the function succeeded, FALSE otherwise.
• If the function fails, use DDEError to get the description of the error.

Warnings
• If the function fails with DMLERR_NO_CONV_ESTABLISHED, then the DDE application was probably not running. Use the execute or shellexec front call to start the DDE application.
• In Microsoft™ Office 2010, the use of DDE is disabled by default. You need to uncheck Ignore other applications that use Dynamic Data Exchange(DDE) in advanced options, otherwise DDEConnect will fail.

DDEExecute
DDEExecute executes a DDE command.

Syntax
Important: This feature is deprecated, its use is discouraged although not prohibited.

CALL ui.Interface.frontCall("WINDDE", "DDEExecute", [ program, document, command, encoding ], [result] )

• program is the name of the DDE application.
• document is the document that is to be used.
• command is the command that needs to be executed.
• encoding is an optional parameter. It allows you to force the encoding to use between ASCII and wide char/unicode. When not specified, WinDDE will try to retrieve the correct encoding itself. Possible values are: "UNICODE", "ASCII"
• Refer to the program documentation to know the syntax of command.
• result is an integer variable receiving the status.
• result is TRUE if the function succeeded, FALSE otherwise.
• If the function fails, use DDEError to get the description of the error.

Warnings
• The DDE connection must be opened. See DDEConnect.

DDEFinish
DDEFinish closes a DDE connection.

Syntax
Important: This feature is deprecated, its use is discouraged although not prohibited.

CALL ui.Interface.frontCall("WINDDE", "DDEFinish", [ program, document ], [result] )

• program is the name of the DDE application.
• document is the document that is to be closed.
• result is an integer variable receiving the status.
• result is TRUE if the function succeeded, FALSE otherwise.
• If the function fails, use DDEError to get the description of the error.
**Warnings**

- The DDE connection must be opened, see `DDEConnect`.

### DDEFinishAll

`DDEFinishAll` closes all DDE connections.

**Syntax**

**Important:** This feature is deprecated, its use is discouraged although not prohibited.

```plaintext
CALL ui.Interface.frontCall("WINDE", "DDEFinishAll", [], [result])
```

- `result` is TRUE if the function succeeded, FALSE otherwise.

**Usage**

This function closes all DDE connections, as well as the DDE server program.

### DDEError

`DDEError` returns error information about the last DDE operation.

**Syntax**

**Important:** This feature is deprecated, its use is discouraged although not prohibited.

```plaintext
CALL ui.Interface.frontCall("WINDE", "DDEError", [], [errmsg])
```

- `errmsg` is the error message. It is set to NULL if no error occurred.

### DDEPeek

`DDEPeek` retrieves data from the specified program and document using the DDE channel.

**Syntax**

**Important:** This feature is deprecated, its use is discouraged although not prohibited.

```plaintext
CALL ui.Interface.frontCall("WINDE", "DDEPeek", [program, container, cells, encoding], [result, value])
```

- `program` is the name of the DDE application.
- `container` is the document or sub-document that is to be used. A sub-document can, for example, be a sheet in Microsoft™ Excel.
- `cells` represents the working items; see the `program` documentation to know the format of `cells`.
- `encoding` is an optional parameter. It allows to force the encoding to use between ASCII and wide char/unicode. When not specified, WinDDE will try to retrieve the correct encoding itself. Possible values are: "UNICODE", "ASCII"
- `value` represents the data to be retrieved; see the `program` documentation for details of the format of `values`.
- `result` is an integer variable receiving the status.
- `result` is TRUE if the function succeeded, FALSE otherwise.
- If the function fails, use `DDEError` to get the description of the error.
- `value` is a variable receiving the cells values.

**Warnings**

- The DDE connection must be opened; see `DDEConnect`.
- `DDEError` can only be called once to check if an error occurred.
**DDEPoke**
DDEPoke sends data to the specified program and document using the DDE channel.

**Syntax**

**Important:** This feature is deprecated, its use is discouraged although not prohibited.

```
CALL ui.Interface.frontCall("WINDDE", "DDEPoke", [program, container, cells, values, encoding], [result])
```

- **program** is the name of the DDE application.
- **container** is the document or sub-document that is to be used. A sub-document can, for example, be a sheet in Microsoft™ Excel.
- **cells** represents the working items; see the program documentation to know the format of cells.
- **values** represents the data to be sent; see the program documentation to know the format of values.
- **encoding** is an optional parameter. It allows to force the encoding to use between ASCII and wide char/unicode. When not specified, WinDDE will try to retrieve the correct encoding itself. Possible values are: "UNICODE", "ASCII"
- **result** is an integer variable receiving the status.
- **result** is TRUE if the function succeeded, FALSE otherwise.
- If the function fails, use DDEError to get the description of the error.

**Warnings**

- The DDE connection must be opened; see DDEConnect.
- An error may occur if you try to set many (thousands of) cells in a single operation.

**WinDDE example**
This section provides a WinDDE example.

**dde_example.per**

```
DATABASE formonly
SCREEN
{
Value to be given to top-left corner :
  [f00                                
Value found on top-left corner :
  [f01                                
}
ATTRIBUTES
  f00 = formonly.val;
  f01 = formonly.rval, NOENTRY;
```

**dde_example.4gl**

```
MAIN
  -- Excel must be open beforehand
CONSTANT file = "Sheet1"
CONSTANT prog = "EXCEL"
DEFINE val, rval STRING
DEFINE res INTEGER
OPEN WINDOW w1 AT 1,1 WITH FORM "dde_example.per"
INPUT BY NAME val
CALL ui.Interface.frontCall("WINDDE", "DDEConnect", [prog,file], [res])
CALL checkError(res)
CALL ui.Interface.frontCall("WINDDE", "DDEPoke", [prog,file,"R1C1",val], [res]);
CALL checkError(res)
```
CALL ui.Interface.frontCall("WINDDE","DDEPeek", [prog,file,"R1C1"], [res,rval] );
CALL checkError(res)
DISPLAY BY NAME rval
INPUT BY NAME val WITHOUT DEFAULTS
CALL ui.Interface.frontCall("WINDDE","DDEExecute", [prog,file,"[save]"], [res]);
CALL checkError(res)
CALL ui.Interface.frontCall("WINDDE","DDEFinish", [prog,file], [res] );
CALL checkError(res)
CALL ui.Interface.frontCall("WINDDE","DDEFinishAll", [], [res] );
CALL checkError(res)
CLOSE WINDOW w1
END MAIN

FUNCTION checkError(res)
DEFINE res INTEGER
DEFINE mess STRING
IF res THEN RETURN END IF
DISPLAY "DDE Error:"
CALL ui.Interface.frontCall("WINDDE","DDEError",[],[mess]);
DISPLAY mess
CALL ui.Interface.frontCall("WINDDE","DDEFinishAll", [], [res] );
DISPLAY "Exit with DDE Error."
EXIT PROGRAM (-1)
END FUNCTION

Windows™ COM Support
“COM” stands for Component Object Model. It allows anyone to directly access Windows™ Applications Objects. You can create instances of those objects, call methods on them, and get or set their properties.

Important: This feature is deprecated, its use is discouraged although not prohibited.

Using the WinCOM API
With WinCOM Support, you can invoke a Windows® application and send or receive data to or from it.

Prerequisites
To use this functionality, the program must use the GDC front-end.

Supported syntax
COM language syntax is very flexible and allows lots of notation. Genero WinCOM API is slightly more strict:

- := notation is allowed in version 2.00.1e (or later) only; for instance: myFunction(SourceType:=3)
- “no parenthesis” notation is not allowed; for instance: myFunction 3 must be written myFunction(3)
- numeric constants are allowed in version 2.00.1e (or later) only. The constant list depends on the application used via WinCOM, therefore the list is configurable. A file named etc/WinCOM.cst gathers all the constants provided today by Microsoft™ for Office XP. It can be modified to add user-defined constants. Example with Word:


Here, “9999998” stands for the constant wdToggle (see etc/WinCOM.cst).
- There is no way to handle an array as a method argument. This is also due to BDL limitation: you can’t pass BDL Arrays to front calls.
**WINCOM API front calls**

The WinCOM API is based on the front call technique as described in front-end functions. All WinCOM functions are grouped in the WinCOM front-end function module.

### Table 576: Windows® COM front-end functions

<table>
<thead>
<tr>
<th>Function name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Important: This feature is deprecated, its use is discouraged although not prohibited.</td>
<td>The CreateInstance function creates an instance of a registered COM object.</td>
</tr>
<tr>
<td>CALL</td>
<td><code>ui.Interface.frontCall(&quot;WINCOM&quot;,&quot;CreateInstance&quot;, [program], [handle] )</code></td>
</tr>
</tbody>
</table>

| Important: This feature is deprecated, its use is discouraged although not prohibited. | The CallMethod function calls a method on a specified object. |
| CALL                       | `ui.Interface.frontCall("WINCOM","CallMethod", [ handle, method, arg1, ... ], [result] )` |

| Important: This feature is deprecated, its use is discouraged although not prohibited. | The GetProperty function gets a property of an object. |
| CALL                       | `ui.Interface.frontCall("WINCOM","GetProperty", [ handle, member ], [result] )` |

| Important: This feature is deprecated, its use is discouraged although not prohibited. | The SetProperty function sets a property of an object. |
| CALL                       | `ui.Interface.frontCall("WINCOM","SetProperty", [ handle, member, value ], [result] )` |

| Important: This feature is deprecated, its use is discouraged although not prohibited. | The GetError function gets a description of the last error which occurred. |
| CALL                       | `ui.Interface.frontCall("WINCOM","GetError", [], [result] )` |

| Important: This feature is deprecated, its use is discouraged although not prohibited. | The ReleaseInstance function releases an instance of a COM object. |
| CALL                       | `ui.Interface.frontCall("WINCOM","ReleaseInstance", [handle], [result] )` |
CreateInstance
The CreateInstance function creates an instance of a registered COM object.

Syntax
Important: This feature is deprecated, its use is discouraged although not prohibited.

CALL ui.Interface.frontCall("WINCOM","CreateInstance", [program], [handle])

- program is the class name of the registered COM object.
- handle is an integer variable receiving the status.
- handle is -1 if there is an error, otherwise an integer value that can be used for a later call to the API.
- If the function fails, use GetLastError to get the description of the error.

CallMethod
The CallMethod function calls a method on a specified object.

Syntax
Important: This feature is deprecated, its use is discouraged although not prohibited.

CALL ui.Interface.frontCall("WINCOM","CallMethod", [handle, method, arg1, ...], [result])

- handle is the handle returned by another front call (CreateInstance, CallMethod, GetProperty).
- method is the WINCOM method to call.
- arg1 (and ...) are the arguments to pass to the method call.
- result is either a handle or a value of a predefined type. -1 in case of error.

Usage
Depending on the syntax allowed by the version of the program you're interacting with, arguments might be used inside brackets or outside.

The best way for Microsoft™ applications is to initially test your code with a macro of the manipulation you're expecting to do. Depending on the method used, arguments may or may not be optional.

In case of error, result is set to -1. Use GetLastError to get the description of the error.

GetProperty
The GetProperty function gets a property of an object.

Syntax
Important: This feature is deprecated, its use is discouraged although not prohibited.

CALL ui.Interface.frontCall("WINCOM","GetProperty", [handle, member], [result])

- handle is the handle returned by another front call (CreateInstance, CallMethod, GetProperty).
- member is the member property name to get.
- result is either a handle or a value of a predefined type.
- result is -1 in case of error (use GetLastError to get the description of the error).
setProperty

The `setProperty` function sets a property of an object.

Syntax

**Important:** This feature is deprecated, its use is discouraged although not prohibited.

```plaintext
CALL ui.Interface.frontCall("WINCOM", "setProperty",
    [ handle, member, value ], [ result ] )
```

- `handle` is the handle returned by another front call (`createinstance`, `callmethod`, `getProperty`).
- `member` is the member property name to set.
- `value` is the value to which the property will be set.
- `result` is -1 in case of error (use `geterror` to get the description of the error), otherwise it is 0.

geterror

The `geterror` function gets a description of the last error which occurred.

Syntax

**Important:** This feature is deprecated, its use is discouraged although not prohibited.

```plaintext
CALL ui.Interface.frontCall("WINCOM", "geterror",
    [], [ result ] )
```

- `result` is the description of the last error.
- the returned value is NULL if there was no error.

releaseinstance

The `releaseinstance` function releases an instance of a COM object.

Syntax

**Important:** This feature is deprecated, its use is discouraged although not prohibited.

```plaintext
CALL ui.Interface.frontCall("WINCOM", "releaseinstance",
    [ handle ], [ result ] )
```

- `handle` is the handle returned by another front call (`createinstance`, `callmethod`, `getProperty`).
- `result` is -1 in case of error (use `geterror` to get the description of the error), otherwise it is 0.

WinCOM examples

Various WinCOM examples.

**Wincom and Excel example**

This example puts "foo" in the first row of the 1st column of an Excel Sheet.

```plaintext
DEFINE xlapp INTEGER
DEFINE x1wb INTEGER
MAIN
    DEFINE result INTEGER
    DEFINE str STRING
    --initialization of global variables
    LET xlapp = -1
    LET x1wb = -1
    --first, we must create an Instance of an Excel Application
    CALL ui.Interface.frontCall("WINCOM", "CreateInstance",
        ["Excel.Application"], [xlapp])
    CALL CheckError(xlapp, __LINE__)
    --then adding a Workbook to the current document
    CALL ui.interface.frontCall("WINCOM", "CallMethod",
        ["Application", "CreateWindow", 0], [result])
    CALL CheckError(result, __LINE__)
    --next, we will call a method of the Workbook
    CALL ui.Interface.frontCall("WINCOM", "CallMethod",
        ["Worksheet(1)", "Write", "foo"], [result])
    CALL CheckError(result, __LINE__)
    --then release the instance
    CALL ui.Interface.frontCall("WINCOM", "ReleaseInstance",
        [xlapp], [result])
```
CALL CheckError(xlwb, __LINE__)  
--then, setting it to be visible  
CALL ui.interface.frontCall("WinCOM", "SetProperty",  
[xlapp, "Visible", true], [result])  
CALL CheckError(result, __LINE__)  
--then CALL SetProperty to set the value of the cell  
CALL ui.interface.frontCall("WinCOM", "SetProperty",  
[xlwb, 'activesheet.Range("A1").Value', "foo"], [result])  
CALL CheckError(result, __LINE__)  
--then CALL GetProperty to check the value again  
CALL ui.interface.frontCall("WinCOM", "GetProperty",  
[xlwb, 'activesheet.Range("A1").Value'], [str])  
CALL CheckError(str, __LINE__)  
DISPLAY "content of the cell is: " || str  
--then Free the memory on the client side  
CALL freeMemory()  
END MAIN

FUNCTION freeMemory()  
DEFINE res INTEGER  
IF xlwb != -1 THEN  
CALL ui.Interface.frontCall("WinCOM","ReleaseInstance", [xlwb], [res] )  
END IF  
IF xlapp != -1 THEN  
CALL ui.Interface.frontCall("WinCOM","ReleaseInstance", [xlapp], [res] )  
END IF  
END FUNCTION

FUNCTION checkError(res, lin)  
DEFINE res INTEGER  
DEFINE lin INTEGER  
DEFINE mess STRING  
IF res = -1 THEN  
DISPLAY "COM Error for call at line: ", lin  
CALL ui.Interface.frontCall("WinCOM","GetError",[],[mess])  
DISPLAY mess  
--let's release the memory on the GDC side  
CALL freeMemory()  
DISPLAY "Exit with COM Error."  
EXIT PROGRAM (-1)  
END IF  
END FUNCTION

**Wincom and Word example**

This example puts "This is a title" centered on the page, underlined, and in bold.

```plaintext
DEFINE wdapp INTEGER  
DEFINE wddoc INTEGER  
MAIN  
  DEFINE result INTEGER  
  --initialization of global variables  
  LET wdapp = -1  
  LET wddoc = -1  
  --first, we must create an Instance of a Word Application  
  CALL ui.interface.frontCall("WINCOM","CreateInstance",  
    ["Word.Application"],[wdapp])  
  CALL CheckError(wdapp, __LINE__)  
  --then adding a document  
  CALL ui.interface.frontCall("WINCOM","CallMethod",  
    [wdapp,"Documents.Add"],[wddoc])
```
CALL CheckError(wddoc, __LINE__)  
--then, setting it to be visible  
CALL ui.Interface.frontCall("WINCOM","SetProperty",  
    [wdapp,"Visible",true],[result])  
CALL CheckError(result, __LINE__)  
--Centering the cursor for the title  
CALL ui.Interface.frontCall("WINCOM","SetProperty",  
    [wdapp,"Selection.ParagraphFormat.Alignment","1"],[wddoc])  
CALL CheckError(wddoc, __LINE__)  
--Underlining the title  
CALL ui.Interface.frontCall("WINCOM","SetProperty",  
    [wdapp,"Selection.Font.Underline","1"],[wddoc])  
CALL CheckError(wddoc, __LINE__)  
--Putting the title in bold  
CALL ui.Interface.frontCall("WINCOM","SetProperty",  
    [wdapp,"Selection.Font.Bold","9999998"],[wddoc])  
CALL CheckError(wddoc, __LINE__)  
--Typing the title's text  
CALL ui.Interface.frontCall("WINCOM","CallMethod",  
    [wdapp,'Selection.TypeText("This is a title")'],[wddoc])  
CALL CheckError(wddoc, __LINE__)  
--then Free the memory on the client side  
CALL freeMemory()  
END MAIN

FUNCTION freeMemory()  
DEFINE res INTEGER  
IF wddoc != -1 THEN  
    CALL ui.Interface.frontCall("WinCOM","ReleaseInstance", [wddoc],[res])  
END IF  
IF wdapp != -1 THEN  
    CALL ui.Interface.frontCall("WinCOM","ReleaseInstance", [wdapp],[res])  
END IF  
END FUNCTION

FUNCTION checkError(res, lin)  
DEFINE res INTEGER  
DEFINE lin INTEGER  
DEFINE mess STRING  
IF res = -1 THEN  
    DISPLAY "COM Error for call at line: ", lin  
    CALL ui.Interface.frontCall("WinCOM","GetError",[],[mess])  
    DISPLAY mess  
END IF  
--let's release the memory on the GDC side  
CALL freeMemory()  
DISPLAY "Exit with COM Error."  
EXIT PROGRAM (-1)  
END FUNCTION

Wincom and Outlook example

This example executes Outlook, creates a new contact, and saves it in your contact list.

DEFINE outapp INTEGER  
DEFINE outit INTEGER  
DEFINE outcon INTEGER  
DEFINE outsav INTEGER  
MAIN  
DEFINE result INTEGER  
DEFINE str STRING  
-- initialization of global variables  
LET outapp = -1
LET outit = -1
LET outcon = -1
LET outsav = -1

-- first, we must create an Instance of an Outlook Application
CALL ui.interface.frontcall("WinCOM", "CreateInstance", 
    ["Outlook.Application"], [outapp])
CALL CheckError(outapp, __LINE__)

-- then, creating a contact object
CALL ui.interface.frontcall("WinCOM", "CallMethod", 
    [outapp, "CreateItem(olContactItem)", [outit])
CALL CheckError(outit, __LINE__)

-- then, displaying the contact form
CALL ui.interface.frontCall("WinCOM", "CallMethod", 
    [outit, "Display"], [outcon])
CALL CheckError(outcon, __LINE__)

-- CALL SetProperty to fill the various fields with the values you expect
# First Name
CALL ui.interface.frontCall("WinCOM", "SetProperty", 
    [outit, "FirstName", "Lionel"], [result])
CALL CheckError(result, __LINE__)

# 1st email address
CALL ui.interface.frontCall("WinCOM", "SetProperty", 
    [outit, "Email1Address", "lif@4js.com"], [result])
CALL CheckError(result, __LINE__)

# Business address
CALL ui.interface.frontCall("WinCOM", "SetProperty", 
    [outit, "BusinessAddress", "1 rue de Berne"], [result])
CALL CheckError(result, __LINE__)

-- then, CALL GetProperty to check the values again
CALL ui.Interface.frontCall("WinCOM", "GetProperty", 
    [outit, "FirstName"], [str])
CALL CheckError(str, __LINE__)
DISPLAY "First Name of the new contact is " || str
CALL ui.Interface.frontCall("WinCOM", "GetProperty", 
    [outit, "Email1Address"], [str])
CALL CheckError(str, __LINE__)
DISPLAY "1st email of the new contact is " || str
CALL ui.Interface.frontCall("WinCOM", "GetProperty", 
    [outit, "BusinessAddress"], [str])
CALL CheckError(str, __LINE__)
DISPLAY "Business Address of the new contact is " || str

-- at the end, saving the contact
CALL ui.interface.frontCall("WinCOM", "CallMethod", [outit, "Save"], 
    [outsav])
CALL CheckError(outsav, __LINE__)

-- then Free the memory on the client side
CALL freeMemory()

END MAIN

FUNCTION freeMemory()
DEFINE res INTEGER
IF outit != -1 THEN
    CALL ui.Interface.frontCall("WinCOM", "ReleaseInstance", [outit], [res] )
END IF
IF outapp != -1 THEN
    CALL ui.Interface.frontCall("WinCOM", "ReleaseInstance", [outapp], [res] )
END IF
END FUNCTION

FUNCTION checkError(res, lin)
DEFINE res INTEGER
DEFINE lin INTEGER
DEFINE mess STRING
IF res = -1 THEN
    DISPLAY "COM Error for call at line: ", lin
    CALL ui.Interface.frontCall("WinCOM", "GetError", [], [mess])
    DISPLAY mess
    -- let's release the memory on the GDC side
    CALL freeMemory()
    DISPLAY "Exit with COM Error."
    EXIT PROGRAM (-1)
END IF
END FUNCTION

Tip: You may find the various Outlook objects (such as ContactItem object), methods (such as the CreateItem method), and properties (such as the FirstName or BusinessAddress properties) on the Microsoft™ Developer Network.

Wincom and Internet Explorer example

This example executes Internet Explorer on a defined URL with the address bar masked.

DEFINE ieapp INTEGER
DEFINE ienav INTEGER
MAIN
    DEFINE result INTEGER
    -- initialization of global variables
    LET ieapp = -1
    LET ienav = -1
    -- first, we must create an Instance of Internet Explorer application
    CALL ui.Interface.frontCall("WinCOM", "CreateInstance", 
        ["InternetExplorer.Application"], [ieapp])
    CALL CheckError(ieapp, __LINE__)
    -- then, specifying the URL you want to load
    CALL ui.Interface.frontCall("WinCOM", "CallMethod", 
        [ieapp, "Navigate", "www.4js.com"], [ienav])
    CALL CheckError(ienav, __LINE__)
    -- then, masking the address bar
    CALL ui.Interface.frontCall("WinCOM", "SetProperty", 
        [ieapp, "AddressBar", false], [result])
    CALL CheckError(result, __LINE__)
    -- then, setting it to visible
    CALL ui.Interface.frontCall("WinCOM", "SetProperty", [ieapp, "Visible", true], [result])
    CALL CheckError(result, __LINE__)
    -- then Free the memory on the client side
    CALL freeMemory()
END MAIN

FUNCTION freeMemory()
    DEFINE res INTEGER
    IF ienav != -1 THEN
        CALL ui.Interface.frontCall("WinCOM", "ReleaseInstance", [ienav], [res])
    END IF
    IF ieapp != -1 THEN
        CALL ui.Interface.frontCall("WinCOM", "ReleaseInstance", [ieapp], [res])
    END IF
END FUNCTION

FUNCTION checkError(res, lin)
    DEFINE res INTEGER
    DEFINE lin INTEGER
    DEFINE mess STRING
    IF res = -1 THEN
        DISPLAY "COM Error for call at line: ", lin
    END FUNCTION
CALL ui.Interface.frontCall("WinCOM","GetError",[],[mess])
DISPLAY mess
--let's release the memory on the GDC side
CALL freeMemory()
DISPLAY "Exit with COM Error."
EXIT PROGRAM (-1)
END IF
END FUNCTION

**Windows Mail extension**

Description of the Windows® Mail extension.

**Important:** This feature is deprecated, its use is discouraged although not prohibited.

**Send mail using MAPI**

MAPI is an acronym for Messaging Application Programming Interface. The MAPI extension creates a new mail in the default mailer software, which needs to be "MAPI" compatible, and requests user permission to send the mail. The mail sent using MAPI will be stored by the default mailer software in the same way as any other mail created by the user.

**Send mail using an SMTP server**

Another method of sending mail is to connect directly to a Simple Mail Transfer Protocol (SMTP) server (SMTP is the de facto standard for email transmission across the Internet). The extension will connect to a given SMTP server and send the mail through this server. The mail is not kept on the client side.

**Binary format of GDC and WinMAIN DLL**

**Important:** Since GDC and WinMAIN DLL binary format must match, the 32-bit WinAPI can only be used with a 32-bit GDC.

**WinMail API front calls**

The WinMail API is based on the front call technique as described in front-end functions. All WinMail functions are grouped in the WinMail front-end function module.

**Table 577: WinMail front-end functions: General**

<table>
<thead>
<tr>
<th>Function name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Important:</strong> This feature is deprecated, its use is discouraged although not prohibited.</td>
<td></td>
</tr>
<tr>
<td>CALL ui.Interface.frontCall(&quot;WinMail&quot;,&quot;Init&quot;, [], [id ] )</td>
<td>The Init function initializes the module.</td>
</tr>
<tr>
<td>• ret is the identifier of the message initialized.</td>
<td></td>
</tr>
<tr>
<td>• For each Init function, a Close must be called.</td>
<td></td>
</tr>
<tr>
<td><strong>Important:</strong> This feature is deprecated, its use is discouraged although not prohibited.</td>
<td></td>
</tr>
<tr>
<td>CALL ui.Interface.frontCall(&quot;WinMail&quot;,&quot;Close&quot;, [id], [ result ] )</td>
<td>The Close function clears all information corresponding to a message, and frees the memory occupied by the message.</td>
</tr>
<tr>
<td>Function name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Important</strong>: This feature is deprecated, its use is discouraged although not prohibited.</td>
<td>The <code>SetBody</code> function sets the body of the mail.</td>
</tr>
<tr>
<td>CALL</td>
<td>ui.Interface.frontCall(&quot;WinMail&quot;,&quot;SetBody&quot;, [ id, body ], [ result ] )</td>
</tr>
</tbody>
</table>

**Important**: This feature is deprecated, its use is discouraged although not prohibited.

CALL | ui.Interface.frontCall("WinMail","SetSubject", [ id, subject ], [ result ] ) | The `SetSubject` function sets the subject of the mail.                            |

**Important**: This feature is deprecated, its use is discouraged although not prohibited.

CALL | ui.Interface.frontCall("WinMail","AddTo", [ id, name, address ], [ result ] ) | The `AddTo` function adds a "To" addressee to the mail.                            |

**Important**: This feature is deprecated, its use is discouraged although not prohibited.

CALL | ui.Interface.frontCall("WinMail","AddCC", [ id, name, address ], [ result ] ) | The `AddCC` function adds a "CC" addressee to the mail.                            |

**Important**: This feature is deprecated, its use is discouraged although not prohibited.

CALL | ui.Interface.frontCall("WinMail","AddBCC", [ id, name, address ], [ result ] ) | The `AddBCC` function adds a "BCC" addressee to the mail.                           |

**Important**: This feature is deprecated, its use is discouraged although not prohibited.

CALL | ui.Interface.frontCall("WinMail","AddAttachment", [ id, fileName ], [ result ] ) | The `AddAttachment` function adds a file as an attachment to the mail.             |
<table>
<thead>
<tr>
<th>Function name</th>
<th>Description</th>
</tr>
</thead>
</table>
| **Important:** This feature is deprecated, its use is discouraged although not prohibited. | CALL `ui.Interface.frontCall("WinMail","SendMailSMTP", [ id ], [result] )`  
The `SendMailSMTP` function sends the mail with the SMTP protocol. |
| **Important:** This feature is deprecated, its use is discouraged although not prohibited. | CALL `ui.Interface.frontCall("WinMail","SendMailMAPI", [ id ], [result] )`  
The `SendMailMAPI` function sends the mail with the MAPI protocol. |
| **Important:** This feature is deprecated, its use is discouraged although not prohibited. | CALL `ui.Interface.frontCall("WinMail","GetError", [ id ], [result] )`  
The `GetError` function gets a description of the last error that occurred. |

The following functions are needed when you use SMTP server connections:

### Table 578: WinMail front-end functions: SMTP-specific

<table>
<thead>
<tr>
<th>Function name</th>
<th>Description</th>
</tr>
</thead>
</table>
| **Important:** This feature is deprecated, its use is discouraged although not prohibited. | CALL `ui.Interface.frontCall("WinMail","SetSmtp", [ id, smtp:port ], [result] )`  
The `SetSmtp` function sets the SMTP server to be used. |
| **Important:** This feature is deprecated, its use is discouraged although not prohibited. | CALL `ui.Interface.frontCall("WinMail","SetFrom", [ id, name, address ], [result] )`  
The `SetFrom` function sets sender information. |

**Init**  
The `Init` function initializes the module.

**Syntax**

**Important:** This feature is deprecated, its use is discouraged although not prohibited.

```
CALL ui.Interface.frontCall("WinMail","Init", [], [id] )
```
• ret is the identifier of the message initialized.
• For each Init function, a Close must be called.

Usage
This function initializes the module. It returns the identifier for the message, which will be used in other functions.

Close
The Close function clears all information corresponding to a message, and frees the memory occupied by the message.

Syntax
Important: This feature is deprecated, its use is discouraged although not prohibited.

```plaintext
CALL ui.Interface.frontCall("WinMail","Close", [id], [ result ])
```

• id is the message identifier.
• result is the status of the function.

SetBody
The SetBody function sets the body of the mail.

Syntax
Important: This feature is deprecated, its use is discouraged although not prohibited.

```plaintext
CALL ui.Interface.frontCall("WinMail","SetBody", [ id, body ], [ result ])
```

• id is the message identifier.
• body is the string text containing the body of the mail.
• result is the status of the function.

SetSubject
The SetSubject function sets the subject of the mail.

Syntax
Important: This feature is deprecated, its use is discouraged although not prohibited.

```plaintext
CALL ui.Interface.frontCall("WinMail","SetSubject", [ id, subject ], [ result ])
```

• id is the message identifier.
• subject is the string text containing the subject of the mail.
• result is the status of the function.

AddTo
The AddTo function adds a "To" addressee to the mail.

Syntax
Important: This feature is deprecated, its use is discouraged although not prohibited.

```plaintext
CALL ui.Interface.frontCall("WinMail","AddTo", [ id, name, address ], [ result ])
```
•  *id* is the message identifier.
•  *name* is the name to be displayed in the mail.
•  *address* is the mail address to be used for this addressee.
•  *result* is the status of the function.

**Usage**

This function adds a "To" addressee to the mail. The addressee has a name and a mail address.

**AddCC**

The *AddCC* function adds a "CC" addressee to the mail.

**Syntax**

*Important:* This feature is deprecated, its use is discouraged although not prohibited.

```plaintext
CALL ui.Interface.frontCall("WinMail","AddCC", [ id, name, address ], [ result ])
```

•  *id* is the message identifier.
•  *name* is the name to be displayed in the mail.
•  *address* is the mail address to be used for this addressee.
•  *result* is the status of the function.

**Usage**

This function adds a "CC" addressee to the mail. The addressee has a name and a mail address.

**AddBCC**

The *AddBCC* function adds a "BCC" addressee to the mail.

**Syntax**

*Important:* This feature is deprecated, its use is discouraged although not prohibited.

```plaintext
CALL ui.Interface.frontCall("WinMail","AddBCC", [ id, name, address ], [ result ])
```

•  *id* is the message identifier.
•  *name* is the name to be displayed in the mail.
•  *address* is the mail address to be used for this addressee.
•  *result* is the status of the function.

**Usage**

This function adds a "BCC" addressee to the mail. The addressee has a name and a mail address.

**AddAttachment**

The *AddAttachment* function adds a file as an attachment to the mail.

**Syntax**

*Important:* This feature is deprecated, its use is discouraged although not prohibited.

```plaintext
CALL ui.Interface.frontCall("WinMail","AddAttachment", [ id, fileName], [ result ])
```

•  *id* is the message identifier.
• `fileName` is the path of the attachment; the path can be relative to the directory from which GDC is run, or absolute.
• `result` is the status of the function.

Usage

This function adds a file as an attachment to the mail. The file must be located on the front-end.

`SendMailSMTP`

The `SendMailSMTP` function sends the mail with the SMTP protocol.

Syntax

Important: This feature is deprecated, its use is discouraged although not prohibited.

```plaintext
CALL ui.Interface.frontCall("WinMail","SendMailSMTP",
                           [ id ], [ result ] )
```

• `id` is the message identifier.
• `result` is TRUE in case of success; use `GetError` to get the description of the error when needed.

Usage

This function sends the mail by using the SMTP protocol. Default mailer software is called to create the mail. The user must press the "send" button to send the mail.

`SendMailMAPI`

The `SendMailMAPI` function sends the mail with the MAPI protocol.

Syntax

Important: This feature is deprecated, its use is discouraged although not prohibited.

```plaintext
CALL ui.Interface.frontCall("WinMail","SendMailMAPI",
                           [ id ], [ result ] )
```

• `id` is the message identifier.
• `result` is TRUE in case of success; use `GetError` to get the description of the error when needed.

Important:

• MAPI needs to log in to the mailer software. The first login can take time, depending on the mailer software. Your Genero application will be blocked until MAPI returns.
• MAPI depends on the mailer software for error management. For instance, Mozilla Thunderbird returns "success" when the mail is created, but Outlook 2002 only returns "success" when the mail is sent.

Usage

This function sends the mail by using the MAPI protocol. With MAPI, the default mailer software is called to create the mail. The user must press the "send" button to send the mail.

`GetError`

The `GetError` function gets a description of the last error that occurred.

Syntax

Important: This feature is deprecated, its use is discouraged although not prohibited.

```plaintext
CALL ui.Interface.frontCall("WinMail","GetError",
                           [ id ], [ result ] )
```
• *id* is the message identifier.
• *result* is the description of the last error.
• the returned value is NULL if there was no error.

*SetSmtp*

The *SetSmtp* function sets the SMTP server to be used.

**Syntax**

**Important:** This feature is deprecated, its use is discouraged although not prohibited.

```plaintext
CALL ui.Interface.frontCall("WinMail", "SetSmtp", 
[ id, smtp:port ], [ result ] )
```

• *id* is the message identifier.
• *smtp* is the string text containing the SMTP server to be used.
• *port* is optional. It allows to specify a port for your SMTP server. When not specified, the default port remains 25.
• *result* is the status of the function.

*SetFrom*

The *SetFrom* function sets sender information.

**Syntax**

**Important:** This feature is deprecated, its use is discouraged although not prohibited.

```plaintext
CALL ui.Interface.frontCall("WinMail", "SetFrom", 
[ id, name, address ], [ result ] )
```

• *id* is the message identifier.
• *name* is the name to be displayed in the mail.
• *address* is the mail address to be used for this addressee.
• *result* is the status of the function.

**WinMail examples**

Various WinMail examples.

**Mail using MAPI**

```plaintext
MAIN
  DEFINE result, id INTEGER
  DEFINE str STRING
  -- first, we initialize the module
  CALL ui.Interface.frontCall("WinMail", "Init", [], [id])

  -- Set the body of the mail
  CALL ui.interface.frontCall("WinMail", "SetBody", 
[ id, "This is a text mail using WinMail API - MAPI"], [result])

  -- Set the subject of the mail
  CALL ui.interface.frontCall("WinMail", "SetSubject", 
[ id, "test mail - ignore it"], [result])

  -- Add an Addressee as "TO"
  CALL ui.Interface.frontCall("WinMail", "AddTo", 
[ id, "myBoss", "boss@mycompany.com"], [result])

  -- Add another Addressee as "BCC"
  CALL ui.Interface.frontCall("WinMail", "AddBCC", 
[ id, "my friend", "friend@mycompany.com"], [result])
```
Mail using SMTP server

This topic provides an example of sending mail using an SMTP server.

MAIN
DEFINE result, id INTEGER
DEFINE str STRING

-- first, we initialize the module
CALL ui.Interface.frontCall("WinMail", "Init", [], [id])

-- Set the body of the mail
CALL ui.Interface.frontCall("WinMail", "SetBody", [id, "This is a text mail using WinMail API - MAPI"], [result])

-- Set the subject of the mail
CALL ui.Interface.frontCall("WinMail", "SetSubject", [id, "test mail - ignore it"], [result])

-- Set the mail sender
CALL ui.Interface.frontCall("WinMail", "SetFrom", [id, "mySelf", "me@mycompany.com"], [result])

-- Set the SMTP server
CALL ui.Interface.frontCall("WinMail", "SetSmtp", [id, "smtp.mycompany.com"], [result])

-- Add an Addressee as "TO"
CALL ui.Interface.frontCall("WinMail", "AddTo", [id, "myBoss", "boss@mycompany.com"], [result])

-- Add another Addressee as "BCC"
CALL ui.Interface.frontCall("WinMail", "AddBCC", [id, "my friend", "friend@mycompany.com"], [result])

-- Add Two attachments
CALL ui.Interface.frontCall("WinMail", "AddAttachment", [id, "c:\mydocs\report.doc"], [result])
CALL ui.Interface.frontCall("WinMail", "AddAttachment", [id, "c:\mydocs\demo.png"], [result])

-- Send the mail via smtp
CALL ui.Interface.frontCall("WinMail", "SendMailSMTP", [id], [result])
IF result == TRUE THEN
    DISPLAY "Message sent successfully"
ELSE
    CALL ui.Interface.frontCall("WinMail", "GetError", [id], [str])
    DISPLAY str
END IF
CALL ui.Interface.frontCall("WinMail", "Close", [id], [result])
END MAIN

Theme front calls
This section describes Genero Browser Client (GBC)-specific front calls.

This table shows the functions implemented by the GBC in the "theme" module. These front calls were made available starting with GBC 1.00.47

Table 579: theme module front-end functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ui.Interface.frontCall(&quot;theme&quot;, &quot;setTheme&quot;, [name], [])</td>
<td>Activates a specific GBC theme.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;theme&quot;, &quot;getCurrentTheme&quot;, [], [result])</td>
<td>Gets the active GBC theme.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;theme&quot;, &quot;listThemes&quot;, [], [result])</td>
<td>Lists all available GBC themes.</td>
</tr>
</tbody>
</table>

Related concepts
Front calls on page 583
Front call functions execute on the platform where the front-end is installed.

setTheme
Activates a specific GBC theme.

Syntax

ui.Interface.frontCall("theme", "setTheme", [name], [])

- name - The theme to be activated.

Usage
The setTheme function allows the application to specify the theme to use for the application.

Example

main.4gl

IMPORT util
MAIN

DEFINE themes DYNAMIC ARRAY OF RECORD
    name STRING,
title STRING
-- conditions DYNAMIC ARRAY OF STRING
END RECORD

DEFINE result string

OPEN WINDOW w WITH FORM "myform"

CALL ui.Interface.frontcall("theme", "setTheme", ["highcontrast"], [])
DISPLAY "highcontrast" to formonly.setTheme

CALL ui.Interface.frontcall("theme", "getCurrentTheme", [], [result])
DISPLAY result to formonly.getCurrentTheme

CALL ui.Interface.frontcall("theme", "listThemes", [], [result])
DISPLAY result to formonly.listThemes
CALL util.JSON.parse(result, themes)
DISPLAY ARRAY themes TO themes.*

MENU
  ON ACTION CANCEL
    EXIT MENU
  ON ACTION CLOSE
    EXIT MENU
END MENU
END MAIN

myform.per

LAYOUT
GRID
{
  setTheme       [setTheme       ]
  getCurrentTheme [getCurrentTheme ]
  listThemes     [listThemes      ]

  <T t
    Name       Title
    [name       |title          ]
  <
}
END
END

ATTRIBUTES
EDIT setTheme = formonly.setTheme;
EDIT getCurrentTheme = formonly.getCurrentTheme;
EDIT listThemes = formonly.listThemes;
name = FORMONLY.name;
title = FORMONLY.title;

INSTRUCTIONS
SCREEN RECORD themes(name, title)

gGetCurrentTheme
Gets the active GBC theme.

Syntax

ui.Interface.frontCall("theme", "getCurrentTheme", [], [result])
- **result** - The name of the active theme.

**Usage**

The `getCurrentTheme` function allows the application to ask for and receive the theme that, by default, will be used when displaying the application.

**Example**

```4gl
main.4gl

IMPORT util

MAIN

DEFINE themes DYNAMIC ARRAY OF RECORD
    name STRING,
    title STRING
   -- conditions DYNAMIC ARRAY OF STRING
END RECORD

DEFINE result string

OPEN WINDOW w WITH FORM "myform"

CALL ui.Interface.frontcall("theme", "setTheme", ["highcontrast"], [])
DISPLAY "highcontrast" to formonly.setTheme

CALL ui.Interface.frontcall("theme", "getCurrentTheme", [], [result])
DISPLAY result to formonly.getCurrentTheme

CALL ui.Interface.frontcall("theme", "listThemes", [], [result])
DISPLAY result to formonly.listThemes
CALL util.JSON.parse(result, themes)
DISPLAY ARRAY themes TO themes.*

MENU
    ON ACTION CANCEL
        EXIT MENU
    ON ACTION CLOSE
        EXIT MENU
END MENU
END MAIN

myform.per

LAYOUT
GRID
{
setTheme                         [setTheme                  ]
getCurrentTheme                 [getCurrentTheme           ]
listThemes                      [listThemes                ]

<T t                                               >
    Name           Title
    [name          |title                              ]
    <
} END
END

ATTRIBUTES
EDIT setTheme = formonly.setTheme;
```
EDIT getCurrentTheme = formonly.getCurrentTheme;
EDIT listThemes = formonly.listThemes;
name = FORMONLY.name;
title = FORMONLY.title;

INSTRUCTIONS
SCREEN RECORD themes(name, title)

listThemes
Lists all available GBC themes.

Syntax

ui.Interface.frontCall("theme", "listThemes",
    [], [result])

• result - A string containing a JSON array of the available themes. For each theme, the name, title, and conditions are listed. The conditions are contained within a second JSON array.

Usage
The listThemes function returns a JSON array of the themes available to the running application. The array consists of the name, title and conditions of each available theme. For example, the function returns the following when run against the out-of-the-box runtime installation on a desktop machine (line breaks added for readability):

```
[{
    "name": "default",
    "title": "Default",
    "conditions": ["isDesktop"],
},
{
    "name": "highcontrast",
    "title": "High contrast",
    "conditions": ["isDesktop"]
}]
```

Example

main.4gl

IMPORT util

MAIN

DEFINE themes DYNAMIC ARRAY OF RECORD
    name STRING,
    title STRING
    -- conditions DYNAMIC ARRAY OF STRING
END RECORD

DEFINE result string

OPEN WINDOW w WITH FORM "myform"

CALL ui.Interface.frontcall("theme", "setTheme", ["highcontrast"], [])
DISPLAY "highcontrast" to formonly.setTheme

CALL ui.Interface.frontcall("theme", "getCurrentTheme", [], [result])
DISPLAY result to formonly.getCurrentTheme

CALL ui.Interface.frontcall("theme", "listThemes", [], [result])
DISPLAY result to formonly.listThemes
CALL util.JSON.parse(result, themes)
DISPLAY ARRAY themes TO themes.*

MENU
    ON ACTION CANCEL
        EXIT MENU
    ON ACTION CLOSE
Local storage front calls

This section describes front calls to store data on the front-end platform.

Key/Value pairs can be stored locally on the front-end side with the `localStorage` front calls.

Local storage is supported by all front-ends.

The data is stored on the platform where the front-end executes, and is persistent across application sessions.

This feature can for example be used to keep a trace of the authenticated users, if the authentication mechanism is written in Genero.

The table shows the functions implemented by all front-ends in the "localStorage" module.
### Table 580: localStorage module front-end functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ui.Interface.frontCall(&quot;localStorage&quot;, &quot;clear&quot;, [[]], [[]])</code></td>
<td>Removes all local storage key/value pairs.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;localStorage&quot;, &quot;getItem&quot;, [key], [value])</code></td>
<td>Returns the current value of local storage key.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;localStorage&quot;, &quot;keys&quot;, [[]], [key-list] )</code></td>
<td>Returns the list of defined local storage keys.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;localStorage&quot;, &quot;removeItem&quot;, [key], [[]])</code></td>
<td>Deletes the specified local storage key.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;localStorage&quot;, &quot;setItem&quot;, [key, value], [[]])</code></td>
<td>Sets a value for local storage key.</td>
</tr>
</tbody>
</table>

### setItem

Sets a value for local storage key.

**Syntax**

```javascript
ui.Interface.frontCall("localStorage", "setItem", [key, value], [[]])
```

- `key` is the name of the local storage key.
- `value` is the value to set for the named key.

**Usage**

The `setItem` function sets the specified local storage key with the value passed as parameter.

### getItem

Returns the current value of local storage key.

**Syntax**

```javascript
ui.Interface.frontCall("localStorage", "getItem", [key], [value])
```

- `key` is the name of the local storage key.
- `value` is the current value of the named key.

**Usage**

The `getItem` function returns the value of the specified local storage key.
**removeItem**  
Deletes the specified local storage key.

**Syntax**

```javascript
ui.Interface.frontCall("localStorage", "removeItem", [key], [])
```

- `key` is the name of the local storage key.

**Usage**

The `removeItem` function deletes the specified local storage key.

**keys**  
Returns the list of defined local storage keys.

**Syntax**

```javascript
ui.Interface.frontCall("localStorage", "keys", [], [key-list])
```

- `key-list` list of key names as a JSON array of strings.

**Usage**

The `keys` function returns the current local storage keys defined on the front-end platform. The list of keys is returned in a string variable, as a JSON array of strings.

Convert the JSON array to a BDL dynamic array with `util.JSON.parse` on page 2768:

```javascript
IMPORT util  
...
DEFINE key_list STRING,  
    key_array DYNAMIC ARRAY OF STRINGS
CALL ui.Interface.frontCall("localStorage", "keys", [], [key_list])
CALL util.JSON.parse( key_list, key_array )
```

clear  
Removes all local storage key/value pairs.

**Syntax**

```javascript
ui.Interface.frontCall("localStorage", "clear", [], [])
```

**Usage**

The `clear` function removes all local storage keys currently saved on the front-end side.

**Genero Mobile common front calls**  
This section describes common front calls provided by all mobile front-ends.

This table shows the functions implemented by all mobile front-ends in the "mobile" module.
Table 581: Common mobile module front-end functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ui.Interface.frontCall(&quot;mobile&quot;, &quot;chooseContact&quot;, [], [result])</td>
<td>Lets the user choose a contact from the mobile device contact list and returns the vCard.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;mobile&quot;, &quot;choosePhoto&quot;, [], [path])</td>
<td>Lets the user select a picture from the mobile device's photo gallery and returns a picture identifier.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;mobile&quot;, &quot;chooseVideo&quot;, [], [path])</td>
<td>Lets the user select a video from the mobile device's video gallery and returns a video identifier.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;mobile&quot;, &quot;composeMail&quot;, [to, subject, content, cc, [result]])</td>
<td>Invokes the user's default mail application for a new mail to send.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;mobile&quot;, &quot;composeSMS&quot;, [recipients, content], [result])</td>
<td>Sends an SMS text to one or more phone numbers.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;mobile&quot;, &quot;connectivity&quot;, [], [result])</td>
<td>Returns the type of network available for the mobile device.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;mobile&quot;, &quot;getGeolocation&quot;, [], [status, latitude, longitude])</td>
<td>Returns the Global Positioning System (GPS) location of a mobile device.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;mobile&quot;, &quot;getRemoteNotifications&quot;, [], [data])</td>
<td>This front call retrieves push notification messages.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;mobile&quot;, &quot;importContact&quot;, [vcard], [result])</td>
<td>Creates a new contact, or merges to an existing entry, the contact details passed in a vCard string.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;mobile&quot;, &quot;isForeground&quot;, [], [result])</td>
<td>Indicates if the mobile app is in foreground mode.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;mobile&quot;, &quot;newContact&quot;, [defaults], [vcard])</td>
<td>Lets the user input contact information to create a new entry in the contact database of the mobile device.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;mobile&quot;, &quot;registerForRemoteNotifications&quot;, [], [registration-token])</td>
<td>This front call registers a mobile device for push notifications.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;mobile&quot;, &quot;runOnServer&quot;, [appurl, timeout], [])</td>
<td>Run an application from the Genero Application Server using the specified URL.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;mobile&quot;, &quot;scanBarCode&quot;, [], [code, type])</td>
<td>Allow the user to scan a barcode with a mobile device</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;mobile&quot;, &quot;takePhoto&quot;, [], [path])</td>
<td>Lets the user take a picture with the mobile device and returns the corresponding picture identifier.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;mobile&quot;, &quot;takeVideo&quot;, [], [path])</td>
<td>Lets the user take a video with the mobile device and returns the corresponding video identifier.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;mobile&quot;, &quot;unregisterFromRemoteNotifications&quot;, [], [])</td>
<td>This front call unregisters the mobile device from push notifications.</td>
</tr>
</tbody>
</table>
**Related concepts**

**Genero Mobile Android front calls** on page 2713  
This section describes front calls specific to the Android™ platform.

**Genero Mobile iOS front calls** on page 2718  
This section describes front calls specific to the iOS platform.

**chooseContact**

Lets the user choose a contact from the mobile device contact list and returns the vCard.

**Syntax**

```plaintext
ui.Interface.frontCall("mobile", "chooseContact", [], [result])
```

- `result` - The vCard string from the device's contacts database.

**Usage**

The "chooseContact" front call opens the mobile device contact chooser, allows the user to select a contact and returns the contact as a vCard string.

**Important:** For GMA / Android™, using the chooseContact front call needs the `android.permission.READ_CONTACTS` Dangerous Permission to be specified when building the APK. See [Android permissions](#) on page 3595 for more details. On Android™ 5.1 and lower (< API 23), use the `android.permission.GET_ACCOUNTS` permission.

If the user cancels the contact chooser, NULL is returned.

**Example**

```plaintext
DEFINE vcard STRING
CALL ui.Interface.frontCall("mobile", "chooseContact", [], [vcard])
```

**Related concepts**

**importContact** on page 2706  
Creates a new contact, or merges to an existing entry, the contact details passed in a vCard string.

**newContact** on page 2708  
Lets the user input contact information to create a new entry in the contact database of the mobile device.

**choosePhoto**

Lets the user select a picture from the mobile device's photo gallery and returns a picture identifier.

**Syntax**

```plaintext
ui.Interface.frontCall("mobile", "choosePhoto", [], [path])
```

1. `path` - Holds the device opaque path to the chosen photo.

**Usage**

The "choosePhoto" front call starts the system's photo chooser (the device's photo gallery), allows the user to choose a photo, and returns the path/URL on the mobile device of the chosen photo.

If the user cancels the photo chooser, NULL is returned.
**Important:** For GMA / Android™, using the `choosePhoto` front call needs the `android.permission.READ_EXTERNAL_STORAGE` Dangerous Permission to be specified when building the APK. See Android permissions on page 3595 for more details.

The value returned in the `path` variable contains a reference to the system location of the picture on the mobile device. This path is platform dependent, and may change in future versions. Consider the path returned by this front call as an opaque local file identifier, and do not use this path as a persistent file name for the picture.

For more details about mobile image handling, see images handling on mobile devices.

**Related concepts**

- **Using images** on page 1147
  Describes how to use pictures in the forms of your application.

- **fgl_getfile()** on page 2286
  Retrieves a file from the front-end context to the virtual machine context.

- **takePhoto** on page 2712
  Lets the user take a picture with the mobile device and returns the corresponding picture identifier.

- **chooseVideo**
  Lets the user select a video from the mobile device's video gallery and returns a video identifier.

**Syntax**

```plaintext
ui.Interface.frontCall("mobile", "chooseVideo",
 [], [path])
```

1. `path` - Holds the device opaque path to the selected video.

**Usage**

The "chooseVideo" front call starts the system's video chooser (the device's video gallery), allows the user to choose a video, and returns the path/URL on the mobile device of the selected video.

If the user cancels the video chooser, `NULL` is returned.

**Important:** For GMA / Android™, using the `chooseVideo` front call needs the `android.permission.READ_EXTERNAL_STORAGE` Dangerous Permission to be specified when building the APK. See Android permissions on page 3595 for more details.

The value returned in the `path` variable contains a reference to the system location of the video on the mobile device. This path is platform dependent, and may change in future versions. Consider the path returned by this front call as an opaque local file identifier, and do not use this path as a persistent file name for the video.

Once the video identifier/path is known, it is possible to fetch the video file from the device to the program context with the `fgl_getfile()` API. The procedure is similar to fetching photos from the device. For more details, see the section about video handling on mobile devices.

To play the video, you can perform a "launchURL" front call, with the opaque path returned by this front call.

**Related concepts**

- **Using images** on page 1147
  Describes how to use pictures in the forms of your application.

- **fgl_getfile()** on page 2286
  Retrieves a file from the front-end context to the virtual machine context.

- **takeVideo** on page 2712
Lets the user take a video with the mobile device and returns the corresponding video identifier.

**composeMail**
Invokes the user's default mail application for a new mail to send.

**Syntax**
```
ui.Interface.frontCall("mobile", "composeMail",
    [to, subject, content, cc, bcc, attachments ...],
    [result])
```

- **to** - A list of recipients, separated by commas. While the list uses commas to separate the recipients in the list, the list itself is enclosed in a single set of quotes.
- **subject** - The subject of the email.
- **content** - The body of the email.
- **cc** - (optional) A list of recipients for the carbon-copy email field, separated by commas. While the list uses commas to separate the recipients in the list, the list itself is enclosed in a single set of quotes.
- **bcc** - (optional) A list of recipients for the blind carbon-copy email field, separated by commas. While the list uses commas to separate the recipients in the list, the list itself is enclosed in a single set of quotes.
- **attachments** - (optional) All remaining arguments are treated as paths to attachment files. Each attachment file name is enclosed in its own set of quotes. The comma is used to separate the attachments in the list.
- **result** - Holds a status message.

**Usage**
The "composeMail" front call invokes the user's default mail application and sets up a new mail to send.

The returned result string takes one of the following values:

- "ok": The email was sent.
- "cancel": The email was canceled.
- "saved": The email was saved.
- "failed: reason": The email was not sent.

This example opens an email and populates the To, CC, and BCC fields, the Subject line, the message body, and it specifies two attachments.

```
DEFINE result STRING
CALL ui.Interface.frontCall("mobile", "composeMail",
    ["john.doe@4js.com,jane.doe@4js.com", "Hello world",
     "This is the hello world text", "john.doe@4js.com,jane.doe@4js.com",
     "hidden@4js.com",
     "/sdcard/Pictures/photo1.jpg", "/sdcard/Pictures/photo2.jpg " ], [result])
```

The next example opens an email and populates the To field, the Subject line, and the message body. No CC or BCC recipients and no attachments are specified.

```
DEFINE result STRING
CALL ui.Interface.frontCall("mobile","composeMail",
    ["huhu@haha.com","test mail","sent from my device"],[result])
```

**composeSMS**
Sends an SMS text to one or more phone numbers.

**Syntax**
```
ui.Interface.frontCall("mobile", "composeSMS",
    [ recipients, content ],
```
Library reference

2702

[ result ]

- recipients - A list of phone numbers, separated by commas. While the list uses commas to separate the phone numbers in the list, the list itself is enclosed in a single set of quotes.
- content - The SMS message.
- result - Holds a status message.

Usage

The "composeSMS" front call sends an SMS text to one or more phone numbers.

Consider using global phone numbers with a + plus sign, as described in [RFC3966].

The returned result string can take one of the following values:

- "ok": The SMS was send.
- "cancel": The SMS was canceled.
- "failed": The SMS could not be sent.

Error -6333 is raised, if there is no permission to compose an SMS on the mobile phone.

Example

```c
DEFINE result STRING
CALL ui.Interface.frontCall("mobile", "composeSMS",
    [+332781211,+339956789", "This is the SMS text"],
    [result])
```

connectivity

Returns the type of network available for the mobile device.

Syntax

```c
ui.Interface.frontCall("mobile", "connectivity",
    [], [result] )
```

- result - Holds the type of network available.

Usage

The "connectivity" front call checks for the best available mobile network connectivity to the internet.

The returned result string can take one of the following values:

- "NONE": No connectivity is available to the internet or the specified host.
- "MobileNetwork": Connectivity is available via the mobile network (Edge, 3G, 4G).
- "WIFI": Connectivity is available via a WIFI connection.

Example

```c
DEFINE network STRING
CALL ui.Interface.frontCall("mobile", "connectivity", [], [network] )
IF network == "WIFI" THEN
    ...
END IF
```

Related concepts

Front calls on page 583
Front call functions execute on the platform where the front-end is installed.

**getGeolocation**
Returns the Global Positioning System (GPS) location of a mobile device.

**Syntax**

```java
ui.Interface.frontCall("mobile", "getGeolocation", [], [status, latitude, longitude] )
```

1. **status** - Holds the status of the front call execution.
2. **latitude** - Holds the current latitude.
3. **longitude** - Holds the current longitude.

**Usage**
The "getGeolocation" front call returns the current location of the mobile device, based on the current GPS information.

**Important:** For GMA / Android™, using the getGeolocation front call needs the android.permission.ACCESS_FINE_LOCATION and android.permission.ACCESS_COARSE_LOCATION Dangerous Permissions to be specified when building the APK. See Android permissions on page 3595 for more details.

The possible values returned in the **status** parameter are:

- "ok": The mobile device location was found.
- In case of failure, the status variable contains the error description, for example, "location services not enabled".

It is recommended that the returned coordinates are stored in FLOAT variables.

If the device's location cannot be found within a given period, the front call returns an error status.

**Example**

```java
DEFINE status STRING, latitude, longitude FLOAT
CALL ui.Interface.frontCall("mobile", "getGeolocation", [], [status, latitude, longitude] )
MESSAGE SFMT( "Geo location: (status=%1) Latitude=%2 Longitude=%3", status, latitude, longitude )
```

**getRemoteNotifications**
This front call retrieves push notification messages.

**Syntax**

```java
ui.Interface.frontCall("mobile", "getRemoteNotifications", [], [data] )
```

1. **data** - STRING containing a JSON array of notifications.

**Usage**
After registering for push notifications with the registerForRemoteNotifications on page 2708 front call, the getRemoteNotifications front call can be called in the context of an ON ACTION notificationpushed action handler.
The GMI or GMA front-end will send the notification pushed special action, when it receives notifications from the push notification server. When this action is fired, use the getRemoteNotifications front call to get notification data.

Important:

When an app restarts, if notifications are pending and the app has already registered for push notification in a previous execution, the notification pushed action will be raised as soon as a dialog with the corresponding ON ACTION handler activates. The app then performs a getRemoteNotifications on page 2703 front call as in the usual way, to get the pending notifications pushed to the device while the app was off.

However, special consideration needs to be given to iOS devices. When push notification arrives for an iOS app that has not started, there is no mechanism to wake up the app and get the push data. Therefore, when the user starts the app from the springboard, there will never be any push data available. Depending on the context, implement the following programming patterns to solve this problem:

1. If the push notification contains a badge number, the app can verify if the badge is greater than 0 (with the getBadgeNumber front call) in order to perform a getRemoteNotifications front call. Even if there is no data available with the front call, it is recommended that the app sends a request directly to the server push provider to get last push data.

2. If the push notification does not contain badge numbers, it is still recommended that the app performs a getRemoteNotification front call when it starts. If there is no push data available from the front call, the recommendation is that the app sends a request to the server push provider to see if there is push data available. This is by the way also recommended when receiving a notification pushed action during application life time.

3. If the user starts the app from the Notification Center, the app is launched with push data transmitted from the system, and the notification pushed action is sent. It is recommended that the app perform a getRemoteNotifications front call and get the push data.

The "getRemoteNotifications" front call returns a list of notification records as a JSON array string. Use the util.JSONArray or util.JSON class to extract notification data from the returned string. The structure of a push notification is platform specific. See below for details.

Important: When an iOS app is in background, silent push notifications can occur, but notification message data (i.e. the payload) may not be available. In such case, GMI is able to detect that a notification arrived (i.e. when the app badge number is greater than zero) and raise the notification pushed action, but the getRemoteNotifications front call will return no message data (data return param is NULL). If such case, implement a fallback mechanism (based on RESTful web services for example), to contact the push notification provider and retrieve the message information.

Push notification records with GMA / Android™

The returned JSON string from a FCM notification server contains an array of notification records.

A notification record contains the following JSON keys:

- "type" - can be "message" or "token".
- "data" - Contains notification data.
  - When "type": "message", the notification record is a FCM application message, and the data attribute contains custom notification information.

An element of "data" can be a "genero_notification" record, that will produce an Android™ graphical notification. This record must define the following attributes:

- "title" - title of the graphical notification
- "content" - text content of the graphical notification
- "icon" - icon of the graphical notification

The "genero_notification" record can be followed by custom notification data.

- When "type": "token", the notification record is a registration token update, and the "data" attribute contains the new registration token, that is required to be re-sent to the push notification server.
• "from" - Contains the FCM project id.

JSON push notification data example for GMA:

```
[
    {
        "type": "message",
        "data": { custom-attributes ... },
        "from": "project-id"
    },
    {
        "type": "token",
        "data": "new-registration-token",
        "from": "project-id"
    },
    ...
]
```

Note that the JSON push notification data can contain a "data" attribute with a "genero_notification" record, that will produce an Android™ graphical notification:

```
[
    {
        "type": "message",
        "data": {
            "genero_notification": {
                "title": "Game Request!",
                "content": "Bob wants to play poker...",
                "icon": "smiley"
            },
            custom-attributes
            ...
        },
        "from": "project-id"
    },
    ...
]
```

**Push notification records with GMI / iOS**

The returned JSON string from an Apple® Push Notification contains an array of notification records.

A push notification record contains the following JSON attributes:

- "aps" (required) - key to be recognized by devices as an Apple® Push Notification
  - "alert" (required) - key of the push notification content. If not specified as a single value, the alert key can hold:
    - "title" - title of the alert.
    - "body" - the message to be displayed.
  - "badge" (optional) - the number to display as the badge of the app icon. If this property is absent, the badge is not changed. You need to manage it through your push notification provider.
  - "sound" (optional) - the sound played by the alert (aiff, wav, or caf format) default value: "default". To use a custom file you will need to use the GMI extension project and be familiar with Objective-C. The file must bundled with the app.
  - "content-available" (required) - The content-available property with a value of 1 allows the remote notification to act as a “silent” notification. The recommendation is that notifications received in background mode are stored for delivery when the app enters foreground mode.
JSON push notification data example for GMI:

```
[
  {
    "aps": {
      "alert": "My first push",
      "badge": 1,
      "sound": "default",
      "content-available": 1
    }
  },
  {
    "aps": {
      "alert": {
        "title": "Push",
        "body": "My second push"
      },
      "badge": 2,
      "sound": "default",
      "content-available": 1
    },
    "new_ids": [ "XV234", "ZF452", "RT563" ],
    "updated_ids": [ "AC634", "HJ153" ]
  }
]
```

In the last record, custom information is provided in the "new_ids" and "updated_ids" attributes, as a JSON array of identifiers.

For more details, see Apple Push Notification Service.

**Example**

```less
IMPORT util -- JSON API
DEFINE notif_list STRING
DIALOG ...
  ...
  ON ACTION notificationpushed
    CALL ui.Interface.frontCall(
      "mobile", "getRemoteNotifications",
      [ ], [ notif_list ] )
    -- Analyse content of notiflist
    DISPLAY util.JSON.format(notif_list)
  ...
```

**Related concepts**

- [registerForRemoteNotifications](#) on page 2708
  This front call registers a mobile device for push notifications.
- importContact
  Creates a new contact, or merges to an existing entry, the contact details passed in a vCard string.

**Syntax**

```less
ui.Interface.frontCall("mobile", "importContact",
```
1. **vcard** - Holds a vCard string to be imported into the device's contacts database.
2. **result** - Holds the completed vCard string.

**Usage**

The "importContact" front call sends the vCard definition passed as parameter to the mobile device.

If the contact import is canceled, the front-end returns NULL. Otherwise, it returns the vCard data.

On iOS devices, the user has the choice to create a new contact, or complete an existing contact entry. When creating a new entry, the contact input form is opened on the mobile device, to let the user complete the default values passed as parameter. When merging contact information to an existing entry, the user selects an entry from the contact list. If the contact import is validated, the front call returns the completed vCard string.

On Android™ devices, this front call creates a new contact entry directly in the mobile contact list, depending on the VCard definition passed as parameter, no intermediate input form is presented to the end user. If the contact import is validated, the front call returns the original vCard string passed as parameter.

**Important:** For GMA / Android™, using the importContact front call needs the android.permission.WRITE_EXTERNAL_STORAGE Dangerous Permission to be specified when building the APK. See Android permissions on page 3595 for more details.

**Example**

```velvet
DEFINE vcard, result STRING
LET vcard="BEGIN:VCARD\n"||"VERSION:3.0\n"||"N:Willi;;;;\n"||"TEL;type=HOME;type=VOICE;type=pref:03812225610\n"
||"END:VCARD\n"
CALL ui.interface.frontcall("mobile", "importContact", [vcard], [result])
```

**Related concepts**

chooseContact on page 2699

Lets the user choose a contact from the mobile device contact list and returns the vCard.

newContact on page 2708

Lets the user input contact information to create a new entry in the contact database of the mobile device.

isForeground

Indicates if the mobile app is in foreground mode.

**Syntax**

```velvet
ui.Interface.frontCall("mobile", "isForeground",
[], [result] )
```

- **result** - Is set to TRUE if the app is in foreground mode, or FALSE if in background mode.

**Usage**

The "isForeground" front call checks if the mobile app is currently in foreground or background mode.

```velvet
DEFINE fg BOOLEAN
CALL ui.Interface.frontCall("mobile", "isForeground", [], [fg] )
IF fg THEN
    ...
END IF
```
Use the isForeground front call in conjunction with the **enterforeground** and **enterbackground** predefined actions.

### newContact

Lets the user input contact information to create a new entry in the contact database of the mobile device.

**Syntax**

```
ui.Interface.frontCall("mobile", "newContact", 
[defaults], [vcard])
```

- `defaults` - A vCard string with default values for the new contact input.
- `vcard` - Holds the vCard string of the new created contact.

**Usage**

The "newContact" front call opens the contact input form on the mobile device, with default values passed in the vCard structure of the first parameter, allows the user to enter contact information.

**Important:** This front call is only available for an application running on an iOS device.

If the contact creation is validated, the front call returns the completed vCard string. If the contact import is canceled, the front-end returns NULL.

**Example**

```
DEFINE defaults, vcard STRING
LET defaults="BEGIN:VCARD\n" "VERSION:3.0\n" "N:Willi;;;;\n" "TEL;type=HOME;type=VOICE;type=pref:03812225610\n" "END:VCARD\n"
CALL ui.interface.frontcall("mobile","newContact",[defaults],[vcard])
```

**Related concepts**

- chooseContact on page 2699
  Lets the user choose a contact from the mobile device contact list and returns the vCard.

- importContact on page 2706
  Creates a new contact, or merges to an existing entry, the contact details passed in a vCard string.

- registerForRemoteNotifications
  This front call registers a mobile device for push notifications.

**Syntax**

```
ui.Interface.frontCall("mobile","registerForRemoteNotifications", 
[], [registration-token] )
```

1. `registration-token` - Registration token to be sent to the push notification provider. For GMA/Android, this is the "registration token" obtained from Firebase Cloud Messaging (FCM), for GMI/iOS, this is the "device token" obtained from Apple Push Notification services (APNs).

**Usage**

The "registerForRemoteNotifications" front call registers the mobile device for push notifications. Once the registration procedure is done (see below for platform specifics), it is possible to get notification events through the **notificationpushed** predefined action, and retrieve notification data with the `getRemoteNotifications` on page 2703 front call.
**Note:** The app does not need to register for notification each time it is restarted. Even if the app is closed, the registration is still active until the `unregisterFromRemoteNotifications` front call is performed. At first execution, an app will typically ask if the user wants to get push notifications and register to the push service if needed. To disable push notification, apps usually implement an option that can be disabled (to unregister) and re-enabled (to register again) by the user. On Android™, the app must register for notification each time it is upgraded.

**Registering with FCM on Android**

On Android™ the `registration-token` is the registration token returned by FCM. Once registered with the FCM service, the app must also send this registration token to the FCM application server. Registration tokens are typically sent to the FCM application server using a RESTful HTTP POST.

**Note:** Android™ apps using push notification services need specific permissions to be defined in the manifest, such as `android.permission.GET_ACCOUNTS`, `com.google.android.c2dm.permission.RECEIVE`, and especially `application-package-name.permission.C2D_MESSAGE`. These Android permissions will be automatically set by the `gmabuildtool`, depending on the package name specified with the `--build-app-package-name` option.

For more details, see FCM documentation and About FCM Connection Server.

**Registering with APNs on iOS**

On iOS when using APNs, the `registration-token` is the device token returned by the Apple Push Notification service. Once registered with the Apple Push Notification service, the app must also send this device token to the push notification provider, typically using a RESTful HTTP POST.

For more details about Apple Push Notification Provider, see the Apple Push Notification Service web site.

**Example**

The following code example registers with Firebase Cloud Messaging or Apple Push Notification service. It then sends the registration token to the push notification provider:

```groovy
IMPORT com  -- For RESTful post
IMPORT util -- JSON API
DEFINE registration_token STRING
DEFINE req com.HTTPRequest,
    obj util.JSONObject,
    resp com.HTTPResponse

-- First get the registration token
CALL ui.Interface.frontCall(
    "mobile", "registerForRemoteNotifications",
    [ ], [ registration_token ] )

-- Then send registration token to push notification provider
TRY
    LET req = com.HTTPRequest.create("http://SERVER_IP:4930")
    CALL req.setHeader("Content-Type", "application/json")
    CALL req.setMethod("POST")
    CALL req.setTimeOut(5)
    LET obj = util.JSONObject.create()
    CALL obj.put("registration_token", registration_token)
    CALL req.doTextRequest(obj.toString())
    LET resp = req.getResponse()
    IF resp.getStatusCode() != 200 THEN
        MESSAGE SFMT("HTTP Error (%1) %2",
            resp.getStatusCode(),
            resp.getStatusDescription())
    ELSE
        MESSAGE "Registration token sent."
    END IF
END TRY
```
CATCH
    MESSAGE SFMT("Could not post registration token to server: %1", STATUS)
END TRY

Related concepts

unregisterFromRemoteNotifications on page 2713
This front call unregisters the mobile device from push notifications.

The APNS class on page 2923
The com.APNS class implements Apple® Push Notification Service APIs.

runOnServer
Run an application from the Genero Application Server using the specified URL.

Syntax

ui.Interface.frontCall("mobile", "runOnServer",
    [ appurl, timeout ], [] )

- **appurl** - The GAS URL to the Genero application (this must be a ua/r URL).
- **timeout** - The timeout (in seconds) to wait for the remote application.

Usage

The runOnServer front call allows you to start an application in the Genero Application Server (GAS), from an embedded/local application running on the mobile device. The remote application's graphical user interface displays on the mobile device.

The front call returns when the called application ends, and the control goes back to the initial application executing on the mobile device.

The applications executed on the GAS server must use the UTF-8 encoding. Mobile front-ends will reject any attempt to display forms of an application using an encoding other than UTF-8.

The remote application cannot use RUN WITHOUT WAITING to start child programs. Only RUN is supported.

The first parameter (*appurl*) identifies the remote application to be started and must contain an "ua/r" URL syntax (the UA protocol introduced with the GAS 3.00).

For example: http://myappserver:6394/ua/r/myapp.

This URL may contain a query string, with parameters for the application to be executed by the GAS.

The *timeout* parameter is optional. It can be used to give the control back to the local app, if the remote app takes too long to respond. If not specified, or when zero is passed, the timeout is infinite.

In case of failure (such as application not found, or timeout expired), the front call raises the runtime error -6333 and the HTTP status code of the request can be found in the error message details.

**Note**: The application running on the GAS can only access the data-directory directory, in the sandbox of the embedded application that executes the runOnServer front call. File handling APIs like fgl_getfile() and fgl_putfile() can only access this directory on the mobile device. If no absolute path is specified in the file path for the mobile device, the data-directory is used.

Example

TRY
    CALL ui.interface.frontcall("mobile","runOnServer","[http://santana:6394/ua/r/orders"],[])
CATCH
    ERROR err_get(STATUS)
Related concepts
Running mobile apps on an application server on page 3616
From the mobile device, programs can be started remotely on an application server, and displayed on the device.

scanBarCode
Allow the user to scan a barcode with a mobile device

Syntax

ui.Interface.frontCall("mobile", "scanBarCode",
    [], [code, type] )

1. code - Holds a string representation of the barcode.
2. type - Holds the name of the barcode type.

Usage
The "scanBarCode" front call starts the barcode scanner to let the user scan a barcode with the device.
After reading the barcode, the front call returns the string representation of the barcode and the barcode type (symbology).
The code return parameter contains the barcode string.
The type return parameter indicates the type of barcode that was scanned.
If the barcode scan was canceled, the code return parameter is set to NULL and type is set to "canceled".

• On iOS devices, the barcode reader used by GMI is "ZBar". For more details, see ZBar bar code reader.
• On Android™ devices, the barcode reader used by GMA is "zxing". The zxing barcode reader must be installed as a separate app. For more details, see Barcode Scanner.

Table 582: Barcode types returned by GMI and GMA

<table>
<thead>
<tr>
<th>Barcode type (GMI/iOS)</th>
<th>Barcode type (GMA/Android)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AZTEC</td>
<td>AZTEC</td>
<td>Aztec barcode format</td>
</tr>
<tr>
<td>Unsupported</td>
<td>CODABAR</td>
<td>CODABAR format</td>
</tr>
<tr>
<td>CODE_39</td>
<td>CODE_39</td>
<td>AKA Alpha39, Code 3 of 9 or USD-3 format</td>
</tr>
<tr>
<td>CODE_93</td>
<td>CODE_93</td>
<td>Intermec (Canada Post) format</td>
</tr>
<tr>
<td>CODE_128</td>
<td>CODE_128</td>
<td>High-density barcode (128 chars) format</td>
</tr>
<tr>
<td>DATA_MATRIX</td>
<td>DATA_MATRIX</td>
<td>Data Matrix format</td>
</tr>
<tr>
<td>EAN_8</td>
<td>EAN_8</td>
<td>European/International Article Number (8 digits) format</td>
</tr>
<tr>
<td>EAN_13</td>
<td>EAN_13</td>
<td>European/International Article Number (13 digits) format</td>
</tr>
<tr>
<td>I2/5</td>
<td>I2/5</td>
<td>Interleaved 2 of 5 format</td>
</tr>
<tr>
<td>Unsupported</td>
<td>MAXICODE</td>
<td>ISO/IEC 16023 format</td>
</tr>
<tr>
<td>PDF_417</td>
<td>PDF_417</td>
<td>Portable Data File - 417 format</td>
</tr>
<tr>
<td>QR_CODE</td>
<td>QR_CODE</td>
<td>Quick Response Code format</td>
</tr>
<tr>
<td>Unsupported</td>
<td>RSS_14</td>
<td>GS1 DataBar (Reduce Space Symbology) format</td>
</tr>
<tr>
<td>Barcode type (GMI/iOS)</td>
<td>Barcode type (GMA/Android)</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Unsupported</td>
<td>RSS_EXPANDED</td>
<td>GS1 DataBar Expanded (Reduce Space Symbology expanded) format</td>
</tr>
<tr>
<td>Unsupported</td>
<td>UPC_A</td>
<td>Universal Product Code (12 digits) format</td>
</tr>
<tr>
<td>UPC_E</td>
<td>UPC_E</td>
<td>Universal Product Code (6 digits) format</td>
</tr>
<tr>
<td>Unsupported</td>
<td>UPC_EAN_EXTENSION</td>
<td>UPC/EAN extension format</td>
</tr>
</tbody>
</table>

**takePhoto**

Lets the user take a picture with the mobile device and returns the corresponding picture identifier.

**Syntax**

```javascript
takePhoto
```

**Usage**

The "takePhoto" front call invokes the mobile device's camera to let the user take a picture and returns the local path/URL on the mobile device to the picture.

If the photo is canceled by the user, the front call returns `NULL`.

**Important:** For GMA / Android™, using the `takePhoto` front call needs the `android.permission.WRITE_EXTERNAL_STORAGE` Dangerous Permission to be specified when building the APK. See Android permissions on page 3595 for more details.

The value returned in the `path` variable contains a reference to the system location of the picture on the mobile device. This path is platform dependent, and may change in future versions. Consider the path returned by this front call as an opaque local file identifier, and do not use this path as a persistent file name for the picture.

For more details about mobile image handling, see images handling on mobile devices.

**Related concepts**

- Using images on page 1147
  Describes how to use pictures in the forms of your application.
- `fgl_getfile()` on page 2286
  Retrieves a file from the front-end context to the virtual machine context.
- `choosePhoto` on page 2699
  Lets the user select a picture from the mobile device's photo gallery and returns a picture identifier.

**takeVideo**

Lets the user take a video with the mobile device and returns the corresponding video identifier.

**Syntax**

```javascript
takeVideo
```

**Usage**

1. `path` - Holds the device opaque path to the video.
Usage

The "takeVideo" front call invokes the mobile device's camera to let the user take a video and returns the local path/URL to the video on the mobile device.

If the photo is canceled by the user, the front call returns NULL.

Important: For GMA / Android™, using the takeVideo front call needs the android.permission.WRITE_EXTERNAL_STORAGE Dangerous Permission to be specified when building the APK. See Android permissions on page 3595 for more details.

The value returned in the path variable contains a reference to the system location of the video on the mobile device. This path is platform dependent, and may change in future versions. Consider the path returned by this front call as an opaque local file identifier, and do not use this path as a persistent file name for the video.

Once the video identifier/path is known, it is possible to fetch the video file from the device to the program context with the fgl_getfile() API. The procedure is similar to fetching photos from the device. For more details, see the section about video handling on mobile devices.

To play the video, you can perform a "launchURL" front call, with the opaque path returned by this front call.

Related concepts

Using images on page 1147
Describes how to use pictures in the forms of your application.

fgl_getfile() on page 2286
Retrieves a file from the front-end context to the virtual machine context.

chooseVideo on page 2700
Lets the user select a video from the mobile device's video gallery and returns a video identifier.

unregisterFromRemoteNotifications
This front call unregisters the mobile device from push notifications.

Syntax

ui.Interface.frontCall("mobile","unregisterFromRemoteNotifications",
                    [], [])

Usage

The "unregisterFromRemoteNotifications" front call unregisters the device from push notifications after it has been registered with the registerForRemoteNotifications on page 2708 front call.

Example

CALL ui.Interface.frontCall(
   "mobile", "unregisterFromRemoteNotifications",
   [ ], [ ] )

Related concepts

registerForRemoteNotifications on page 2708
This front call registers a mobile device for push notifications.

Genero Mobile Android™ front calls

This section describes front calls specific to the Android™ platform.

This table shows the functions implemented by the Android™ front-end in the "android" module.
Table 583: Android™ module front-end functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ui.Interface.frontCall(&quot;android&quot;,&quot;askForPermission&quot;, [permission], [result])</td>
<td>Ask the user to enable a dangerous feature on the Android device.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;android&quot;,&quot;showAbout&quot;, [])</td>
<td>Shows the GMA about box displaying version information.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;android&quot;,&quot;showSettings&quot;, [])</td>
<td>Shows the GMA settings box controlling debug options.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;android&quot;,&quot;startActivity&quot;, [action, data, category, type, component, extras], [])</td>
<td>Starts an external Android™ application (activity), and returns to the GMA application immediately.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;android&quot;,&quot;startActivityForResult&quot;, [action, data, category, type, component, extras], [outdata, outextras])</td>
<td>Starts an external application (Android™ activity) and waits until the activity is closed.</td>
</tr>
</tbody>
</table>

Related concepts

Genero Mobile common front calls on page 2697

This section describes common front calls provided by all mobile front-ends.

askForPermission (Android™)

Ask the user to enable a dangerous feature on the Android device.

Syntax

ui.Interface.frontCall("android","askForPermission", [permission], [result])

1. permission - Identifies the Android™ permission to enable.
2. result - Holds the execution status of the front call:
   • "ok": the user accepted the permission.
   • "rejected": the user refused the permission.

Usage

The "askForPermission" front call opens a message box, to let the end user confirm the access to a "Dangerous Permission" on Android™, to enable a risky feature of the mobile device.

Important: The askForPermission front call has been introduced for Android™ 6: Since this version of Android, permissions to access dangerous mobile functions are no longer asked during app installation: The app code must explicitly ask the user for dangerous permissions when needed, with an askForPermission front call.

The permissions parameter defines the Android™ permission to be asked. It must be a string representing one of the permission constants, as defined in Android's Manifest permissions, prefixed by the "android.permission." string. For example, the "android.permission.WRITE_EXTERNAL_STORAGE" string can be used to identify the permission to access the SDCARD storage unit.
**Important:** Android™ Dangerous Permissions required by the app also need to be specified when building the app. For more details, see Android permissions on page 3595.

The front call will raise a runtime exception if the permission identifier is not valid.

**Example**

The following code example asks the user to access the SDCARD, and handles the user choice:

```gen
DEFINE result STRING
CALL ui.Interface.frontCall(
  "android", "askForPermission",
  ["android.permission.WRITE_EXTERNAL_STORAGE"],
  [result] )
CASE result
  WHEN "ok"
    CALL os.Path.mkDir("/sdcard/myfiles")
  WHEN "rejected"
    ERROR "SDCARD access was denied by user"
END CASE
```

**Related concepts**

- Building Android apps with Genero on page 3591
  Genero provides a command-line tool to create applications for Android™ devices.

**showAbout (Android™)**

Shows the GMA about box displaying version information.

**Syntax**

```gen
ui.Interface.frontCall("android", "showAbout", [], [])
```

**Usage**

This front call simply shows a typical about box, indicating GMA version information.

**Important:** This front call is only available for an application running on an Android™ device.

No input parameters are required, and no parameters are returned.

**showSettings (Android™)**

Shows the GMA settings box controlling debug options.

**Syntax**

```gen
ui.Interface.frontCall("android", "showSettings", [], [])
```

**Usage**

This front call opens the settings box to enable or disable GMA programming options.

**Important:** This front call is only available for an application running on an Android™ device.

No input parameters are required, and no parameters are returned.

The following features can be controlled with the GMA settings box:

- HTTP debug server on port 6480 (to inspect the AUI tree and show app logs)
- GUI display (FGLSERVER on page 280) and remote debug with fgldb on port 6400
- Android™ logcat recording
• Managing allowed certificates (SSH connections)
• Cookies cleanup (for SSO authentication tokens)

Related concepts

Debugging a mobile app on page 3587
Different solutions are available to debug a mobile app.

**startActivity (Android™)**

Starts an external Android™ application (activity), and returns to the GMA application immediately.

**Syntax**

```javascript
ui.Interface.frontCall("android","startActivity",
    [action, data, category, type, component, extras],
    []
)
```

1. **action** - Identifies the activity to be started on the Android™ device.
2. **data** (optional) The data to operate on in the activity (URL, etc).
3. **category** (optional) A comma separated list of categories.
4. **type** (optional) Specifies the type of the data passed to the activity.
5. **component** (optional) Specifies a component class to use for the intent.
6. **extras** (optional) This is a JSON string containing parameters to pass to the activity.

**Usage**

The "startActivity" front call starts an external application (Android™ activity), and returns to the GMA application immediately after invoking the activity.

**Important:** This front call is only available for an application running on an Android™ device.

This front call is similar to the RUN WITHOUT WAITING statement: It allows the user to switch between the GMA and the launched application.

The parameters passed to this front call are used to build an Android™ "intent" object to start an "activity". For more details about Android™ intent object, refer to the Android™ Intent definition.

The **action** parameter defines the Android™ activity to perform, such as "android.intent.action.MAIN", "android.intent.action.VIEW", and so on.

The **data** (optional) parameter contains the data to operate on. This is the main parameter to transmit data to the activity. It can for example be an URL.

The **category** (optional) parameter contains a comma separated list of categories, where a category gives additional information about the action to execute. For example, "android.intent.category.LAUNCHER" means it appears in the Launcher as a top-level application. See the Android™ documentation for details about possible categories for a given activity.

The **type** (optional) parameter defines the type (in fact, a MIME type) of the activity data. Normally the type is inferred from the data itself. By setting this attribute, you disable that evaluation and force an explicit type.

The **component** (optional) parameter defines the name of a component class to use for the intent. Normally this is determined by looking at the other information in the intent. The component name typically specified as "apk-package-name/java-class-name" or "java-class-name" (the APK package name is optional). If the APK package is not specified, GMA considers that the Java class is included in the current APK.

The **extras** (optional) parameter specifies a JSON string containing parameters to pass to the activity. This can be used to provide extended information to the component. For example, with an action sending an e-mail message, the extra data can include data to supply a subject, body, for the e-mail.
Example

The following code example starts the VIEW Android™ activity to show an image. The Genero program flow will continue after this call, but the started activity will be shown. Note that such action is better performed with a `launchurl` front call.

```java
CALL ui.Interface.frontCall("android", "startActivity", [
    "android.intent.action.VIEW",
    "file:///storage/path_to_image_file",
    NULL, "image/*" ],
[ ] )
```

Related concepts

- **startActivityForResult (Android)** on page 2717
  Starts an external application (Android™ activity) and waits until the activity is closed.

- **startActivityForResult (Android™)**
  Starts an external application (Android™ activity) and waits until the activity is closed.

Syntax

```java
ui.Interface.frontCall("android", "startActivityForResult", [
    action, data, category, type, component, extras],
[ outdata, outextras])
```

1. `action` - Identifies the activity to be started on the Android™ device.
2. `data` - (optional) The data to operate on in the activity (URL, etc).
3. `category` - (optional) A comma separated list of categories.
4. `type` - (optional) Specifies the type of the data passed to the activity.
5. `component` - (optional) Specifies a component class to use for the intent.
6. `extras` - (optional) This is a JSON string containing parameters to pass to the activity.

Return values include:

1. `outdata` - holds the flat value returned by the invoked activity.
2. `outextras` - holds the JSON data of structured value returned by the invoked activity.

The return values depend entirely on the invoked activity.

Usage

The "startActivityForResult" front call starts an external application (Android™ activity), then waits for the user to exit the external application prior to returning the GMA application.

**Important:** This front call is only available for an application running on an Android™ device.

This front call is similar to the `RUN` statement: The user cannot return to the GMA application while the activity is executing.

The parameters passed to this front call are used to build an Android™ "intent" object to start an "activity". For more details about Android™ intent object, refer to the Android™ "Intent" definition.

The `action` parameter defines the Android™ activity to perform, such as "android.intent.action.MAIN", "android.intent.action.VIEW", and so on.

The `data` (optional) parameter contains the data to operate on. This is the main parameter to transmit data to the activity. It can for example be an URL.

The `category` (optional) parameter contains a comma separated list of categories, where a category gives additional information about the action to execute. For example, "android.intent.category.LAUNCHER" means
it appears in the Launcher as a top-level application. See the Android™ documentation for details about possible categories for a given activity.

The *type* (optional) parameter defines the type (in fact, a MIME type) of the activity data. Normally the type is inferred from the data itself. By setting this attribute, you disable that evaluation and force an explicit type.

The *component* (optional) parameter defines the name of a component class to use for the intent. Normally this is determined by looking at the other information in the intent. The component name typically specified as "apk-package-name/java-class-name" or "java-class-name" (the APK package name is optional). If the APK package is not specified, GMA considers that the Java class is included in the current APK.

The *extras* (optional) parameter specifies a JSON string containing parameters to pass to the activity. This can be used to provide extended information to the component. For example, with an action sending an e-mail message, the extra data can include data to supply a subject, body, for the e-mail.

The *outdata* returning argument will contain the flag value returned from the activity, typically when the data is simple and not structured.

The *outextras* returning argument can hold JSON data of any structured value returned by the invoked activity, or NULL in case of error (for example, when the application corresponding to the activity is not installed).

**Example**

This example invokes the barcode scanner application, and returns the scanned barcode.

```plaintext
IMPORT util
...
DEFINE data, extras STRING,
    json_object util.JSONObject,
    scanned_value STRING
...
CALL ui.Interface.frontCall(
    "android", "startActivityForResult",
    [ "com.google.zxing.client.android.SCAN",
        NULL, "android.intent.category.DEFAULT" ],
    [ data, extras ])
IF extras IS NULL THEN
   -- If the application isn’t installed invoke
   -- the Play Store to give the user a chance to install it
   CALL ui.Interface.frontCall("standard", "launchurl",
        ["market://details?id=com.google.zxing.client.android"], [ ])
ELSE
   LET json_object = util.JSONObject.parse(extras)
   -- Fetch the scanned value
   LET scanned_value = json_object.get("SCAN_RESULT")
END IF
```

**Related concepts**

*startActivity (Android)* on page 2716
Starts an external Android™ application (activity), and returns to the GMA application immediately.

**Genero Mobile iOS front calls**

This section describes front calls specific to the iOS platform.

This table shows the functions implemented by the iOS front-end in the "ios" module.
Table 584: iOS module front-end functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ui.Interface.frontCall(&quot;ios&quot;, &quot;getBadgeNumber&quot;, [], [value])</td>
<td>Returns the current badge number associated to the app.</td>
</tr>
<tr>
<td>ui.Interface.frontCall(&quot;ios&quot;, &quot;setBadgeNumber&quot;, [value], [])</td>
<td>Sets the current badge number associated to the app.</td>
</tr>
</tbody>
</table>

Related concepts

Genero Mobile common front calls on page 2697
This section describes common front calls provided by all mobile front-ends.

getBadgeNumber (iOS)
Returns the current badge number associated to the app.

Syntax

ui.Interface.frontCall("ios", "getBadgeNumber", [], [value])

• value - Holds the current badge number.

Usage
The iOS "getBadgeNumber" front call returns the current badge number associated to the app.

Important: This front call is only available for an application running on an iOS device.

The badge number appears on the app icon and is typically used for Push notifications on page 3619.

Important: In order to query or set the badge number, the app program must have executed a registerForRemoteNotifications front call before (in the current or prior execution instance). This registration is required in order to set the appropriate app permissions to access badge number data.

Example

DEFINE value INTEGER
CALL ui.interface.frontcall("ios","getBadgeNumber",[],[value])

Related concepts

Deploying mobile apps on iOS devices on page 3604
This section contains information to create a mobile application to be deployed on iOS devices.

setBadgeNumber (iOS) on page 2719
Sets the current badge number associated to the app.

setBadgeNumber (iOS)
Sets the current badge number associated to the app.

Syntax

ui.Interface.frontCall("ios", "setBadgeNumber", [value], [])

• value - Holds the badge number to be set.
Usage
The iOS "setBadgeNumber" front call sets the badge number associated to the app.

Important: This front call is only available for an application running on an iOS device.

The badge number appears on the app icon and is typically used for Push notifications on page 3619.

Important: In order to query or set the badge number, the app program must have executed a registerForRemoteNotifications front call before (in the current or prior execution instance). This registration is required in order to set the appropriate app permissions to access badge number data.

Example

```
DEFINE value INTEGER
LET value = 2
CALL ui.interface.frontcall("ios","setBadgeNumber",[value],[])
```

Related concepts
Deploying mobile apps on iOS devices on page 3604
This section contains information to create a mobile application to be deployed on iOS devices.

getBadgeNumber (iOS) on page 2719
Returns the current badge number associated to the app.

Cordova plugin front calls
Genero provides a set of Cordova plugin front calls that make use of the Cordova plugins.

This table shows the functions implemented by the Android™ and iOS front-end in the "cordova" module.
Table 585: Cordova front-end functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ui.Interface.frontCall(&quot;cordova&quot;, &quot;call&quot;, [plugin-name, function-name [, param1, param2, ... ]], [result])</code></td>
<td>Calls a function in a Cordova plugin and returns a result.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;cordova&quot;, &quot;callWithoutWaiting&quot;, [plugin-name, function-name [, param1, param2, ... ]], [callback-id])</code></td>
<td>Calls a function asynchronously in a Cordova plugin, without waiting for a result.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;cordova&quot;, &quot;getAllCallbackData&quot;, [callback-id-filter], [results])</code></td>
<td>Returns all results for asynchronous Cordova plugin front calls, based on a callback ID filter.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;cordova&quot;, &quot;getCallbackDataCount&quot;, []) , [count])</code></td>
<td>Returns the number of pending Cordova plugin results.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;cordova&quot;, &quot;getCallbackData&quot;, [], [result, callback-id])</code></td>
<td>Returns the first Cordova plugin result from the result queue of all asynchronous Cordova plugin front calls, and removes it from the queue.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;cordova&quot;, &quot;getPluginInfo&quot;, [plugin-name], [result])</code></td>
<td>Returns details about a specific Cordova plugin.</td>
</tr>
<tr>
<td><code>ui.Interface.frontCall(&quot;cordova&quot;, &quot;listPlugins&quot;, []) , [plugins])</code></td>
<td>Returns the list of available Cordova plugins.</td>
</tr>
</tbody>
</table>

Related concepts

**Cordova plugins** on page 3644

This section describes how to use Cordova plugins.

**call**

Calls a function in a Cordova plugin and returns a result.

**Syntax**

```javascript
ui.Interface.frontCall("cordova", "call", [plugin-name, function-name [, param1, param2, ... ]], [result])
```

1. `plugin-name` - This is the name of the Cordova plugin.
2. `function-name` - This is the name of plugin function to be called.
3. `param1, param2, ...` - Optional parameters to be passed to the Cordova function.
4. `result` - Holds the Cordova function return value.

**Usage**

The `call` front call executes synchronously the Cordova plugin function identified by the `plugin-name` and the `function-name`.

The other arguments (`param1, param2, ...`) are arguments for the Cordova plugin function. Each argument may have a different type like `FLOAT`, `INTEGER`, `STRING`, `RECORD` or `DYNAMIC ARRAY` (for `RECORD` and `DYNAMIC ARRAY`, the runtime system will do the BDL to JSON conversion automatically).

The front call returns one `result` variable of type `FLOAT`, `INTEGER`, `STRING`, `RECORD` or `DYNAMIC ARRAY`, that matches the JSON equivalent of the plugin function result (for `RECORD` and `DYNAMIC ARRAY`, the runtime system will do the JSON to BDL conversion automatically).

The `call` Cordova front call is synchronous: This means that the call will not return until the plugin returns a result. Some functions may not return a result at all (such as start/stop functions) and therefore would cause the front call to wait forever. Those functions need to be called asynchronously with the `callWithoutWaiting` front call.

In case of an error, the front call raises a runtime error `-6333` that can be caught with `TRY/CATCH` or `WHENEVER ERROR`.

**Note:** Use the `err_get()` function, to identify the reason of the error. For more details about front call error handling, see `ui.Interface.frontCall()`.

**Example**

```
DEFINE calendars DYNAMIC ARRAY OF STRING
CALL ui.interface.frontcall( "cordova", "call",
                           ["Calendar", "listCalendars"], [calendars] )
```

**callWithoutWaiting**

Calls a function asynchronously in a Cordova plugin, without waiting for a result.

**Syntax**

```
ui.Interface.frontCall( "cordova", "callWithoutWaiting",
                       [ plugin-name, function-name [, param1, param2, ... ] ],
                       [ callback-id ])
```

1. `plugin-name` - This is the name of the Cordova plugin.
2. `function-name` - This is the name of plugin function to be called.
3. `param1, param2, ...` - Optional parameters to be passed to the Cordova function.
4. `callback-id` - Holds the callback identifier for this Cordova asynchronous front call.

**Usage**

The `callWithoutWaiting` front call executes asynchronously the Cordova plugin function identified by the `plugin-name` and the `function-name`.

The other arguments (`param1, param2, ...`) are arguments for the Cordova plugin function. Each argument may have a different type like `FLOAT`, `INTEGER`, `STRING`, `RECORD` or `DYNAMIC ARRAY` (for `RECORD` and `DYNAMIC ARRAY`, the runtime system will do the BDL to JSON conversion automatically).

The `callWithoutWaiting` front call behaves the same as the `call` front call, but does not wait for a result from the plugin. Instead it returns directly a unique callback ID, to be able to identify the call later on. The program execution can continue, while the plugin processes the result asynchronously.

When the plugin produces the result, it is stored internally by GMI/GMA in a result queue. Then result data can be retrieved with `getCallbackData/getAllCallbackData` front calls in conjunction with the "cordovaCallback" action.
When the current dialog contains an ON ACTION handler for the `cordovaCallback` predefined action, this action is triggered by the front-end to notify the Genero program that there is plugin data to fetch.

The `cordovaCallback` action is triggered:

1. If the result queue is empty, and a result is added to the queue.
2. If a new dialog that contains this action is entered or re-entered, and the result queue is non empty.

**Important:** Some plugin functions may return results repeatedly in a short time intervals with one and the same `callback-id`, examples are for delivering motion/audio/bluetooth data.

In case of an error, the front call raises a runtime error -6333 that can be caught with `TRY/CATCH` or `WHENEVER ERROR`.

**Note:** Use the `err_get()` function, to identify the reason of the error. For more details about front call error handling, see `ui.Interface.frontCall()`.

**Example**

```
DEFINE id, song STRING
CALL ui.interface.frontcall("cordova", "callWithoutWaiting",
    ["Media", "play", song], [id])
...
    ON ACTION cordovacallback ATTRIBUTE(DEFAULTVIEW=NO)  
        -- Process results
...
```

### getAllCallbackData

Returns all results for asynchronous Cordova plugin front calls, based on a callback ID filter.

**Syntax**

```
ui.Interface.frontCall("cordova", "getAllCallbackData",
    [callback-id-filter], [results])
```

1. `callback-id-filter` - Provides the callback ID filter for Cordova asynchronous front calls.
2. `results` - The array of results returned by the Cordova asynchronous front call identified by `callback-id`.

**Usage**

When initiating an asynchronous Cordova plugin front call with `callWithoutWaiting`, results are stored in the result queue when the Cordova function terminates, and a `cordovaCallback` action is fired if the current dialog defines a corresponding ON ACTION handler.

The purpose of the `getAllCallbackData` front call is to retrieve high traffic data in bloc (such as motion sensor data) to avoid too many front calls in a short time frame.

The `getAllCallbackData` front call returns all results produced by Cordova asynchronous front calls, that match the `callback-id-filter` provided as parameter. For example, if you specify "Media-" as filter, all results related to callback ids starting with "Media-" will be retrieved.

All results in the queue matching the filter are retrieved into the array, and are removed from the result queue.

The results are returned into a DYNAMIC ARRAY OF STRING or a DYNAMIC ARRAY OF RECORD, a structured record to be defined as the JSON equivalent of the plugin function result (when using a DYNAMIC ARRAY OF RECORD, the runtime system can do the JSON to BDL conversion automatically).

In case of an error, the front call raises a runtime error -6333 that can be caught with `TRY/CATCH` or `WHENEVER ERROR`.

**Note:** Use the `err_get()` function, to identify the reason of the error. For more details about front call error handling, see `ui.Interface.frontCall()`.
Example

DEFINE events DYNAMIC ARRAY OF RECORD
  x FLOAT,
  y FLOAT,
  z FLOAT,
  timestamp DECIMAL
END RECORD
DEFINE id STRING
DEFINE x INTEGER

CALL ui.interface.frontcall("cordova", "callWithoutWaiting",
  ["Accelerometer","start"], [id])

ON IDLE 2
  CALL ui.interface.frontcall("cordova", "getAllCallbackData",
    [id], [events] )
  FOR x=1 TO events.getLength()
    CALL process_motion_event( events[x].* )
  END FOR

Related concepts

getCallbackDataCount on page 2724
Returns the number of pending Cordova plugin results.

getCallbackDataCount
Returns the number of pending Cordova plugin results.

Syntax

ui.Interface.frontCall("cordova", "getCallbackDataCount",
  [], [count])

1. count - Holds the number of pending results.

Usage

When initiating an asynchronous Cordova plugin front call with callWithoutWaiting, results are stored in the result queue when the Cordova function terminates, and a cordovacallback action is fired if the current dialog defines a corresponding ON ACTION handler.

The getCallbackDataCount front call returns the number of results currently in the result queue, for all asynchronous Cordova plugin front calls initiated by a callWithoutWaiting.

It is then possible to implement a FOR loop to retrieve all results with the getCallbackData front call.

In case of an error, the front call raises a runtime error -6333 that can be caught with TRY/CATCH or WHENEVER ERROR.

Note: Use the err_get() function, to identify the reason of the error. For more details about front call error handling, see ui.Interface.frontCall().

Example

DEFINE cnt SMALLINT
CALL ui.Interface.frontCall("cordova", "getCallbackDataCount",
  [], [cnt])
**getCallbackData**

Returns the first Cordova plugin result from the result queue of all asynchronous Cordova plugin front calls, and removes it from the queue.

**Syntax**

```
ui.Interface.frontCall("cordova", "getCallbackData", [], [result, callback-id])
```

1. **result** - Holds the result returned from the result queue.
2. **callback-id** - Holds the callback identifier of the Cordova asynchronous front call.

**Usage**

When initiating an asynchronous Cordova plugin front call with `callWithoutWaiting`, results are stored in the result queue when the Cordova function terminates, and a `cordovaCallback` action is fired if the current dialog defines a corresponding `ON ACTION` handler.

The `getCallbackData` front call returns the first Cordova plugin result from the result queue, and removes it from the queue. A subsequent `getCallbackData` front call gives back the next result and so on.

The first value returned by the front call (`result`) is the actual result. This can be a variable of type `FLOAT`, `INTEGER`, `STRING`, `RECORD` or `DYNAMIC ARRAY`, that matches the JSON equivalent of the plugin function result (for `RECORD` and `DYNAMIC ARRAY`, the runtime system will do the JSON to BDL conversion automatically).

The second returned value is `callback-id`, the identifier returned by the `callWithoutWaiting` front call that is causing this result. For example, for a Media plugin front call, the `callback-id` can look like:

```
Media-messageChannel:0
```

If the result queue is empty, both `result` and `callback-id` are NULL.

In case of an error, the front call raises a runtime error -6333 that can be caught with `TRY/CATCH` or `WHENEVER ERROR`.

**Note:** Use the `err_get()` function, to identify the reason of the error. For more details about front call error handling, see `ui.Interface.frontCall()`.

**Example**

```
DEFINE res, id STRING
CALL ui.Interface.frontCall("cordova", "getCallbackData", [], [res, id])
```

**Related concepts**

- **getCallbackDataCount** on page 2724
  Returns the number of pending Cordova plugin results.

- **getPluginInfo**
  Returns details about a specific Cordova plugin.

**Syntax**

```
ui.Interface.frontCall("cordova", "getPluginInfo", [plugin-name], [result])
```

1. **plugin-name** - This is the name of the Cordova plugin.
2. **result** - Holds details about the Cordova plugin, in JSON format.
Usage

The `getPluginInfo` front call returns details about the Cordova plugin passed as parameter. The `result` variable is of type `STRING` filled with a JSON string containing the details about the plugin such as:

1. The `id` of the plugin
2. The `version` of the plugin
3. The `git_version` of the plugin

Example

```c
DEFINE result STRING
CALL ui.interface.frontcall( "cordova", "getPluginInfo", [], [result] )
DISPLAY util.JSON.format(result)
```

listPlugins

Returns the list of available Cordova plugins.

Syntax

```c
ui.Interface.frontCall("cordova", "listPlugins", [], [plugins])
```

1. `plugins` - Is a `DYNAMIC ARRAY OF STRING`, to hold the list of available plugins.

Usage

The `listPlugins` front call fetches the names of available Cordova plugins bundled with the app. The `result` variable must be of type `DYNAMIC ARRAY OF STRING`.

Example

```c
DEFINE plugins DYNAMIC ARRAY OF STRING
CALL ui.interface.frontcall( "cordova", "listPlugins", [], [plugins] )
```

Extension packages

Several utility classes and functions are provided in additional packages.

Note: Unlike built-in packages, extension packages must be imported with the `IMPORT` statement.

The `util` package

These topics cover the classes for the `util` package.

The `util.Date` class

The `util.Date` class provides `DATE` data-type related utility methods.

This class is provided in the `util C-Extension` library; To use this class, import the `util` package with:

```c
IMPORT util
```

This class does not have to be instantiated; it provides class methods for the current program.

Related concepts

`IMPORT C-Extension` on page 481
The `IMPORT` instruction imports C extension module elements to be used by the current module.

**util.Date methods**
Methods for the `util.Date` class.

**Table 586: Class methods**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>util.Date.isLeapYear(year INTEGER)</code></td>
<td>Checks if the year passed as parameter is a leap year.</td>
</tr>
<tr>
<td><code>util.Date.parse(s STRING, format STRING)</code></td>
<td>Converts a string to a DATE value based on a format specification.</td>
</tr>
</tbody>
</table>

**util.Date.parse**
Converts a string to a DATE value based on a format specification.

**Syntax**

```
util.Date.parse(
  s STRING,
  format STRING
)
RETURNS DATE
```

1.  `s` is the source string to be parsed.
2.  `format` is the format specification (see Formatting DATE values on page 321).

**Usage**
The `util.Date.parse()` method parses a string based on a format specification, to produce a DATE value.
The format specification must be a combination of `dd`, `mm`, `yyyy` place holders as used in the `USING` operator.
The method returns NULL, if the source string cannot be converted to a DATE value based on the format specification.
For more details about the supported formats, see Formatting DATE values on page 321.

**Example**

```
IMPORT util
MAIN
  DISPLAY util.Date.parse( "2014-03-15", "yyyy-mm-dd" )
END MAIN
```

**Related concepts**

USING on page 357
The **USING** operator converts date and numeric values to a string based on a formatting mask.

**`util.Date.isLeapYear`**
Checks if the year passed as parameter is a leap year.

### Syntax

```
util.Date.isLeapYear(
    year INTEGER
)  
RETURNS BOOLEAN
```

1. `year` is an INTEGER representing a year.

### Usage
The `util.Date.isLeapYear()` method returns TRUE if the year passed in parameter is a leap year.

### Example

```
IMPORT util
MAIN
    DISPLAY util.Date.isLeapYear( 2003 )
    DISPLAY util.Date.isLeapYear( 2004 )
END MAIN
```

### The **util.Datetime** class

The **util.Datetime** class provides DATETIME data-type related utility methods.

This class is provided in the **util C-Extension** library; To use this class, import the **util** package with:

```
IMPORT util
```

This class does not have to be instantiated; it provides class methods for the current program.

**Related concepts**

IMPORT C-Extension on page 481
The IMPORT instruction imports c extension module elements to be used by the current module.

**util.Datetime methods**  
Methods for the util.Datetime class.

**Table 587: Class methods**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>util.Datetime.format</code> (</td>
<td></td>
</tr>
<tr>
<td><code>t DATETIME q1 TO q2,</code></td>
<td></td>
</tr>
<tr>
<td><code>format STRING</code></td>
<td></td>
</tr>
<tr>
<td>)</td>
<td></td>
</tr>
<tr>
<td>RETURNS STRING</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Formats a date/time value based on a specified format.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>util.Datetime.fromSecondsSinceEpoch</code> (</td>
<td></td>
</tr>
<tr>
<td><code>t FLOAT</code></td>
<td></td>
</tr>
<tr>
<td>)</td>
<td></td>
</tr>
<tr>
<td>RETURNS DATETIME q1 TO q2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Converts a number of seconds since Epoch to a date/time.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>util.Datetime.getCurrentAsUTC</code> ( )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RETURNS DATETIME YEAR TO FRACTION(5)</td>
</tr>
<tr>
<td></td>
<td>Returns the current date/time in UTC.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>util.Datetime.parse</code> (</td>
<td></td>
</tr>
<tr>
<td><code>s STRING,</code></td>
<td></td>
</tr>
<tr>
<td><code>format STRING</code></td>
<td></td>
</tr>
<tr>
<td>)</td>
<td></td>
</tr>
<tr>
<td>RETURNS DATETIME q1 TO q2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Converts a string to a DATETIME value based on a specified format.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>util.Datetime.toLocalTime</code> (</td>
<td></td>
</tr>
<tr>
<td><code>t DATETIME q1 TO q2</code></td>
<td></td>
</tr>
<tr>
<td>)</td>
<td></td>
</tr>
<tr>
<td>RETURNS DATETIME q1 TO q2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Converts a UTC date/time to the local time.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>util.Datetime.toSecondsSinceEpoch</code> (</td>
<td></td>
</tr>
<tr>
<td><code>t DATETIME q1 TO q2</code></td>
<td></td>
</tr>
<tr>
<td>)</td>
<td></td>
</tr>
<tr>
<td>RETURNS FLOAT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Converts a date/time to a number of seconds since Epoch.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>util.Datetime.toUTC</code> (</td>
<td></td>
</tr>
<tr>
<td><code>t DATETIME q1 TO q2</code></td>
<td></td>
</tr>
<tr>
<td>)</td>
<td></td>
</tr>
<tr>
<td>RETURNS DATETIME q1 TO q2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Converts a date/time value to the UTC date/time.</td>
</tr>
</tbody>
</table>

**util.Datetime.format**  
Formats a date/time value based on a specified format.

**Syntax**

```
util.Datetime.format (  
   `t DATETIME q1 TO q2,`  
   `format STRING`  
)
```
1. \( t \) is the date/time value to be formatted.
2. `format` is the format string, as described in Formatting DATETIME values on page 322.

**Usage**
The `util.Datetime.format()` method formats a DATETIME value based on the format specification.

**Note:** The `format()` method is designed to accept DATETIME values. However, since Genero BDL supports implicit type conversion, it is also possible to pass a DATE value to the `format()` method.

The format string must be a combination of place holders such as `%Y`, `%m`, `%d`, as described in Formatting DATETIME values on page 322.

If the source value is NULL the result will be NULL.

**Example**

```plaintext
IMPORT util
MAIN
  DISPLAY util.Datetime.format( CURRENT, "%Y-%m-%d %H:%M" )
END MAIN
```

**Related concepts**
util.Datetime.parse on page 2731
Converts a string to a DATETIME value based on a specified format.

util.Datetime.fromSecondsSinceEpoch
Converts a number of seconds since Epoch to a date/time.

**Syntax**

```plaintext
util.Datetime.fromSecondsSinceEpoch( 
  t FLOAT
) 
RETURNS DATETIME q1 TO q2
```

1. \( t \) is the number of seconds since Epoch. This can be a whole integer or a decimal, in the target datetime.

**Usage**
The `util.Datetime.fromSecondsSinceEpoch()` method converts the number of seconds since the UNIX® Epoch (1970-01-01 00:00:00 GMT) passed as parameter, to a DATETIME value in local time.

**Important:** If the number of seconds passed as parameter is a floating point number including a fraction of seconds, the result will be a DATETIME YEAR TO FRACTION(N), otherwise it is DATETIME YEAR TO SECOND.

**Example**

```plaintext
IMPORT util
MAIN
  DEFINE dt DATETIME YEAR TO SECOND
  LET dt = util.Datetime.fromSecondsSinceEpoch( 9876234 )
  DISPLAY dt
END MAIN
```

**Related concepts**
util.Datetime.toSecondsSinceEpoch on page 2733
Converting a date/time to a number of seconds since Epoch.

`util.Datetime.getCurrentAsUTC`

Returns the current date/time in UTC.

**Syntax**

```plaintext
util.Datetime.getCurrentAsUTC( )
RETURNS DATETIME YEAR TO FRACTION(5)
```

**Usage**

The `util.Datetime.getCurrentAsUTC()` method returns the current system date/time in UTC (Universal Time).

This method is provided to solve the daylight saving time transition issue of the `util.Datetime.toUTC()` method.

**Note:** The precision of the value returned by this method is a `DATETIME YEAR TO FRACTION(5)`. Note that this precision is different from the default `CURRENT` precision when no qualifiers are specified.

**Example**

```plaintext
IMPORT util
MAIN
  DEFINE utc DATETIME YEAR TO FRACTION(5)
  LET utc = util.Datetime.getCurrentAsUTC( )
  DISPLAY "Current UTC: ", utc
END MAIN
```

**Related concepts**

- `util.Datetime.toLocalTime` on page 2732
  Converts a UTC date/time to the local time.

- `util.Datetime.parse`
  Converts a string to a `DATETIME` value based on a specified format.

**Syntax**

```plaintext
util.Datetime.parse(
  s STRING,
  format STRING
)
RETURNS DATETIME q1 TO q2
```

1. `s` is the source string to be parsed.
2. `format` is the format specification (see Formatting `DATETIME` values on page 322).

**Usage**

The `util.Datetime.parse()` method parses a string based on a format specification, to produce a `DATETIME` value.

**Note:** The `parse()` method produces a `DATETIME` value. However, since Genero BDL supports implicit type conversion, it is possible to assign a `DATE` variable with the value returned from `parse()`, as long as the `DATETIME` contains a date part.

The format specification must be a combination of place holders such as `%Y`, `%m`, `%d`, etc.
The precision of the resulting DATETIME value depends on the format specification. For example, when using "%y-%m-%d %H:%M", the resulting value will be a DATETIME YEAR TO MINUTE.

The method returns NULL, if the source string cannot be converted to a DATETIME value based on the format specification.

For more details about the supported formats, see Formatting DATETIME values on page 322.

Example

```import util
main
  define dt datetime year to minute
  let dt = util.datetime.parse( "2014-12-24 23:45", "%Y-%m-%d %H:%M" )
  display dt
end main
```

Related concepts

util.Datetime.format on page 2729
Formats a date/time value based on a specified format.

util.Datetime.toLocalTime
Converts a UTC date/time to the local time.

Syntax

```util.datetime.toLocalTime( t datetime q1 to q2 )
returns datetime q1 to q2```

1. `t` is the datetime value in UTC.

Usage

The `util.Datetime.toLocalTime()` method converts a DATETIME value from "Coordinated Universal Time" (UTC), also known as "Greenwich Mean Time" (GMT), to the local timezone date/time.

Example

```import util
main
  define loc datetime year to second
  let loc = util.datetime.toLocalTime( datetime(2015-08-22 15:34:56) year to second )
  display "LOC: ", loc
end main```

Related concepts

util.Datetime.toUTC on page 2733
Converts a date/time value to the UTC date/time.

util.Datetime.getCurrentAsUTC on page 2731
Returns the current date/time in UTC.

`util.Datetime.toSecondsSinceEpoch`
Converts a date/time to a number of seconds since Epoch.

**Syntax**

```plaintext
util.Datetime.toSecondsSinceEpoch(
  t DATETIME q1 TO q2
)
RETURNS FLOAT
```

1. `t` is the local datetime value.

**Usage**

The `util.Datetime.toSecondsSinceEpoch()` method converts the `DATETIME` value passed as parameter to a number of seconds since the UNIX® Epoch (1970-01-01 00:00:00 GMT)

**Important:** The result is a whole number when the source is a `DATETIME YEAR TO SECOND`, but will be a floating point number when the source is a `DATETIME YEAR TO FRACTION(N)`, to include the fractional part.

**Example**

```plaintext
IMPORT util
MAIN
  DEFINE sec INTEGER, loc DATETIME YEAR TO SECOND
  LET loc = CURRENT YEAR TO SECOND
  LET sec = util.Datetime.toSecondsSinceEpoch( loc )
  DISPLAY sec
END MAIN
```

**Related concepts**

- `util.Datetime.fromSecondsSinceEpoch` on page 2730
  Converts a number of seconds since Epoch to a date/time.

- `util.Datetime.toUTC`
  Converts a date/time value to the UTC date/time.

**Syntax**

```plaintext
util.Datetime.toUTC(
  t DATETIME q1 TO q2
)
RETURNS DATETIME q1 TO q2
```

1. `t` is the local timezone datetime value.

**Usage**

The `util.Datetime.toUTC()` method converts the local timezone `DATETIME` value passed as parameter to the "Coordinated Universal Time" (UTC), also known as "Greenwich Mean Time" (GMT).

The `toUTC()` method on local timezone information settings.

**Fall/Autumn daylight saving time transition period**

**Important:** The `toUTC()` function cannot determine if the local date/time value represents a time before or after the daylight saving time change, when the value is in the hour of the daylight saving time transition period in the fall (this is for example, the hour 02:00 PM to 03:00 PM on the last Sunday of October in Europe.
and first Sunday of November in the USA). Depending on the operating system, the `toUTC()` method can interpret the local time as summer time or as winter time. In order to get the current system time in UTC, use the `util.Datetime.getCurrentAsUTC()` method.

The `DATETIME` value passed as parameter to the `toUTC()` method is the date/time in the local timezone. However, this value does not contain the GMT offset indicator or daylight saving time information.

When passing local date/time values in the hour of the daylight saving time transition period in the fall (when clocks roll back one hour), the `toUTC()` function cannot determine if the local date/time value represents a point in time before or after the daylight saving time transition occurred. Depending on the operating system, the `toUTC()` method can interpret the local time as summer time or as a winter time. As a result, the conversion to the UTC time can be misinterpreted.

For example, in Europe, the fall daylight saving time changes on the 25 of October, at 3:00 PM. The ambiguous period is between 2:00 PM and 3:00 PM (local time). If you pass, for example, the date/time value 2015-10-25 02:34:11 to the `toUTC()` method, there is no way for the method to know if this local time is the time before (CET / UTC+1h) or after (CET / UTC+1h) the daylight saving time change.

This behavior can be illustrated with the following code example:

```c
IMPORT util

MAIN

DISPLAY "Original UTC         Local time (Paris)   toUTC(local-time)     ( toUTC() - Orig UCT )"
CALL test( "2015-10-24 23:59:59" )
CALL test( "2015-10-25 00:59:59" )
CALL test( "2015-10-25 01:59:59" )
CALL test( "2015-10-25 02:59:59" )
END MAIN

FUNCTION test(utc)
DEFINE utc, loc, utc2 DATETIME YEAR TO SECOND
LET loc = util.Datetime.toLocalTime(utc)
LET utc2 = util.Datetime.toUTC(loc)
DISPLAY SFMT("%1  %2  %3  %4", utc,loc,utc2,utc2-utc)
END FUNCTION
```

The above code will produce the following output on Linux®, with `TZ='Europe/Paris'`:

<table>
<thead>
<tr>
<th>Original UTC</th>
<th>Local time (Paris)</th>
<th>toUTC(local-time)</th>
<th>( toUTC() - Orig UCT )</th>
</tr>
</thead>
<tbody>
<tr>
<td>00:00:00</td>
<td>2015-10-25 02:59:59</td>
<td>2015-10-25 00:59:59</td>
<td>0</td>
</tr>
<tr>
<td>00:00:00</td>
<td>2015-10-25 03:59:59</td>
<td>2015-10-25 01:59:59</td>
<td>0</td>
</tr>
<tr>
<td>00:00:00</td>
<td>2015-10-25 04:59:59</td>
<td>2015-10-25 02:59:59</td>
<td>0</td>
</tr>
</tbody>
</table>

As you can see, the local time 2015-10-25 02:59:59 is always converted to UTC 2015-10-25 00:59:59.

**Example**

```c
IMPORT util

MAIN

DEFINE utc DATETIME YEAR TO SECOND
LET utc = util.Datetime.toUTC( DATETIME(2015-08-22 15:34:56) YEAR TO SECOND )
DISPLAY "UTC: ", utc
END MAIN
```
Related concepts
util.Datetime.toLocalTime on page 2732
Converts a UTC date/time to the local time.

The util.Integer class
The util.Integer class provides INTEGER data-type related utility methods.

This class is provided in the util C-Extension library; To use this class, import the util package with:

```
IMPORT util
```

This class does not have to be instantiated; it provides class methods for the current program.

Related concepts
IMPORT C-Extension on page 481
The IMPORT instruction imports C extension module elements to be used by the current module.

**util.Integer methods**
Methods for the `util.Integer` class.
## Table 588: Class methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>util.Integer.abs</code> (i INTEGER)</td>
<td>Returns the absolute value of an integer.</td>
</tr>
<tr>
<td><code>util.Integer.and</code> (x INTEGER, y INTEGER)</td>
<td>Returns the result of a bitwise AND on two INTEGER values.</td>
</tr>
<tr>
<td><code>util.Integer.andNot</code> (x INTEGER, y INTEGER)</td>
<td>Returns the result of a bitwise AND of the 1st INTEGER and the inverted 2nd INTEGER.</td>
</tr>
<tr>
<td><code>util.Integer.clearBit</code> (i INTEGER, n SMALLINT)</td>
<td>Returns the INTEGER parameter with the bit at the designated position set to 0.</td>
</tr>
<tr>
<td><code>util.Integer.not</code> (i INTEGER)</td>
<td>Returns the INTEGER value with all bits inverted.</td>
</tr>
<tr>
<td><code>util.Integer.or</code> (x INTEGER, y INTEGER)</td>
<td>Returns the result of a bitwise OR on two INTEGER values.</td>
</tr>
<tr>
<td><code>util.Integer.parseBinaryString</code> (s STRING)</td>
<td>Returns an INTEGER from its binary (base 2) string representation.</td>
</tr>
<tr>
<td><code>util.Integer.parseHexString</code> (s STRING)</td>
<td>Returns an INTEGER from its hexadecimal (base 16) string representation.</td>
</tr>
<tr>
<td><code>util.Integer.setBit</code> (i INTEGER, n INTEGER)</td>
<td>Returns the INTEGER parameter with the bit at the designated position set to 1.</td>
</tr>
<tr>
<td><code>util.Integer.shiftLeft</code> (i INTEGER, n SMALLINT)</td>
<td>Returns the INTEGER value left-shifted by the given bit places.</td>
</tr>
</tbody>
</table>
**util.Integer.abs**
Returns the absolute value of an integer.

**Syntax**

```plaintext
util.Integer.abs(
    i INTEGER
)  RETURNS INTEGER
```

1. `i` is the integer value to convert.

**Usage**
The `util.Integer.abs()` method converts the value passed as parameter to a positive integer when it is negative.

**Example**

```plaintext
IMPORT util
MAIN
    DISPLAY util.Integer.abs( -234 ) -- displays 234
END MAIN
```

**util.Integer.and**
Returns the result of a bitwise AND on two INTEGER values.

**Syntax**

```plaintext
util.Integer.and(
    x INTEGER,
    y INTEGER
)  RETURNS INTEGER
```

1. `x` is an integer value.
2. `y` is an integer value.

**Usage**
The `util.Integer.and()` method makes a bitwise AND operation with the integer values passed as parameter:

```plaintext
x       : 210 (00000000 00000000 00000000 11010010)
y       : 135 (00000000 00000000 00000000 10000011)
----------------------------------------------------
result   : 130 (00000000 00000000 00000000 10000010)
```

**Note:** Bitwise methods provided by the `util.Integer` class are based on the INTEGER type. Consider the following facts when using these methods:

1. The INTEGER type is a four-byte signed integer: If the bit at position 31 is set to 1, the corresponding INTEGER value will be negative. Thus, `util.Integer.not(0)` produces the INTEGER value -1 (11111111 11111111 11111111 11111111).
2. The NULL value for the INTEGER type is represented internally with the value 0x80000000 (10000000 00000000 00000000 00000000). When NULL is used with the `util.Integer` bitwise methods, it will be interpreted as 0x80000000 instead of a null value. However, if the result of the bitwise operation produces the value 0x80000000, it will be interpreted as NULL when used in an expression.
Example

```
IMPORT util
MAIN
    DISPLAY util.Integer.and( 3, 1 ) -- displays 2
    DISPLAY util.Integer.and( util.Integer.parseBinaryString("111"),
                             util.Integer.parseBinaryString("010") ) --
                              displays 2
END MAIN
```

Related concepts

**util.Integer.andNot** on page 2739
Returns the result of a bitwise AND of the 1st INTEGER and the inverted 2nd INTEGER.

**util.Integer.not** on page 2741
Returns the INTEGER value with all bits inverted.

**util.Integer.or** on page 2742
Returns the result of a bitwise OR on two INTEGER values.

**util.Integer.xor** on page 2749
Returns the result of a bitwise XOR on two INTEGER values.

**util.Integer.andNot**
Returns the result of a bitwise AND of the 1st INTEGER and the inverted 2nd INTEGER.

Syntax

```
util.Integer.andNot(
    x INTEGER,
    y INTEGER
)
RETURNS INTEGER
```

1. x is an integer value.
2. y is an integer value.

Usage

The `util.Integer.andNot()` method makes a bitwise AND operation with the first integer value and the second inverted integer:

```
x : 210 (00000000 00000000 00000000 11010010)
y : 135 (00000000 00000000 00000000 10000111)
~y : (11111111 11111111 11111111 01111000)
-------------------------
result : 80 (00000000 00000000 00000000 01010000)
```

This method simplifies mask operations.

**Note:** Bitwise methods provided by the `util.Integer` class are based on the INTEGER type. Consider the following facts when using these methods:

1. The INTEGER type is a four-byte signed integer: If the bit at position 31 is set to 1, the corresponding INTEGER value will be negative. Thus, `util.Integer.not(0)` produces the INTEGER value -1 (11111111 11111111 11111111 01111000).
2. The NULL value for the INTEGER type is represented internally with the value 0x80000000 (10000000 00000000 00000000 00000000). When NULL is used with the `util.Integer` bitwise methods, it will be interpreted as 0x80000000 instead of a null value. However, if the result of the bitwise operation produces the value 0x80000000, it will be interpreted as NULL when used in an expression.
Example

```sql
IMPORT util
MAIN
    DISPLAY util.Integer.andNot( 3, 2 ) -- displays 1
END MAIN
```

Related concepts

util.Integer.and on page 2738
Returns the result of a bitwise AND on two INTEGER values.

util.Integer.not on page 2741
Returns the INTEGER value with all bits inverted.

util.Integer.or on page 2742
Returns the result of a bitwise OR on two INTEGER values.

util.Integer.xor on page 2749
Returns the result of a bitwise XOR on two INTEGER values.

util.Integer.clearBit
Returns the INTEGER parameter with the bit at the designated position set to 0.

Syntax

```sql
util.Integer.clearBit(
    i INTEGER,
    n SMALLINT
)
RETURNS INTEGER
```

1. `i` is the integer value to modify.
2. `n` is the bit position (LSB is at position zero).

Usage

The `util.Integer.clearBit()` method modifies the integer value by clearing the bit at the position passed as second parameter.

Note: The position of the least significant bit (LSB) is the position zero:

- In 00000001 (integer 1), the bit set to 1 is at position 0.
- In 00100010 (integer 34), the bits set to 1 are at position 1 and 5.

Note: Bitwise methods provided by the `util.Integer` class are based on the INTEGER type. Consider the following facts when using these methods:

1. The INTEGER type is a four-byte signed integer: If the bit at position 31 is set to 1, the corresponding INTEGER value will be negative. Thus, `util.Integer.not(0)` produces the INTEGER value -1 (11111111 11111111 11111111 11111111).
2. The NULL value for the INTEGER type is represented internally with the value 0x80000000 (10000000 00000000 00000000 00000000). When NULL is used with the `util.Integer` bitwise methods, it will be interpreted as 0x80000000 instead of a null value. However, if the result of the bitwise operation produces the value 0x80000000, it will be interpreted as NULL when used in an expression.

Example

```sql
IMPORT util
MAIN
    DISPLAY util.Integer.clearBit( 34, 5 ) -- displays 2
END MAIN
```
Related concepts

util.Integer.setBit on page 2744
Returns the INTEGER parameter with the bit at the designated position set to 1.

util.Integer.testBit on page 2747
Returns TRUE, if in the INTEGER value, the bit at the designed position is set.

util.Integer.not
Returns the INTEGER value with all bits inverted.

Syntax

```plaintext
util.Integer.not (i INTEGER)
RETURNS INTEGER
```

1. `i` is the integer value to convert.

Usage

The `util.Integer.not()` method inverts all bits of the integer values passed as parameter:

<table>
<thead>
<tr>
<th>i</th>
<th>210 (00000000 00000000 00000000 11010010)</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>-211 (11111111 11111111 11111111 00101101)</td>
</tr>
</tbody>
</table>

Note: Bitwise methods provided by the `util.Integer` class are based on the INTEGER type. Consider the following facts when using these methods:

1. The INTEGER type is a four-byte signed integer: If the bit at position 31 is set to 1, the corresponding INTEGER value will be negative. Thus, `util.Integer.not(0)` produces the INTEGER value -1 (11111111 11111111 11111111 11111111).
2. The NULL value for the INTEGER type is represented internally with the value 0x80000000 (00000000 00000000 00000000 00000000). When NULL is used with the `util.Integer` bitwise methods, it will be interpreted as 0x80000000 instead of a null value. However, if the result of the bitwise operation produces the value 0x80000000, it will be interpreted as NULL when used in an expression.

Example

```plaintext
IMPORT util
MAIN
    DISPLAY util.Integer.not(
        util.Integer.parseBinaryString("11111111111111111111111111111110") ) -- displays 1
END MAIN
```

Related concepts

util.Integer.and on page 2738
Returns the result of a bitwise AND on two INTEGER values.

util.Integer.andNot on page 2739
Returns the result of a bitwise AND of the 1st INTEGER and the inverted 2nd INTEGER.

util.Integer.or on page 2742
Returns the result of a bitwise OR on two INTEGER values.

util.Integer.xor on page 2749
Returns the result of a bitwise XOR on two INTEGER values.

util.Integer.or
Returns the result of a bitwise OR on two INTEGER values.

Syntax

```java
util.Integer.or(
    x INTEGER,
    y INTEGER
)
RETURNS INTEGER
```

1. x is an integer value.
2. y is an integer value.

Usage

The `util.Integer.or()` method makes a bitwise OR operation with the integer values passed as parameter:

```
x     : 210 (00000000 00000000 00000000 11010010)
y     : 135 (00000000 00000000 00000000 10000111)
--------------------------------------------------
result : 215 (00000000 00000000 00000000 11010111)
```

Note: Bitwise methods provided by the `util.Integer` class are based on the INTEGER type. Consider the following facts when using these methods:

1. The INTEGER type is a four-byte signed integer: If the bit at position 31 is set to 1, the corresponding INTEGER value will be negative. Thus, `util.Integer.not(0)` produces the INTEGER value -1 (11111111 11111111 11111111 11111111).
2. The NULL value for the INTEGER type is represented internally with the value 0x80000000 (10000000 00000000 00000000 00000000). When NULL is used with the `util.Integer` bitwise methods, it will be interpreted as 0x80000000 instead of a null value. However, if the result of the bitwise operation produces the value 0x80000000, it will be interpreted as NULL when used in an expression.

Example

```java
IMPORT util
MAIN
    DISPLAY util.Integer.or( 3, 1 ) -- displays 1
END MAIN
```

Related concepts

util.Integer.and on page 2738
Returns the result of a bitwise AND on two INTEGER values.

util.Integer.andNot on page 2739
Returns the result of a bitwise AND of the 1st INTEGER and the inverted 2nd INTEGER.

util.Integer.not on page 2741
Returns the INTEGER value with all bits inverted.

util.Integer.xor on page 2749
Returns the result of a bitwise XOR on two INTEGER values.

`util.Integer.parseHexString`
Returns an INTEGER from its hexadecimal (base 16) string representation.

**Syntax**

```
util.Integer.parseHexString(
    s STRING
)
RETURNS INTEGER
```

1. `s` is the string in hexadecimal.

**Usage**
The `util.Integer.parseHexString()` method scans the hexadecimal string and converts it to an integer value.

**Note:** Regarding A-F hexadecimal letters, the `parseHexString()` method is case-insensitive. The hexadecimal strings "ae3f", "aE3f" and "AE3F" are equivalent.

If the hexadecimal string does not fit in an INTEGER or if it contains an invalid hexadecimal representation, the method raises the numeric conversion error -1213, that can be trapped with a TRY/CATCH or WHENEVER ANY ERROR.

**Note:** Bitwise methods provided by the `util.Integer` class are based on the INTEGER type. Consider the following facts when using these methods:

1. The INTEGER type is a four-byte signed integer: If the bit at position 31 is set to 1, the corresponding INTEGER value will be negative. Thus, `util.Integer.not(0)` produces the INTEGER value -1 (`11111111 11111111 11111111 11111111`).
2. The NULL value for the INTEGER type is represented internally with the value 0x80000000 (`10000000 00000000 00000000 00000000`). When NULL is used with the `util.Integer` bitwise methods, it will be interpreted as 0x80000000 instead of a null value. However, if the result of the bitwise operation produces the value 0x80000000, it will be interpreted as NULL when used in an expression.

**Example**

``` Nim
IMPORT util
MAIN
  DISPLAY util.Integer.parseHexString("6F12") -- displays 28434
END MAIN
```

**Related concepts**

- `util.Integer.toHexString` on page 2747
  Returns the string representation of an INTEGER, as an unsigned integer in base 16.

- `util.Integer.parseBinaryString`
  Returns an INTEGER from its binary (base 2) string representation

**Syntax**

```
util.Integer.parseBinaryString(
    s STRING
)
RETURNS INTEGER
```

1. `s` is the string in binary format (0 and 1).
Usage
The `util.Integer.parseBinaryString()` method scans the string as a set of 0 and 1 digits, and converts it to an integer value.

If the binary representation does not fit in an `INTEGER` or if it contains an invalid binary representation, the method raises the numeric conversion error `-1213`, that can be trapped with a `TRY/CATCH` or `WHENEVER ANY ERROR`.

Note: Bitwise methods provided by the `util.Integer` class are based on the `INTEGER` type. Consider the following facts when using these methods:

1. The `INTEGER` type is a four-byte signed integer: If the bit at position 31 is set to 1, the corresponding `INTEGER` value will be negative. Thus, `util.Integer.not(0)` produces the `INTEGER` value `-1(11111111 11111111 11111111 11111111)`.
2. The `NULL` value for the `INTEGER` type is represented internally with the value `0x80000000` (`10000000 00000000 00000000 00000000`). When `NULL` is used with the `util.Integer` bitwise methods, it will be interpreted as `0x80000000` instead of a null value. However, if the result of the bitwise operation produces the value `0x80000000`, it will be interpreted as `NULL` when used in an expression.

Example

```
IMPORT util
MAIN
    DISPLAY util.Integer.parseBinaryString( "0010" ) -- displays 2
END MAIN
```

Related concepts
`util.Integer.toBinaryString` on page 2748
Returns the string representation of an `INTEGER`, as an unsigned integer in base 2.

`util.Integer.setBit`
Returns the `INTEGER` parameter with the bit at the designated position set to 1.

Syntax

```
util.Integer.setBit(  
    i INTEGER,  
    n INTEGER  
)  
RETURNS INTEGER
```

1. `i` is the integer value to modify.
2. `n` is the bit position (LSB is at position zero).

Usage

The `util.Integer.setBit()` method modifies the integer value by setting the bit at the position passed as second parameter.

Note: The position of the least significant bit (LSB) is the position zero:

- In `00000001` (integer 1), the bit set to 1 is at position 0.
- In `00100010` (integer 34), the bits set to 1 are at position 1 and 5.

Note: Bitwise methods provided by the `util.Integer` class are based on the `INTEGER` type. Consider the following facts when using these methods:

1. The `INTEGER` type is a four-byte signed integer: If the bit at position 31 is set to 1, the corresponding `INTEGER` value will be negative. Thus, `util.Integer.not(0)` produces the `INTEGER` value `-1(11111111 11111111 11111111 11111111)`. 
2. The NULL value for the INTEGER type is represented internally with the value 0x80000000 (10000000 00000000 00000000 00000000). When NULL is used with the util.Integer bitwise methods, it will be interpreted as 0x80000000 instead of a null value. However, if the result of the bitwise operation produces the value 0x80000000, it will be interpreted as NULL when used in an expression.

Example

```
IMPORT util
MAIN
    DISPLAY util.Integer.setBit( 0, 0 ) -- displays 1 (00000001)
    DISPLAY util.Integer.setBit( 1, 0 ) -- displays 1 (00000001)
    DISPLAY util.Integer.setBit( 16, 2 ) -- displays 20 (00010100)
END MAIN
```

Related concepts

util.Integer.clearBit on page 2740
Returns the INTEGER parameter with the bit at the designated position set to 0.

util.Integer.testBit on page 2747
Returns TRUE, if in the INTEGER value, the bit at the designated position is set.

util.Integer.shiftLeft
Returns the INTEGER value left-shifted by the given bit places.

Syntax

```
util.Integer.shiftLeft(
    i INTEGER,
    n SMALLINT
) RETURNS INTEGER
```

1. i is the integer value to shift.
2. n is the left-shift distance in bits.

Usage

The util.Integer.shiftLeft() method shifts the bits to the left in the integer value passed in the first parameter by the number places specified by the second parameter.

```
i   : 210 (00000000 11010010)
n   : 3
--------------------------------
result :1680 (00000110 10010000)
```

Bits shifted out from the left end are lost.

Bits shifted in from the right end are set to 0.

Note: Bitwise methods provided by the util.Integer class are based on the INTEGER type. Consider the following facts when using these methods:

1. The INTEGER type is a four-byte signed integer: If the bit at position 31 is set to 1, the corresponding INTEGER value will be negative. Thus, util.Integer.not(0) produces the INTEGER value -1 (11111111 11111111 11111111 11111111).
2. The NULL value for the INTEGER type is represented internally with the value 0x80000000 (10000000 00000000 00000000 00000000). When NULL is used with the util.Integer bitwise methods, it will be interpreted as 0x80000000 instead of a null value. However, if the result of the bitwise operation produces the value 0x80000000, it will be interpreted as NULL when used in an expression.
Example

IMPORT util
MAIN
   DISPLAY util.Integer.shiftLeft(
       util.Integer.parseBinaryString("11"),
       3 ) -- displays 24
END MAIN

util.Integer.shiftRight
Returns the INTEGER value right-shifted by the given bit places.

Syntax

util.Integer.shiftRight(
   i INTEGER,
   n SMALLINT
) RETURNS INTEGER

1. i is the integer value to shift.
2. n is the right-shift distance in bits.

Usage

The util.Integer.shiftRight() method shifts the bits to the right in the integer value passed in the first parameter by the number of places specified by the second parameter.

<table>
<thead>
<tr>
<th>i</th>
<th>210 (00000000 11010010)</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>3</td>
</tr>
<tr>
<td>result</td>
<td>26 (00000000 00011010)</td>
</tr>
</tbody>
</table>

Bits shifted out from the right end are lost.
Bits shifted in from the left end are set to 0.

Note: Bitwise methods provided by the util.Integer class are based on the INTEGER type. Consider the following facts when using these methods:

1. The INTEGER type is a four-byte signed integer: If the bit at position 31 is set to 1, the corresponding INTEGER value will be negative. Thus, util.Integer.not(0) produces the INTEGER value -1 (11111111 11111111 11111111 11111111).
2. The NULL value for the INTEGER type is represented internally with the value 0x80000000 (10000000 00000000 00000000 00000000). When NULL is used with the util.Integer bitwise methods, it will be interpreted as 0x80000000 instead of a null value. However, if the result of the bitwise operation produces the value 0x80000000, it will be interpreted as NULL when used in an expression.

Example

IMPORT util
MAIN
   DISPLAY util.Integer.shiftRight(
       util.Integer.parseBinaryString("1100"),
       2 ) -- displays 3
END MAIN
**util.Integer.testBit**

Returns TRUE, if in the INTEGER value, the bit at the designed position is set.

**Syntax**

```java
util.Integer.testBit(
    i INTEGER
    n INTEGER
)
RETURNS BOOLEAN
```

1. *i* is an integer value to check.
2. *n* is the bit position (LSB is at position zero).

**Usage**

The `util.Integer.clearBit()` method returns TRUE if the passed integer value has the bit set to 1 at the specified position.

**Note:** The position of the least significant bit (LSB) is the position zero:

- In **00000001** (integer 1), the bit set to 1 is at position 0.
- In **00100010** (integer 34), the bits set to 1 are at position 1 and 5.

**Note:** Bitwise methods provided by the `util.Integer` class are based on the INTEGER type. Consider the following facts when using these methods:

1. The INTEGER type is a four-byte signed integer: If the bit at position 31 is set to 1, the corresponding INTEGER value will be negative. Thus, `util.Integer.not(0)` produces the INTEGER value `-1 (11111111 11111111 11111111 11111111)`.
2. The NULL value for the INTEGER type is represented internally with the value **0x80000000** (**10000000 00000000 00000000 00000000**). When NULL is used with the `util.Integer` bitwise methods, it will be interpreted as **0x80000000** instead of a null value. However, if the result of the bitwise operation produces the value **0x80000000**, it will be interpreted as NULL when used in an expression.

**Example**

```java
IMPORT util
MAIN
    DISPLAY util.Integer.testBit( 3, 1 ) -- displays 1 (TRUE)
END MAIN
```

**Related concepts**

- `util.Integer.clearBit` on page 2740
  Returns the INTEGER parameter with the bit at the designated position set to 0.

- `util.Integer.setBit` on page 2744
  Returns the INTEGER parameter with the bit at the designated position set to 1.

- `util.Integer.toHexString`
  Returns the string representation of an INTEGER, as an unsigned integer in base 16.

**Syntax**

```java
util.Integer.toHexString(
    i INTEGER
)
RETURNS STRING
```

1. *i* is the source integer value to convert.
Usage

The `util.Integer.toHexString()` method generates the hexadecimal representation of the integer passed as parameter.

The resulting hexadecimal string represents the unsigned integer, in base 16.

**Note:** The result has no leading zeros.

**Note:** Bitwise methods provided by the `util.Integer` class are based on the `INTEGER` type. Consider the following facts when using these methods:

1. The `INTEGER` type is a four-byte signed integer: If the bit at position 31 is set to 1, the corresponding `INTEGER` value will be negative. Thus, `util.Integer.not(0)` produces the `INTEGER` value `-1 (11111111 11111111 11111111 11111111).

2. The NULL value for the `INTEGER` type is represented internally with the value `0x80000000 (10000000 00000000 00000000 00000000)`. When NULL is used with the `util.Integer` bitwise methods, it will be interpreted as `0x80000000` instead of a null value. However, if the result of the bitwise operation produces the value `0x80000000`, it will be interpreted as NULL when used in an expression.

Example

```java
IMPORT util
MAIN
    DISPLAY util.Integer.toHexString( 234 ) -- displays "ea"
END MAIN
```

Related concepts

`util.Integer.parseHexString` on page 2743
Returns an `INTEGER` from its hexadecimal (base 16) string representation.

`util.Integer.toBinaryString`
Returns the string representation of an `INTEGER`, as an unsigned integer in base 2.

Syntax

```java
util.Integer.toBinaryString(  
    i INTEGER  
)  
RETURNS STRING
```

1. `i` is the source integer value to convert.

Usage

The `util.Integer.toBinaryString()` method generates the binary representation of the integer passed as parameter.

The resulting binary string represents the unsigned integer, in base 2.

**Note:** The result has no leading zeros.

**Note:** Bitwise methods provided by the `util.Integer` class are based on the `INTEGER` type. Consider the following facts when using these methods:

1. The `INTEGER` type is a four-byte signed integer: If the bit at position 31 is set to 1, the corresponding `INTEGER` value will be negative. Thus, `util.Integer.not(0)` produces the `INTEGER` value `-1 (11111111 11111111 11111111 11111111).

2. The NULL value for the `INTEGER` type is represented internally with the value `0x80000000 (10000000 00000000 00000000 00000000)`. When NULL is used with the `util.Integer` bitwise methods, it will be interpreted as `0x80000000` instead of a null value. However, if the result of the bitwise operation produces the value `0x80000000`, it will be interpreted as NULL when used in an expression.
Example

```achsen
IMPORT util
MAIN
    DISPLAY util.Integer.toBinaryString( 24 ) -- displays "11000"
END MAIN
```

Related concepts

`util.Integer.parseBinaryString` on page 2743
Returns an INTEGER from its binary (base 2) string representation

`util.Integer.xor`
Returns the result of a bitwise XOR on two INTEGER values.

Syntax

```
util.Integer.xor(
    x INTEGER,
    y INTEGER
)
RETURNS INTEGER
```

1. x is an integer value.
2. y is an integer value.

Usage

The `util.Integer.xor()` method makes a bitwise XOR operation with the integer values passed as parameter:

```
x     : 210 (00000000 00000000 00000000 11010010)
y     : 135 (00000000 00000000 00000000 10000111)
--------------------------------------------------
result :  85 (00000000 00000000 00000000 01010101)
```

Note: Bitwise methods provided by the `util.Integer` class are based on the INTEGER type. Consider the following facts when using these methods:

1. The INTEGER type is a four-byte signed integer: If the bit at position 31 is set to 1, the corresponding INTEGER value will be negative. Thus, `util.Integer.not(0)` produces the INTEGER value -1 (11111111 11111111 11111111 11111111).
2. The NULL value for the INTEGER type is represented internally with the value 0x80000000 (00000000 00000000 00000000 00000000). When NULL is used with the `util.Integer` bitwise methods, it will be interpreted as 0x80000000 instead of a null value. However, if the result of the bitwise operation produces the value 0x80000000, it will be interpreted as NULL when used in an expression.

Example

```achsen
IMPORT util
MAIN
    DISPLAY util.Integer.xor( 6, 2 ) -- displays 4
END MAIN
```

Related concepts

`util.Integer.and` on page 2738
Returns the result of a bitwise AND on two INTEGER values.

`util.Integer.andNot` on page 2739
Returns the result of a bitwise AND of the 1st INTEGER and the inverted 2nd INTEGER.

`util.Integer.not` on page 2741
Returns the INTEGER value with all bits inverted.

util.Integer.or on page 2742

Returns the result of a bitwise OR on two INTEGER values.

The util.Interval class

The util.Interval class provides INTERVAL data-type related utility methods.

This class is provided in the util C-Extension library; To use this class, import the util package with:

```
IMPORT util
```

This class does not have to be instantiated; it provides class methods for the current program.

Related concepts

IMPORT C-Extension on page 481

The IMPORT instruction imports c extension module elements to be used by the current module.

util.Interval methods

Methods for the util.Interval class.

Table 589: Class methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>util.Interval.format (</td>
<td>Formats an interval value based on a specified format.</td>
</tr>
<tr>
<td>t INTERVAL q1 TO q2,</td>
<td></td>
</tr>
<tr>
<td>format STRING</td>
<td></td>
</tr>
<tr>
<td>)</td>
<td>RETURN STRING</td>
</tr>
<tr>
<td>util.Interval.parse (</td>
<td>Converts a string to a DATETIME value based on a specified format.</td>
</tr>
<tr>
<td>s STRING,</td>
<td></td>
</tr>
<tr>
<td>format STRING</td>
<td></td>
</tr>
<tr>
<td>)</td>
<td>RETURN DATETIME q1 TO q2</td>
</tr>
</tbody>
</table>

**util.Interval.format**

Formats an interval value based on a specified format.

**Syntax**

```
util.Interval.format (       
   t INTERVAL q1 TO q2,     
   format STRING             
)                           
   RETURNS STRING
```

1. `t` is the interval value to be formatted.
2. `format` is the format string, as described in Formatting INTERVAL values on page 324.

**Usage**

The util.Interval.format () method formats an INTERVAL value based on the format specification.

The format string must be a combination of place holders such as %Y, %m, %d, as described in Formatting INTERVAL values on page 324.
If the source value is NULL the result will be NULL.

Example

```plaintext
IMPORT util
MAIN
    DEFINE iv INTERVAL DAY(6) TO MINUTE
    LET iv = "-157 11:23"
    DISPLAY util.Interval.format(iv, "%d %H:%M")
END MAIN
```

Related concepts

util.Interval.parse on page 2751
Converts a string to a DATETIME value based on a specified format.

util.Interval.parse
Converts a string to a DATETIME value based on a specified format.

Syntax

```plaintext
util.Interval.parse(
    s STRING,
    format STRING
)  
    RETURNS DATETIME q1 TO q2
```

1. `s` is the source string to be parsed.
2. `format` is the format specification (see Formatting INTERVAL values on page 324).

Usage

The `util.Interval.parse()` method parses a string based on a format specification, to produce an INTERVAL value.

The format specification must be a combination of place holders such as `%Y`, `%m`, `%d`, etc.

The precision of the resulting INTERVAL value depends on the format specification. For example, when using "%Y-%m", the resulting value will be an INTERVAL YEAR TO MONTH.

The method returns NULL, if the source string cannot be converted to an INTERVAL value based on the format specification.

For more details about the supported formats, see Formatting INTERVAL values on page 324.

Example

```plaintext
IMPORT util
MAIN
    DEFINE iv INTERVAL DAY(6) TO FRACTION(5)
    LET iv = util.Interval.parse( "-37467 + 23:45:34.12345", "%d + %H:%M:%S %F5" )
END MAIN
```

Related concepts

util.Interval.format on page 2750
Formats an interval value based on a specified format.

**The util.Strings class**
The `util.Strings` class provides `STRING` data-type related utility methods.

This class is provided in the `util C-Extension` library; To use this class, import the `util` package with:

```
IMPORT util
```

This class does not have to be instantiated; it provides class methods for the current program.

**Related concepts**
`IMPORT C-Extension` on page 481
The `IMPORT` instruction imports C extension module elements to be used by the current module.

**util.Strings methods**
Methods for the `util.Strings` class.

**Table 590: Class methods**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>util.Strings.base64Decode</code> (</td>
<td>Decodes a Base64 encoded string and writes the bytes to a file.</td>
</tr>
<tr>
<td>base64 STRING,</td>
<td></td>
</tr>
<tr>
<td>filename STRING</td>
<td></td>
</tr>
<tr>
<td><code>util.Strings.base64Encode</code> (</td>
<td>Converts the content of a file to a Base64 encoded string.</td>
</tr>
<tr>
<td>filename STRING</td>
<td></td>
</tr>
<tr>
<td><code>util.Strings.base64DecodeToHexString</code></td>
<td>Decodes a base64 encoded string and returns the corresponding hexadecimal string.</td>
</tr>
<tr>
<td>( base64 STRING</td>
<td></td>
</tr>
<tr>
<td><code>util.Strings.base64DecodeToString</code></td>
<td>Decodes a base64 encoded string and returns the corresponding string.</td>
</tr>
<tr>
<td>( base64 STRING</td>
<td></td>
</tr>
<tr>
<td><code>util.Strings.base64EncodeFromHexString</code></td>
<td>Converts the hexadecimal string passed as parameter to a Base64 encoded string.</td>
</tr>
<tr>
<td>( s STRING</td>
<td></td>
</tr>
<tr>
<td><code>util.Strings.base64EncodeFromString</code></td>
<td>Converts the string passed as parameter to a Base64 encoded string.</td>
</tr>
<tr>
<td>( s STRING</td>
<td></td>
</tr>
<tr>
<td><code>util.Strings.urlDecode</code></td>
<td>Converts the URL-encoded string to a string in the current application locale.</td>
</tr>
<tr>
<td>( s STRING</td>
<td></td>
</tr>
<tr>
<td><code>util.Strings.urlEncode</code></td>
<td>Converts a string from the current codeset to a URL-encoded string.</td>
</tr>
<tr>
<td>( source STRING</td>
<td></td>
</tr>
</tbody>
</table>
**util.Strings.base64Decode**
Decodes a Base64 encoded string and writes the bytes to a file.

**Syntax**

```plaintext
util.Strings.base64Decode(
    base64 STRING,
    filename STRING
)
```

1. `base64` is the Base64 encoded string.
2. `filename` is the name of the file to write to.

**Usage**
The `util.Strings.base64Decode()` method converts the Base64 encoded string passed as first parameter, and writes the bytes to file specified as second parameter.

**Example**

```plaintext
IMPORT util
MAIN
    DEFINE base64 STRING
    LET base64 = util.Strings.base64Encode( "picture1.png" )
    DISPLAY base64
    CALL util.Strings.base64Decode( base64, "picture2.png" )
END MAIN
```

**Related concepts**

- **util.Strings.base64Encode** on page 2754
  Converts the content of a file to a Base64 encoded string.

- **util.Strings.base64Encode**
  Converts the content of a file to a Base64 encoded string.

**Syntax**

```plaintext
util.Strings.base64Encode(
    filename STRING
)
```

1. `filename` is the name of the file to read from.

**Usage**
The `util.Strings.base64Encode()` method reads the content of the file passed as parameter, and converts the bytes to a Base64 encoded string.

**Example**

```plaintext
IMPORT util
MAIN
    DISPLAY util.Strings.base64Encode( "picture.png" )
END MAIN
```

**Related concepts**

- **util.Strings.base64Decode** on page 2754
Decodes a Base64 encoded string and writes the bytes to a file.

**util.Strings.base64DecodeToHexString**
Decodes a base64 encoded string and returns the corresponding hexadecimal string.

**Syntax**

```plaintext
util.Strings.base64DecodeToHexString(
    base64 STRING
)  
RETURNS STRING
```

1. `base64` is the Base64 encoded string.

**Usage**

The `util.Strings.base64DecodeToHexString()` method converts the Base64 encoded string passed as parameter to an array of bytes, then it converts the byte array to the hexadecimal representation of this array of bytes, and returns that string.

**Tip:** After converting the Base64 string to an hexadecimal string, use the `util.parseHexString()` method to convert bytes of the hexadecimal string to integers.

**Example**

```plaintext
IMPORT util
MAIN
DEFINE hexa VARCHAR(50)
    DISPLAY util.Strings.base64DecodeToHexString( "AA==" )
    DISPLAY util.Strings.base64DecodeToHexString( "AAAAAA==" )
    DISPLAY util.Strings.base64DecodeToHexString( "AQ==" )
    DISPLAY util.Strings.base64DecodeToHexString( "/=8=" )
    DISPLAY util.Strings.base64DecodeToHexString( "QUJDRA==" )
    LET hexa = util.Strings.base64DecodeToHexString( "QUJDRA==" )
    DISPLAY util.Integer.parseHexString( hexa[1,2] )
    DISPLAY util.Integer.parseHexString( hexa[3,4] )
    DISPLAY util.Integer.parseHexString( hexa[5,6] )
    DISPLAY util.Integer.parseHexString( hexa[7,8] )
END MAIN
```

**Output:**

```
00
00000000
01
ffff
41424344
65
66
67
68
```

**Related concepts**

`util.Strings.base64EncodeFromHexString` on page 2756
Converts the hexadecimal string passed as parameter to a Base64 encoded string.

**util.Strings.base64DecodeToString**
Decodes a base64 encoded string and returns the corresponding string.

**Syntax**

```plaintext
util.Strings.base64DecodeToString(
  base64 STRING
)  
RETURNS STRING
```

1. `base64` is the Base64 encoded string.

**Usage**

The `util.Strings.base64DecodeToString()` method converts the Base64 encoded string passed as parameter to an array of bytes, then it converts the byte array to a string representation in the current locale, and returns that string.

If the Base64 source string contains a sequence of bytes that does not represent a valid character in the current application locale, the function returns NULL.

**Note:** In contrast to `util.Strings.urlDecode()`, the original string is not converted from UTF-8 to the application character encoding: The Base64 source string must represent valid characters in the current application locale.

**Example**

(Character encoding is UTF-8)

```plaintext
IMPORT util
MAIN
  DEFINE base64 STRING
  LET base64 = util.Strings.base64EncodeFromString( "Forêt" )
  DISPLAY base64
  DISPLAY util.Strings.base64DecodeToString( base64 )
END MAIN
```

Output:

```
Rm9yw6p0
Forêt
```

**Related concepts**

- **util.Strings.base64EncodeFromString** on page 2757
  Converts the string passed as parameter to a Base64 encoded string.

- **util.Strings.base64EncodeFromHexString**
  Converts the hexadecimal string passed as parameter to a Base64 encoded string.

**Syntax**

```plaintext
util.Strings.base64EncodeFromHexString(
  s STRING
)  
RETURNS STRING
```

1. `s` is the source hexadecimal string to convert in Base64.
Usage

The `util.Strings.base64EncodeFromHexString()` method first converts the hexadecimal string passed as parameter to an array of bytes, then it converts the array of bytes to a Base64 representation, and returns the resulting Base64 encoded string.

Tip: Use the `util.Integer.toHexString()` method to build an hexadecimal string from a set of integer/byte values, then convert the hexadecimal string to a Base64 representation.

Example

```sql
IMPORT util

DEFINE hexa VARCHAR(50)
DISPLAY util.Strings.base64EncodeFromHexString( "00" )
DISPLAY util.Strings.base64EncodeFromHexString( "0000" )
DISPLAY util.Strings.base64EncodeFromHexString( "FFFF" )
DISPLAY util.Strings.base64EncodeFromHexString( "42EFB3E5" )
LET hexa = util.Integer.toHexString(65)
|       | util.Integer.toHexString(66)
|       | util.Integer.toHexString(67)
DISPLAY hexa
DISPLAY util.Strings.base64EncodeFromHexString( hexa )
END MAIN
```

Output:

```
AA==
AAA=
//8=
Qu+z5Q==
414243
QUJD
```

Related concepts

`util.Strings.base64DecodeToHexString` on page 2755
Decodes a base64 encoded string and returns the corresponding hexadecimal string.

`util.Strings.base64EncodeFromString`
Converts the string passed as parameter to a Base64 encoded string.

Syntax

```sql
util.Strings.base64EncodeFromString(
    s STRING
) RETURNS STRING
```

1. `s` is the source string to convert in Base64.

Usage

The `util.Strings.base64EncodeFromString()` method first converts the string passed as parameter (in the current character encoding of the application) to an array of bytes, then it converts the array of bytes to a Base64 representation, and returns the resulting Base64 encoded string.

Note: In contrast to `util.Strings.urlEncode()`, the original string is not converted from the application locale to UTF-8, before performing the encoding to Base64: The resulting Base64 encoded string will contain byte sequences representing characters in the current application locale.
**Example**

(Characteristic encoding is UTF-8)

```
IMPORT util
MAIN
    DISPLAY util.Strings.base64EncodeFromString("Forêt")
END MAIN
```

Output:

```
Rm9yw6p0
```

**Related concepts**

`util.Strings.base64DecodeToString` on page 2756
Decodes a base64 encoded string and returns the corresponding string.

`util.Strings.urlDecode`
Converts the URL-encoded string to a string in the current application locale.

**Syntax**

```
util.Strings.urlDecode(
    s STRING
)
RETURNS STRING
```

1. *s* is the URL-encoded source string (UTF-8 bytes).

**Usage**

The `util.Strings.urlDecode()` method converts the URL-encoded string passed as parameter to a character string.

The source string must contain ASCII characters and/or %xx hexadecimal representation of UTF-8 encoding bytes.

The decoder is error tolerant:

- Alphabetical characters of a %xx element can be uppercase or lowercase (%b2 = %B2).
- If the source string contains a set of %xx elements that represent a UTF-8 encoded character which is not existing in the current application locale, it will be converted to a ? question mark.
- If the percent character is not followed by two hexadecimal digits, then a "%" is copied to the result string and the decoder continues at the next character.

**Example**

```
IMPORT util
MAIN
    DISPLAY util.Strings.urlDecode("abc%C3%84%E2%82%AC")
END MAIN
```

Output:

```
abcĀ€
```

**Related concepts**

`Defining the application locale` on page 519
This section describes the settings defining the application locale, changing the behavior of the compilers and runtime system.

`util.Strings.urlEncode` on page 2759
Converts a string from the current codeset to a URL-encoded string.

util.Strings.urlEncode
Converts a string from the current codeset to a URL-encoded string.

Syntax

```plaintext
util.Strings.urlEncode(
   source STRING
)
RETURNS STRING
```

1. `source` is the source string to url-encode.

Usage

The `util.Strings.urlEncode()` method converts the character string passed as parameter to a URL-encoded string.

All characters not matching `[-_.~a-zA-Z0-9]` are "percent encoded": Percent-encoding involves converting those characters to UTF-8 and representing its corresponding byte values by a percent sign ("%") and a pair of hexadecimal digits.

Example

```plaintext
IMPORT util
MAIN
   DISPLAY util.Strings.urlEncode("abcĀ")
END MAIN

Output:
abc%C3%84%E2%82%AC
```

Related concepts

Defining the application locale on page 519
This section describes the settings defining the application locale, changing the behavior of the compilers and runtime system.

util.Strings.urlDecode on page 2758
Converts the URL-encoded string to a string in the current application locale.

The util.Math class
The `util.Math` class provides basic mathematical functions based on floating point numbers (FLOAT).

This class is provided in the `util C-Extension` library; To use this class, import the `util` package with:

```plaintext
IMPORT util
```

This class does not have to be instantiated; it provides class methods for the current program.

Related concepts

IMPORT C-Extension on page 481
The IMPORT instruction imports C extension module elements to be used by the current module.

**util.Math methods**
Methods for the *util.Math* class.
### Table 591: Class methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>util.Math.acos</code> (x FLOAT)</td>
<td>Computes the arc cosine of the passed value, measured in radians.</td>
</tr>
<tr>
<td><code>util.Math.asin</code> (x FLOAT)</td>
<td>Computes the arc sine of the passed value, measured in radians.</td>
</tr>
<tr>
<td><code>util.Math.atan</code> (x FLOAT)</td>
<td>Computes the arc tangent of the passed value, measured in radians.</td>
</tr>
<tr>
<td><code>util.Math.cos</code> (x FLOAT)</td>
<td>Computes the cosine of the passed value, measured in radians.</td>
</tr>
<tr>
<td><code>util.Math.exp</code> (x FLOAT)</td>
<td>Computes the base-e exponential of the value passed as parameter.</td>
</tr>
<tr>
<td><code>util.Math.log</code> (x FLOAT)</td>
<td>Computes the natural logarithm of the passed value.</td>
</tr>
<tr>
<td><code>util.Math.pi</code>()</td>
<td>Returns the FLOAT value of PI.</td>
</tr>
<tr>
<td><code>util.Math.pow</code> (x FLOAT, y FLOAT)</td>
<td>Computes the value of $x$ raised to the power $y$.</td>
</tr>
<tr>
<td><code>util.Math.rand</code> (max INTEGER)</td>
<td>Returns a positive pseudo-random number.</td>
</tr>
<tr>
<td><code>util.Math.sin</code> (x FLOAT)</td>
<td>Computes the sine of the passed value, measured in radians.</td>
</tr>
<tr>
<td><code>util.Math.sqrt</code> (x FLOAT)</td>
<td>Returns the square root of the argument provided.</td>
</tr>
<tr>
<td><code>util.Math.srand</code>()</td>
<td>Initializes the pseudo-random numbers generator.</td>
</tr>
<tr>
<td><code>util.Math.tan</code> (x FLOAT)</td>
<td>Computes the tangent of the passed value, measured in radians.</td>
</tr>
</tbody>
</table>
**util.Math.acos**
Computes the arc cosine of the passed value, measured in radians.

**Syntax**

```
util.Math.acos(
    x FLOAT )
RETURNS FLOAT
```

1. `x` is a floating point value.

**Usage**

Returns NULL if the argument provided is invalid.

**util.Math.asin**
Computes the arc sine of the passed value, measured in radians.

**Syntax**

```
util.Math.asin(
    x FLOAT )
RETURNS FLOAT
```

1. `x` is a floating point value.

**Usage**

Returns NULL if the argument provided is invalid.

**util.Math.atan**
Computes the arc tangent of the passed value, measured in radians.

**Syntax**

```
util.Math.atan(
    x FLOAT )
RETURNS FLOAT
```

1. `x` is a floating point value.

**Usage**

Returns NULL if the argument provided is invalid.

**util.Math.cos**
Computes the cosine of the passed value, measured in radians.

**Syntax**

```
util.Math.cos(
    x FLOAT )
RETURNS FLOAT
```

1. `x` is a floating point value.

**Usage:**

Returns NULL if the argument provided is invalid.
**util.Math.exp**
Computes the base-e exponential of the value passed as parameter.

**Syntax**
```
util.Math.exp(
   x FLOAT )
RETURNS FLOAT
```

1. *x* is a floating point value.

**Usage**
Returns **NULL** if the argument provided on error.

**util.Math.pi**
Returns the FLOAT value of PI.

**Syntax**
```
util.Math.pi()
RETURNS FLOAT
```

**util.Math.pow**
Computes the value of *x* raised to the power *y*.

**Syntax**
```
util.Math.pow(
   x FLOAT,
   y FLOAT )
RETURNS FLOAT
```

1. *x* is the value to be raised.
2. *y* is the power operand.

**Usage**
The function returns **NULL** if one of the arguments provided is invalid.
If *x* is negative, the caller must ensure that *y* is an integer value.

**util.Math.rand**
Returns a positive pseudo-random number.

**Syntax**
```
util.Math.rand(
   max INTEGER )
RETURNS INTEGER
```

1. *max* is the maximum random number that can be generated.

**Usage**
The `rand()` function returns a pseudo-random integer number between zero and *max*.

**Important:**
The `srand()` function initializes the pseudo-random numbers generator. It must be called before subsequent calls to the `rand()` function. If you do not call the `srand()` function, the `rand()` function will generate the same sequence of numbers for every program execution. The numbers generated by `rand()` can vary depending on the operating system.

The maximum random number returned by the `rand()` function is 2,147,483,646.

The `rand()` function returns zero if the argument is lower or equal to 0.

**Example**

```plaintext
IMPORT util
MAIN
  DEFINE i SMALLINT
  DISPLAY "Before srand() call:"
  FOR i=1 TO 3
    DISPLAY util.Math.rand(100)
  END FOR
  CALL util.Math.srand()
  DISPLAY "After srand() call:"
  FOR i=1 TO 3
    DISPLAY util.Math.rand(100)
  END FOR
END MAIN
```

(run this example several times)

**util.Math.sin**

Computes the sine of the passed value, measured in radians.

**Syntax**

```plaintext
util.Math.sin(  
  x FLOAT )  
RETURNS FLOAT
```

1. `x` is a floating point value.

**Usage**

Returns NULL if the argument provided is invalid.

**util.Math.sqrt**

Returns the square root of the argument provided.

**Syntax**

```plaintext
util.Math.sqrt(  
  x FLOAT )  
RETURNS FLOAT
```

1. `x` is a floating point value.

**Usage**

The function returns NULL if the argument provided is invalid.
**util.Math.srand**
Initializes the pseudo-random numbers generator.

**Syntax**

```javascript
util.Math.srand()
```

**Usage**
The `srand()` function initializes the pseudo-random numbers generator. It must be called before subsequent calls to the `rand()` function. If you do not call the `srand()` function, the `rand()` function will generate the same sequence of numbers for every program execution. The numbers generated by `rand()` can vary depending on the operating system.

**util.Math.tan**
Computes the tangent of the passed value, measured in radians.

**Syntax**

```javascript
util.Math.tan(
    x FLOAT
) RETURNS FLOAT
```

1. `x` is a floating point value.

**Usage**
Returns NULL if the argument provided is invalid.

**util.Math.log**
Computes the natural logarithm of the passed value.

**Syntax**

```javascript
util.Math.log(
    x FLOAT
) RETURNS FLOAT
```

1. `x` is a floating point value.

**Usage**
Returns NULL if the argument provided is invalid.

**util.Math.toDegrees**
Converts an angle measured in radians to an approximately equivalent angle measured in degrees.

**Syntax**

```javascript
util.Math.toDegrees(
    x FLOAT
) RETURNS FLOAT
```

1. `x` is a floating point value to be converted to degrees.
**util.Math.toRadians**
Converts an angle measured in degrees to an approximately equivalent angle measured in radians.

**Syntax**

```plaintext
util.Math.toRadians(
    x  FLOAT )
RETURNS FLOAT
```

1. `x` is a floating point value to be converted to radians.

**The util.JSON class**
The `util.JSON` class provides a basic interface to convert program variable values to/from JSON data.

The `util.JSON` class is provided in the `util C-Extension` library. To use the `util.JSON` class, you must import the `util` package in your program:

```plaintext
IMPORT util
```

This class does not have to be instantiated; it provides class methods for the current program.

The purpose of the `util.JSON` class is to convert a JSON string from/to a BDL variable, to interface with other software based on the JSON format.

The BDL variable can be:

- a simple variable (defined with a primitive type such as `DATE`, `INTEGER`, `VARCHAR(20)`)  
- a structured variable (`RECORD ... END RECORD`) 
- a dynamic array (`DYNAMIC ARRAY OF ...`) 
- a dictionary (`DICTIONARY OF ...`) 

It is not possible to modify JSON elements with this class. In order to manipulate JSON objects, use the `util.JSONObject` and `util.JSONArray` classes.

**Related concepts**

- [JSON support on page 575](#)
  Genero BDL supports JSON data manipulation.

- [IMPORT C-Extension on page 481](#)
The IMPORT instruction imports c extension module elements to be used by the current module.

**util.JSON methods**
Methods for the util.JSON class.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>util.JSON.format (s STRING )</code>&lt;br&gt;RETURNS STRING</td>
<td>Formats a JSON string with indentation.</td>
</tr>
<tr>
<td>`util.JSON.parse (s STRING, variableRef { primitive-type</td>
<td>record-type</td>
</tr>
<tr>
<td><code>util.JSON.proposeType (s STRING )</code>&lt;br&gt;RETURNS STRING</td>
<td>Describes the record structure that can hold a given JSON data string.</td>
</tr>
<tr>
<td>`util.JSON.stringify (value { primitive-type</td>
<td>record-type</td>
</tr>
<tr>
<td>`util.JSON.stringifyOmitNulls (value { primitive-type</td>
<td>record-type</td>
</tr>
</tbody>
</table>

**util.JSON.format**
Formats a JSON string with indentation.

**Syntax**

```plaintext
util.JSON.format (s STRING )
RETURNS STRING
```

1. `s` is a string value that contains JSON formatted data.
Usage

The `util.JSON.format()` class method takes a JSON string as parameter and reorganizes the JSON data for better readability.

If the provided string is not valid JSON, the `format()` method will raise error `-8109`. Consider enclosing the `format()` method call in a `TRY/CATCH` block, if the source string can be malformed JSON.

The main purpose of this method is to beautify a JSON data string that is on a single line, by adding line breaks and indentation.

Example

```plaintext
IMPORT util
MAIN
    DISPLAY "1: ", util.JSON.format('{}')
    DISPLAY "2: ", util.JSON.format({'pkey':8374,"name":"John"}')
    DISPLAY "3: ", util.JSON.format({'ids':[234,3452,9845] })
    DISPLAY "4: ", util.JSON.format({'pkey':8374,"orders":[{}]}')
END MAIN
```

Output:

```
1: {}
2: {
    "pkey": 8374,
    "name": "John"
}
3: {
    "ids": [
        234,
        3452,
        9845
    ]
}
4: {
    "pkey": 8374,
    "orders": []
}
```

Related concepts

Records on page 408
Records allow structured program variables definitions.

`util.JSON.parse`

Parses a JSON string and fills program variables with the values.

Syntax

```plaintext
util.JSON.parse(
    s STRING,
    variableRef | primitive-type
                  | record-type
                  | array-type
                  | dictionary-type
)
```

1. `s` is a string value that contains JSON formatted data.
2. `variableRef` is the variable to be initialized with values of the JSON string.

   **Important:** The `variableRef` parameter is passed by reference to the method.
3. *primitive-type* is a primitive data type of Genero (INTEGER, DATE, VARCHAR)
4. *record-type* is a RECORD ... END RECORD type.
5. *array-type* is a DYNAMIC ARRAY OF ... or ARRAY[n] OF ... type.
6. *dictionary-type* is a DICTIONARY OF ... type.

**Usage**

The `util.JSON.parse()` class method scans the JSON source string passed as parameter and fills the destination variable members by name.

If the provided string is not valid JSON, the `parse()` method will raise error -8109. Consider enclosing the `parse()` method call in a TRY/CATCH block, if the source string can be malformed JSON.

The destination variable is expected to have the same structure as the JSON source data, it can be a RECORD, DYNAMIC ARRAY or a DICTIONARY.

The `parse()` method initializes the target variable to NULL before the parsing starts.

See [JSON support](#) on page 575 for details on how the destination variable is populated when the structures are not identical.

**Note:** When parsing a JSON string to fill a TEXT or BYTE variable, if the data storage for the LOB variable has not been defined with the LOCATE instruction, the JSON methods will automatically locate the TEXT or BYTE in memory. This applies also to TEXT and BYTE elements of records, arrays and dictionaries.

**Example**

```plaintext
IMPORT util
MAIN
  DEFINE cust_rec RECORD
    cust_num INTEGER,
    cust_name VARCHAR(30),
    order_ids DYNAMIC ARRAY OF INTEGER
  END RECORD
  DEFINE js STRING
  LET js='{ "cust_num":2735, "cust_name":"McCarlson",
         "order_ids":[234,3456,24656,34561] }'
  TRY
    CALL util.JSON.parse( js, cust_rec )
    DISPLAY cust_rec.cust_name
    DISPLAY cust_rec.order_ids[4]
  CATCH
    DISPLAY "ERROR:", STATUS
  END TRY
END MAIN
```

**Related concepts**

*Records* on page 408
Records allow structured program variables definitions.

*util.JSON.proposeType*
Describes the record structure that can hold a given JSON data string.

**Syntax**

```plaintext
util.JSON.proposeType(
  s STRING)
RETURNS STRING
```

1. *s* is a string value that contains JSON formatted data.
Usage

The `util.JSON.proposeType()` class method takes a JSON formatted string as parameter and generates the RECORD definition that can be used as base record definition hold the source JSON string.

If the provided string is not valid JSON, the `proposeType()` method will raise error -8109. Consider enclosing the `proposeType()` method call in a TRY/CATCH block, if the source string can be malformed JSON.

In order to respect the BDL syntax, and match exactly the JSON source structure, the type generated by `proposeType()` may need to be reviewed.

For example, if the JSON property name contains characters not allowed in a BDL identifiers, use the `json_name` variable definition attribute to map the JSON property name to the RECORD element. Furthermore, the type DICTIONARY will not be proposed. As result, a source JSON string like ("First Name": "John", "Last Name": "Piper") will produce:

```plaintext
RECORD
   First Name STRING,
   Last Name STRING
END RECORD
```

That needs to be adapted to use `json_name` attributes:

```plaintext
RECORD
   first_name STRING ATTRIBUTES(json_name="First Name"),
   last_name  STRING ATTRIBUTES(json_name="Last Name")
END RECORD
```

Or to a DICTIONARY type:

```plaintext
DICTIONARY OF STRING
```

Example

```plaintext
IMPORT util
MAIN
   DEFINE js STRING
   LET js='{ "cust_num":2735, "cust_name":"McCarlson",
           "orderids":\[234,3456,24656,34561\] }'
   DISPLAY util.JSON.proposeType( js )
END MAIN
```

Displays:

```plaintext
RECORD
   cust_num FLOAT,
   cust_name STRING,
   orderids DYNAMIC ARRAY OF FLOAT
END RECORD
```

Related concepts

Records on page 408
Records allow structured program variables definitions.

`util.JSON.stringify`
Transforms a record variable to a flat JSON formatted string, by including empty records and empty arrays.

Syntax

```plaintext
util.JSON.stringify()
```
value | $ primitive-type 
| record-type 
| array-type 
| dictionary-type 

) 

RETURNS STRING

1. value is the program variable to be converted to a JSON string.
2. primitive-type is a primitive data type of Genero (INTEGER, DATE, VARCHAR)
3. record-type is a RECORD ... END RECORD type.
4. array-type is a DYNAMIC ARRAY OF ... or ARRAY[n] OF ... type.
5. dictionary-type is a DICTIONARY OF ... type.

Usage

The util.JSON.stringify() class method takes a variable as parameter, and generates the corresponding data string in JSON format, as defined in the [RFC4627] specification.

Important: Unlike util.JSON.stringifyOmitNulls(), empty records (where all members are NULL), and empty arrays will be written in the JSON string. For detailed control on null and empty variables when serializing to JSON elements, use the json_null variable definition attribute.

The method raises error -8110 if the JSON string cannot be generated.

For more details about FGL to JSON conversion, see JSON support on page 575.

Example

IMPORT util
MAIN
DEFINE rec RECORD
   field1 INTEGER,
   field2 CHAR(1),
   subrec1 RECORD
      field11 INTEGER,
      field12 VARCHAR(30)
   END RECORD,
   subarr1 DYNAMIC ARRAY OF INTEGER
END RECORD
INITIALIZE rec.* TO NULL
LET rec.field1 = 999
LET rec.subarr1[3] = 888
DISPLAY "stringify() : ", util.JSON.stringify(rec)
END MAIN

Output:

stringify() : {"field1":999,"subrec1":{},"subarr1":[null,null,888]}

Related concepts

util.JSON.stringifyOmitNulls on page 2772
Transforms a record variable to a flat JSON formatted string, by excluding empty records and empty arrays.

Records on page 408
Records allow structured program variables definitions.

*util.JSON.stringifyOmitNulls*

Transforms a record variable to a flat JSON formatted string, by excluding empty records and empty arrays.

**Syntax**

```plaintext
util.JSON.stringifyOmitNulls(
  value { primitive-type
    | record-type
    | array-type
    | dictionary-type
  }
)
RETURNS STRING
```

1. `value` is the program variable to be converted to a JSON string.
2. `primitive-type` is a primitive data type of Genero (INTEGER, DATE, VARCHAR)
3. `record-type` is a `RECORD ... END RECORD` type.
4. `array-type` is a `DYNAMIC ARRAY OF ...` or `ARRAY[n] OF ...` type.
5. `dictionary-type` is a `DICTIONARY OF ...` type.

**Usage**

The `util.JSON.stringifyOmitNulls()` class method takes a variable as parameter, and generates the corresponding data string in JSON format, as defined in the [RFC4627] specification.

**Important:** Unlike `util.JSON.stringify()`, empty records (where all members are NULL), and empty arrays will **NOT** be written in the JSON string. For detailed control on null and empty variables when serializing to JSON elements, use the `json_null` variable definition attribute and the `stringify()` method. The method `stringifyOmitNulls()` behaves like `stringify()`, when all variable elements are defined with the `json_null="undefined"` attribute.

The method raises error `-8110` if the JSON string cannot be generated.

For more details about FGL to JSON conversion, see JSON support on page 575.

**Example**

```plaintext
IMPORT util
MAIN
  DEFINE rec RECORD
    field1 INTEGER,
    field2 CHAR(1),
    subrec1 RECORD
      field11 INTEGER,
      field12 VARCHAR(30)
    END RECORD,
    subarr1 DYNAMIC ARRAY OF INTEGER
  END RECORD
  INITIALIZE rec.* TO NULL
  LET rec.field1 = 999
  DISPLAY "stringify()   : ", util.JSON.stringify(rec)
  DISPLAY "stringifyOmitNulls(): ", util.JSON.stringifyOmitNulls(rec)
END MAIN
```

Output:

```
stringify()         : {"field1":999,"subrec1":{},"subarr1":[]}
stringifyOmitNulls(): {"field1":999}
```
**Related concepts**
util.JSON.stringify on page 2770
Transforms a record variable to a flat JSON formatted string, by including empty records and empty arrays.

Records on page 408
Records allow structured program variables definitions.

**Examples**
util.JSON usage examples.

*Example 1: Reading a JSON file*

This program reads JSON data from customers.json, then parses the file content to fill the program array custlist, converts the array back to a JSON string and writes a formatted JSON string to the standard output.

**Note:** The parse() method is enclosed in a TRY/CATCH block, to detect potential JSON format errors in the source file.

```c
IMPORT util
MAIN
  DEFINE jsondata TEXT,
  custlist DYNAMIC ARRAY OF RECORD
    num INTEGER,
    name VARCHAR(40)
  END RECORD,
  tmp STRING
  LOCATE jsondata IN FILE "customers.json"
  TRY
    CALL util.JSON.parse( jsondata, custlist )
    DISPLAY "Array length = ", custlist.getLength()
    LET tmp = util.JSON.stringify(custlist)
    DISPLAY util.JSON.format( tmp )
  CATCH
    DISPLAY "ERROR:", STATUS
  END TRY
END MAIN
```

The file customers.json:

```json
[
  {
    "num": 823,
    "name": "Mark Renbing"
  },
  {
    "num": 234,
    "name": "Clark Gambler"
  }
]
```

**Note** that the JSON file does not contain the name of the dynamic array (custlist), but starts directly with the JSON array in [ ] square brackets.

**The util.JSONObject class**
The util.JSONObject class provides methods to handle an structured data object following the JSON string syntax.

The util.JSONObject class is provided in the util C-Extension library. To use the util.JSONObject class, you must import the util package in your program:

```c
IMPORT util
```
A JSONObject is an unordered collection of name/value pairs. The format of a JSON object string is a comma-separated "name":value pairs, wrapped in curly brackets. The value can be simple numeric or string value, but it can also be an array of values enclosed in square brackets, or a sub-element enclosed in curly brackets:

```json
{
  "cust_num":2735,
  "cust_name":"McCarlson",
  "order_ids":[234,3456,24656,34561],
  "address": {
    "street":"34, Sunset Bld",
    "city":"Los Angeles",
    "state":"CA"
  }
}
```

A JSONObject object must be created before usage with one of the class methods like `util.JSONObject.create()`. The JSONObject class provides methods for accessing, adding/replacing or deleting the values by name with the `get()`, `put()` and `remove()` methods.

The `get()` method can return a simple value, a `util.JSONObject` or a `util.JSONArray` object reference. The `put()` method can take a simple value, a RECORD, or an ARRAY as parameter.

If the structure of the JSON object is not known at compile time, you can introspect the elements of the object with the `getLength()`, `getType()` and `name()` methods.

**Related concepts**

**JSON support** on page 575
Genero BDL supports JSON data manipulation.

**IMPORT C-Extension** on page 481
The IMPORT instruction imports C extension module elements to be used by the current module.

**util.JSONObject methods**
Methods for the util.JSONObject class.

**Table 593: Class methods**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>util.JSONObject.create()</code></td>
<td>Creates a new JSON object.</td>
</tr>
<tr>
<td><code>util.JSONObject.fromFGL()</code></td>
<td>Creates a new JSON object from a RECORD.</td>
</tr>
<tr>
<td><code>util.JSONObject.parse()</code></td>
<td>Parses a JSON string and creates a JSON object from it.</td>
</tr>
</tbody>
</table>
### Table 594: Object methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>util.JSONObject.get(name STRING)</code></td>
<td>Returns the value corresponding to the specified entry name.</td>
</tr>
<tr>
<td><code>util.JSONObject.getLength()</code></td>
<td>Returns the number of name-value pairs in the JSON object.</td>
</tr>
<tr>
<td><code>util.JSONObject.getType(name STRING)</code></td>
<td>Returns the type of a JSON object element.</td>
</tr>
<tr>
<td><code>util.JSONObject.has(name STRING)</code></td>
<td>Checks if the JSON object contains a specific entry name.</td>
</tr>
<tr>
<td><code>util.JSONObject.name(index INTEGER)</code></td>
<td>Returns the name of a JSON object entry by position.</td>
</tr>
<tr>
<td><code>util.JSONObject.put(name STRING, value value-type)</code></td>
<td>Sets a name-value pair in the JSON object.</td>
</tr>
<tr>
<td><code>util.JSONObject.remove(name STRING)</code></td>
<td>Removes the specified element in the JSON object.</td>
</tr>
<tr>
<td><code>util.JSONObject.toFGL(recordRef record-type)</code></td>
<td>Fills a record variable with the entries contained in the JSON object.</td>
</tr>
<tr>
<td><code>util.JSONObject.toString()</code></td>
<td>Builds a JSON string from the values contained in the JSON object.</td>
</tr>
</tbody>
</table>

#### `util.JSONObject.create`

Creates a new JSON object.

#### Syntax

```plaintext
util.JSONObject.create()
RETURNS util.JSONObject
```

#### Usage

The `util.JSONObject.create()` method creates a new JSON object.

The created object must be assigned to a program variable defined with the `util.JSONObject` type.
Example

```plaintext
IMPORT util
MAIN
  DEFINE cust_rec RECORD
    cust_num INTEGER,
    cust_name VARCHAR(30),
    order_ids DYNAMIC ARRAY OF INTEGER
  END RECORD
  DEFINE obj util.JSONObject
  LET cust_rec.cust_num = 345
  LET cust_rec.cust_name = "McMaclum"
  LET cust_rec.order_ids[1] = 4732
  LET cust_rec.order_ids[2] = 9834
  LET cust_rec.order_ids[3] = 2194
  LET obj = util.JSONObject.fromFGL(cust_rec)
  DISPLAY obj.toString()
END MAIN
```

Related concepts

util.JSONObject.fromFGL on page 2776
Creates a new JSON object from a RECORD.

util.JSONObject.parse on page 2777
Parses a JSON string and creates a JSON object from it.

util.JSONObject.fromFGL
Creates a new JSON object from a RECORD.

Syntax

```plaintext
util.JSONObject.fromFGL(
  record record-type )
RETURNS util.JSONObject
```

1. `record` is the record variable used to create the JSON object.
2. `record-type` is a RECORD ... END RECORD type.

Usage

The `util.JSONObject.fromFGL()` method creates a new JSON object from the RECORD variable passed as parameter.

The created object must be assigned to a program variable defined with the `util.JSONObject` type.

The members of the RECORD are converted to name/value pairs in the JSON object.

For more details about FGL to JSON conversion, see JSON support on page 575.

Example

```plaintext
IMPORT util
MAIN
  DEFINE cust_rec RECORD
    cust_num INTEGER,
    cust_name VARCHAR(30),
    order_ids DYNAMIC ARRAY OF INTEGER
  END RECORD
  DEFINE obj util.JSONObject
  LET cust_rec.cust_num = 345
  LET cust_rec.cust_name = "McMaclum"
  LET cust_rec.order_ids[1] = 4732
  LET cust_rec.order_ids[2] = 9834
  LET cust_rec.order_ids[3] = 2194
  LET obj = util.JSONObject.fromFGL(cust_rec)
  DISPLAY obj.toString()
END MAIN
```

Related concepts

Records on page 408
Records allow structured program variables definitions.

*util.JSONObject.create* on page 2775
Creates a new JSON object.

*util.JSONObject.parse* on page 2777
Parses a JSON string and creates a JSON object from it.

### Syntax

```c
util.JSONObject.parse(
    s STRING
) RETURNS util.JSONObject
```

1. `s` is a string value that contains JSON formatted data.

### Usage

The `util.JSONObject.parse()` method scans the JSON source string passed as parameter and creates a JSON object from it.

The created object must be assigned to a program variable defined with the `util.JSONObject` type.

The source string must follow the JSON format specification. It can contain multi-level structured data, but it must start with a curly brace.

The method raises error `-8109` if the JSON source string is not properly formatted. Consider enclosing the `parse()` method call in a `TRY/CATCH` block, if the source string can be malformed JSON.

### Example

```c
IMPORT util
MAIN
    DEFINE js STRING
    DEFINE obj util.JSONObject
    LET js='{ "cust_num":2735, "cust_name":"McCarlson", "orderids":[234,3456,24656,34561] }'
    LET obj = util.JSONObject.parse( js )
    DISPLAY obj.get("cust_name")
END MAIN
```

### Related concepts

*util.JSONObject.create* on page 2775
Creates a new JSON object.

*util.JSONObject.fromFGL* on page 2776
Creates a new JSON object from a RECORD.

*util.JSONObject.get*
Returns the value corresponding to the specified entry name.

### Syntax

```c
util.JSONObject.get(
    name STRING
) RETURNS result-type
```

1. `name` is the string identifying the JSON object property.
2. **result-type** can be a simple type, a `util.JSONObject` or a `util.JSONArray` object reference.

**Usage**

The `get()` method returns the value or JSON object corresponding to the element name passed as parameter.

If the element identified by the name is a simple value, the method returns a string. If the element is structured, the method returns a `util.JSONObject` instance and the returned object must be assigned to a program variable defined with the `util.JSONObject` type. If the element is a list of values, the method `util.JSONArray` instance and the returned object must be assigned to a program variable defined with the `util.JSONArray` type.

**Note:** If the element is a simple value, pay attention to the data format when directly assigning the result of the `get()` method to a program variable. For example, if the JSON string value represents a date, it will use the ISO format (YYYY-MM-DD). Assigning this ISO-formatted date string to a `DATE` variable will fail, if `DBDATE` format does not match.

A name/value pair can be set with the `put()` method.

**Example**

```hll
IMPORT util
MAIN
    DEFINE obj, sub util.JSONObject
    DEFINE jarr util.JSONArray
    DEFINE rec RECORD
        id INTEGER,
        name STRING
    END RECORD
    DEFINE arr DYNAMIC ARRAY OF INTEGER
    DEFINE x INT
    LET obj = util.JSONObject.create()
    -- Simple value
    CALL obj.put("simple", 234)
    LET x = obj.get("simple")
    -- Sub-element
    LET rec.id = 234
    LET rec.name = "Barton"
    CALL obj.put("record", rec)
    LET sub = obj.get("record")
    -- Array
    LET arr[1] = 234
    LET arr[2] = 2837
    CALL obj.put("array", arr)
    LET jarr = obj.get("array")
END MAIN
```

**Related concepts**

- **`util.JSONObject.put`** on page 2781
  Sets a name-value pair in the JSON object.

- **`util.JSONObject.getLength`**
  Returns the number of name-value pairs in the JSON object.

**Syntax**

```hll
util.JSONObject.getLength()
RETURNS INTEGER
```

**Usage**

The `getLength()` method returns the number of name-value pairs in the JSON object.
This method can be used along with the `name()` and `getType()` methods to read the entries of a JSON object.

**Example**

```plaintext
IMPORT util
MAIN
    DEFINE obj util.JSONObject
    DEFINE i INTEGER
    LET obj = util.JSONObject.parse('{"id":123,"name":"Scott"}')
    FOR i=1 TO obj.getLength()
        DISPLAY i, " ", obj.name(i), " ", " = ", obj.get(obj.name(i))
    END FOR
END MAIN
```

**Related concepts**

- `util.JSONObject.name` on page 2780
  Returns the name of a JSON object entry by position.
- `util.JSONObject.getType` on page 2779
  Returns the type of a JSON object element.
- `util.JSONObject.getType`
  Returns the type of a JSON object element.

**Syntax**

```plaintext
util.JSONObject.getType(
    name STRING
) RETURNS STRING
```

1. `name` is the name of the element.

**Usage**

The `getType()` method returns the JSON data type name corresponding to the element name passed as parameter. This method can be used along with the `name()` and `getLength()` methods, to read the entries of a JSON object. Possible values returned by this method are:

- **NUMBER**: A numeric value.
- **STRING**: A string value delimited by double quotes.
- **BOOLEAN**: A boolean value (true/false)
- **NULL**: A non-existing element.
- **OBJECT**: A structured object.
- **ARRAY**: An ordered list of elements.

**Example**

```plaintext
IMPORT util
MAIN
    DEFINE obj util.JSONObject
    LET obj = util.JSONObject.create()
    CALL obj.put("id", 8723)
    DISPLAY obj.getType("id") -- NUMBER
    CALL obj.put("name", "Brando")
    DISPLAY obj.getType("name") -- STRING
    DISPLAY obj.getType("undef") -- NULL
END MAIN
```
**Related concepts**

*util.JSONObject.getLength* on page 2778
Returns the number of name-value pairs in the JSON object.

*util.JSONObject.name* on page 2780
Returns the name of a JSON object entry by position.

*util.JSONObject.has*
Checks if the JSON object contains a specific entry name.

**Syntax**

```
util.JSONObject.has(
  name STRING
) RETURNS BOOLEAN
```

1. *name* is a string identifying a JSON object property.

**Usage**

The `has()` method determines if the JSON object holds a property identified by the element name passed as parameter.

The method returns `TRUE` if the name/value pair exists in the JSON object.

A name/value pair can be set with the `put()` method.

**Related concepts**

*util.JSONObject.put* on page 2781
Sets a name-value pair in the JSON object.

*util.JSONObject.name*
Returns the name of a JSON object entry by position.

**Syntax**

```
util.JSONObject.name(
  index INTEGER
) RETURNS STRING
```

1. *index* is the index of the name-value pair in the JSON object.

**Usage**

The `name()` method returns the entry name in the JSON object at the given position.

The index corresponding to the first name-value pair is 1.

If no entry exists at the given index, the method returns `NULL`.

This method can be used along with the `getLength()` and `getType()` methods to read the entries of a JSON object.

**Example**

```
IMPORT util
MAIN
  DEFINE obj util.JSONObject
  DEFINE i INTEGER
  LET obj = util.JSONObject.parse('"id":123,"name":"Scott"')
  FOR i=1 TO obj.getLength()
    DISPLAY i, ": ", obj.name(i)
```
Related concepts

util.JSONObject.getLength on page 2778
Returns the number of name-value pairs in the JSON object.

util.JSONObject.getType on page 2779
Returns the type of a JSON object element.

util.JSONObject.put
Sets a name-value pair in the JSON object.

Syntax

```java
util.JSONObject.put(
    name STRING,
    value value-type
)
```

1. `name` is a string defining the entry name.
2. `value` is the value to be associated to the name.
3. `value-type` can be a simple string or numeric type, a RECORD or a DYNAMIC ARRAY.

Usage

The `put()` method adds a name-value pair to the JSON object.

The first parameter is the name of the element. The second parameter can be a simple string or numeric value, or a complex variable defined as RECORD or DYNAMIC ARRAY.

If the element exists, the existing value is replaced.

Example

```java
IMPORT util
MAIN
    DEFINE obj util.JSONObject
    DEFINE rec RECORD
        id INTEGER,
        name STRING
    END RECORD
    DEFINE arr DYNAMIC ARRAY OF INTEGER
    LET obj = util.JSONObject.create()
    CALL obj.put("simple", 234)
    LET rec.id = 234
    LET rec.name = "Barton"
    CALL obj.put("record", rec)
    LET arr[1] = 234
    LET arr[2] = 2837
    CALL obj.put("array", arr)
    DISPLAY obj.toString()
END MAIN
```

Related concepts

util.JSONObject.get on page 2777
Returns the value corresponding to the specified entry name.

`util.JSONObject.remove`
Removes the specified element in the JSON object.

**Syntax**

```
util.JSONObject.remove(
   name STRING )
```

1. `name` is the string identifying the JSON object property.

**Usage**

The `remove()` method deletes a name-value pair identified by the name passed as parameter.

**Example**

```plaintext
IMPORT util
MAIN
   DEFINE obj util.JSONObject
   LET obj = util.JSONObject.create()
   CALL obj.put("address", "5 Brando Street")
   CALL obj.remove("address")
   DISPLAY obj.get("address")
END MAIN
```

**Related concepts**

- `util.JSONObject.put` on page 2781
  Sets a name-value pair in the JSON object.
- `util.JSONObject.get` on page 2777
  Returns the value corresponding to the specified entry name.
- `util.JSONObject.toFGL`
  Fills a record variable with the entries contained in the JSON object.

**Syntax**

```
util.JSONObject.toFGL(
   recordRef record-type )
```

1. `recordRef` is the variable to be set with values of the JSON string.
   
   **Important:** The `recordRef` is a RECORD variable is passed by reference to the method.

2. `record-type` is a RECORD ... END RECORD type.

**Usage**

The `toFGL()` method fills the RECORD variable passed as parameter with the corresponding values defined in the JSON object.

The destination record must have the same structure as the JSON source data. For more details see JSON support on page 575.

**Example**

```plaintext
IMPORT util
MAIN
   DEFINE cust_rec RECORD
   cust_num INTEGER,
```
cust_name VARCHAR(30),
    order_ids DYNAMIC ARRAY OF INTEGER
END RECORD

DEFINE js STRING
DEFINE obj util.JSONObject
LET js='{"cust_num":2735, "cust_name":"McCarlson",
"order_ids":[234,3456,24656,34561]}'
LET obj = util.JSONObject.parse( js )
CALL obj.toFGL( cust_rec )
DISPLAY cust_rec.cust_name
DISPLAY cust_rec.order_ids[4]
END MAIN

Related concepts

util.JSONObject.toString on page 2783
Builds a JSON string from the values contained in the JSON object.

util.JSONObject.toString
Builds a JSON string from the values contained in the JSON object.

**Syntax**

```plaintext
util.JSONObject.toString()
RETURNS STRING
```

**Usage**

The `toString()` method produces a JSON formatted string from the name-value pairs contained in the JSON object.

**Example**

```plaintext
IMPORT util
MAIN
    DEFINE obj util.JSONObject
    LET obj = util.JSONObject.create()
    CALL obj.put("num", "75263")
    CALL obj.put("name", "Ferguson")
    CALL obj.put("address", "12 Marylon Street")
    DISPLAY obj.toString()
END MAIN
```

Related concepts

util.JSONObject.toFGL on page 2782
Fills a record variable with the entries contained in the JSON object.

The **util.JSONArray class**

The `util.JSONArray` class provides methods to handle an array of values, following the JSON string syntax.

The `util.JSONArray` class is provided in the `util` C-Extension library. To use the `util.JSONArray` class, you must import the `util` package in your program:

```plaintext
IMPORT util
```

A JSONArray is a sequence of unnamed values. The format of a JSON array string is a list of values wrapped in square brackets with commas between the values:

```plaintext
[123, 546, "abc", "def", "xyz"]
```
A JSONArray object must be created before usage with one of the class methods like
util.JSONArray.create().

The JSONArray class provides methods for accessing, adding/replacing or deleting the array values by index with
the get(), put() and remove() methods.

If the structure of the JSON array is not known at compile time, you can introspect the elements of the array with the
getLength() and getType() methods.

Related concepts
JSON support on page 575
Genero BDL supports JSON data manipulation.

IMPORT C-Extension on page 481
The IMPORT instruction imports C extension module elements to be used by the current module.

util.JSONArray methods
Methods for the util.JSONArray class.

Table 595: Class methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>util.JSONArray.create()</td>
<td>Creates a new JSON array object.</td>
</tr>
<tr>
<td>RETURNS util.JSONArray</td>
<td></td>
</tr>
<tr>
<td>util.JSONArray.fromFGL(</td>
<td>Creates a new JSON array object from a DYNAMIC</td>
</tr>
<tr>
<td>array dynamic-array-type )</td>
<td>ARRAY.</td>
</tr>
<tr>
<td>RETURNS util.JSONArray</td>
<td></td>
</tr>
<tr>
<td>util.JSONArray.parse(</td>
<td>Parses a JSON string and creates a JSON array</td>
</tr>
<tr>
<td>s STRING )</td>
<td>object from it.</td>
</tr>
<tr>
<td>RETURNS util.JSONArray</td>
<td></td>
</tr>
</tbody>
</table>
Table 596: Object methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>util.JSONArray.get (index INTEGER) RETURNS result-type</td>
<td>Returns the value of a JSON array element.</td>
</tr>
<tr>
<td>util.JSONArray.getLength () RETURNS INTEGER</td>
<td>Returns the number of elements in the JSON array object.</td>
</tr>
<tr>
<td>util.JSONArray.getType (index INTEGER) RETURNS STRING</td>
<td>Returns the type of a JSON array element.</td>
</tr>
<tr>
<td>util.JSONArray.put (index INTEGER, value value-type)</td>
<td>Sets an element by position in the JSON array object.</td>
</tr>
<tr>
<td>util.JSONArray.remove (index INTEGER)</td>
<td>Removes the specified entry in the JSON array object.</td>
</tr>
<tr>
<td>util.JSONArray.toFGL (arrayRef dynamic-array-type)</td>
<td>Fills a dynamic array variable with the elements contained in the JSON array object.</td>
</tr>
<tr>
<td>util.JSONArray.toString () RETURNS STRING</td>
<td>Builds a JSON string from the elements contained in the JSON array object.</td>
</tr>
</tbody>
</table>

util.JSONArray.create

Creates a new JSON array object.

Syntax

```plaintext
util.JSONArray.create ()
RETURNS util.JSONArray
```

Usage

The `util.JSONArray.create ()` method create a new JSON array object.

The created object must be assigned to a program variable defined with the `util.JSONArray` type.

Example

```plaintext
IMPORT util
MAIN
  DEFINE arr util.JSONArray
  LET arr = util.JSONArray.create ()
  ...
END MAIN
```
**Related concepts**

- `util.JSONArray.fromFGL` on page 2786
  Creates a new JSON array object from a **DYNAMIC ARRAY**.

- `util.JSONArray.parse` on page 2786
  Parses a JSON string and creates a JSON array object from it.

**util.JSONArray.fromFGL**

Creates a new JSON array object from a **DYNAMIC ARRAY**.

**Syntax**

```plaintext
util.JSONArray.fromFGL(
    array dynamic-array-type )
RETURNS util.JSONArray
```

1. `array` is the dynamic array variable used to create the JSON array object.
2. `dynamic-array-type` is a **DYNAMIC ARRAY OF ...** type.

**Usage**

The `util.JSONArray.fromFGL()` method creates a new JSON array from the **DYNAMIC ARRAY** variable passed as parameter.

The created object must be assigned to a program variable defined with the `util.JSONArray` type.

The members of the **DYNAMIC ARRAY** are converted to a list of name / value pairs in the JSON array object.

The dynamic array can be structured with a **RECORD** definition: the elements of the array will be converted individually.

For more details about FGL to JSON conversion, see **JSON support** on page 575.

**Example**

```plaintext
IMPORT util
MAIN
    DEFINE da DYNAMIC ARRAY OF INTEGER
    DEFINE arr util.JSONArray
    LET da[1] = 123
    LET da[2] = 972
    LET arr = util.JSONArray.fromFGL(da)
    DISPLAY arr.toString()
END MAIN
```

**Related concepts**

- **Arrays** on page 418
  Arrays (static or dynamic) allow you to handle an ordered collection of elements.

- `util.JSONArray.parse`  
Parses a JSON string and creates a JSON array object from it.

**Syntax**

```plaintext
util.JSONArray.parse(
    s STRING )
RETURNS util.JSONArray
```

1. `s` is a string value that contains JSON formatted data as a list of elements delimited by square brackets.
Usage

The `util.JSONArray.parse()` method scans the JSON source string passed as parameter and creates a new
JSON array object from it.

The created object must be assigned to a program variable defined with the `util.JSONArray` type.

The source string must follow the JSON format specification. Elements of the list can contain multi-level structured
data, but the string must follow the JSON array string syntax `[ element, ... ]` with square brackets.

The method raises error -8109 if the JSON source string is not properly formatted. Consider enclosing the `parse()` method call in a `TRY/CATCH` block, if the source string can be malformed JSON.

Example

```import util
main
  define da dynamic array of integer
  define arr util.JSONArray
  let arr = util.JSONArray.parse("[1,2,3,4,5]"
  display arr.toString()
end main```

Related concepts

`util.JSONArray.fromFGL` on page 2786
Creates a new JSON array object from a `DYNAMIC ARRAY`.

`util.JSONArray.create` on page 2785
Creates a new JSON array object.

`util JSONArray.get`
Returns the value of a JSON array element.

Syntax

```util JSONArray.get(
  index integer )
returns result-type```

1. `index` is the index of the element in the JSON array object.
2. `result-type` can be a simple type, a `util.JSONObject` or a `util.JSONArray` object reference.

Usage

The `get()` method returns the value or JSON object corresponding to the element at the given position.

The index corresponding to the first element is 1.

If no element exists at the given index, the method returns `NULL`.

If the element identified by the name is a simple value, the method returns a string. If the element is structured, the
method returns a `util.JSONObject` instance and the returned object must be assigned to a program variable
defined with the `util.JSONObject` type. If the element is a list of values, the method `util.JSONArray`
instance and the returned object must be assigned to a program variable defined with the `util.JSONArray` type.

Note: If the element is a simple value, pay attention to the data format when directly assigning the result of the
`get()` method to a program variable. For example, if the JSON string value represents a date, it will use the ISO
format (YYYY-MM-DD). Assigning this ISO-formatted date string to a `DATE` variable will fail, if `DBDATE` format
does not match.

A name/value pair can be set with the `put()` method.
Example

```
IMPORT util
MAIN
  DEFINE arr util.JSONArray
  LET arr = util.JSONArray.parse('"[123,"abc",null]"
  DISPLAY arr.get(2) -- abc
END MAIN
```

Related concepts

`util.JSONArray.put` on page 2789
Sets an element by position in the JSON array object.

`util.JSONArray.getLength`
Returns the number of elements in the JSON array object.

Syntax

```
util.JSONArray.getLength()
RETURNS INTEGER
```

Usage

The `getLength()` method returns the number of elements in the JSON array object.
This method can be used along with the `get()` and `getType()` method to read elements of a JSON array object.

Example

```
IMPORT util
MAIN
  DEFINE arr util.JSONArray
  DEFINE i INTEGER
  LET arr = util.JSONArray.parse('[123,8723,9232]
  FOR i=1 TO arr.getLength()
    DISPLAY i, ", ", arr.get(i)
  END FOR
END MAIN
```

Related concepts

`util.JSONArray.getType` on page 2788
Returns the type of a JSON array element.

`util.JSONArray.getType`
Returns the type of a JSON array element.

Syntax

```
util.JSONArray.getType(
  index INTEGER )
RETURNS STRING
```

1. `index` is the ordinal position of the element.

Usage

The `getType()` method returns the data type name corresponding to the JSON array element at the given position.
The index corresponding to the first element is 1.
This method can be used along with the `getLength()` method to read the entries of a JSON array object.

Possible values returned by this method are:

- **NUMBER**: A numeric value.
- **STRING**: A string value delimited by double quotes.
- **BOOLEAN**: A boolean value (true/false)
- **NULL**: A non-existing element.
- **OBJECT**: A structured object.
- **ARRAY**: An ordered list of elements.

**Example**

```plaintext
IMPORT util
MAIN
  DEFINE arr util.JSONArray
  LET arr = util.JSONArray.parse('[123,"abc",null]')
  DISPLAY arr.getType(1) -- NUMBER
  DISPLAY arr.getType(2) -- STRING
  DISPLAY arr.getType(3) -- NULL
END MAIN
```

**Related concepts**

- **util.JSONArray.getLength** on page 2788
  Returns the number of elements in the JSON array object.

- **util.JSONArray.put**
  Sets an element by position in the JSON array object.

**Syntax**

```plaintext
util.JSONArray.put(  
  index INTEGER,  
  value value-type )
```

1. **index** is the index of the element in the JSON array object.
2. **value** is the value to be associated to the index.
3. **value-type** can be a simple string or numeric type, a **RECORD** or a **DYNAMIC ARRAY**.

**Usage**

The `put()` method sets an element value by position in the JSON array object.

The first parameter is the index of the element. The second parameter can be a simple string or numeric value, or a complex variable defined as **RECORD** or **DYNAMIC ARRAY**.

The index corresponding to the first element is 1.

If the element exists, the existing value is replaced.

**Example**

```plaintext
IMPORT util
MAIN
  DEFINE ja util.JSONArray
  DEFINE rec RECORD
  id INTEGER,
  name STRING
  END RECORD
  DEFINE arr DYNAMIC ARRAY OF INTEGER
  LET ja = util.JSONArray.create()
```
CALL ja.put(1, 234)
LET rec.id = 234
LET rec.name = "Barton"
CALL ja.put(2, rec)
LET arr[1] = 234
LET arr[2] = 2837
CALL ja.put(3, arr)
DISPLAY ja.toString()
END MAIN

Related concepts

util.JSONArray.get on page 2787
Returns the value of a JSON array element.

util.JSONArray.remove
Removes the specified entry in the JSON array object.

Syntax

util.JSONArray.remove(
  index INTEGER )

1. index is the index of the element in the JSON array object.

Usage

The remove() method deletes an element in the JSON array object at the given position.
The index corresponding to the first element is 1.
If no element exists at the specified index, the method returns silently.

Example

IMPORT util
MAIN
  DEFINE arr util.JSONArray
  LET arr = util.JSONArray.parse('["aa","bb","cc"]')
  CALL arr.remove(2)
  DISPLAY arr.get(2) -- cc
END MAIN

Related concepts

util.JSONArray.put on page 2789
Sets an element by position in the JSON array object.

util.JSONArray.toFGL
Fills a dynamic array variable with the elements contained in the JSON array object.

Syntax

util.JSONArray.toFGL(
  arrayRef dynamic-array-type )

1. arrayRef is the array variable to be set with values of the JSON string.
   Important: The arrayRef is a dynamic array passed by reference to the method.
2. dynamic-array-type is a DYNAMIC ARRAY OF ... type.
| Library reference | 2791

Usage
The toFGL() method fills the DYNAMIC ARRAY passed as parameter with the corresponding values defined in the
JSON array object.
The destination array must have the same structure as the JSON source data. For more details see JSON support on
page 575.
Example
IMPORT util
MAIN
DEFINE ja util.JSONArray
DEFINE arr DYNAMIC ARRAY OF STRING
LET ja = util.JSONArray.parse('["aa","bb","cc"]')
CALL ja.toFGL(arr)
DISPLAY arr[2] -- bb
END MAIN
Related concepts
Arrays on page 418
Arrays (static or dynamic) allow you to handle an ordered collection of elements.
Records on page 408
Records allow structured program variables definitions.
util.JSONArray.toString on page 2791
Builds a JSON string from the elements contained in the JSON array object.
util.JSONArray.toString
Builds a JSON string from the elements contained in the JSON array object.
Syntax
util.JSONArray.toString()
RETURNS STRING
Usage
The toString() method produces a JSON formatted string from the elements contained in the JSON array object.
Example
IMPORT util
MAIN
DEFINE ja util.JSONArray
LET ja = util.JSONArray.create()
CALL ja.put(1,"aa")
CALL ja.put(2,"bb")
CALL ja.put(3,"cc")
DISPLAY ja.toString() -- ["aa","bb","cc"]
END MAIN
Related concepts
util.JSONArray.toFGL on page 2790


Fills a dynamic array variable with the elements contained in the JSON array object.

**The os package**

These topics cover the classes of the os package.

**The os.Path class**

The os.Path class provides functions to manipulate files and directories on the machine where the program executes.

This class is provided in the util C-Extension library. To use the os.Path extension, you must import the os package in your program:

```python
IMPORT os
```

In order to manipulate files, this API gives you access to low-level system functions. Pay attention to operating system specific conventions like path separators.

**Important:** Some methods are OS specific, like os.Path.rwx() which works only on UNIX® systems. Other methods may behave differently, depending on the OS. For example, the os.Path.rename() method cannot rename a file across file systems on a UNIX® platform.

**Related concepts**

IMPORT C-Extension on page 481
The **IMPORT** instruction imports C extension module elements to be used by the current module.
## os.Path methods

### Table 597: Class methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>os.Path.atime(path STRING)</code></td>
<td></td>
</tr>
<tr>
<td><code>RETURNS STRING</code></td>
<td>Returns the time of the last file access.</td>
</tr>
<tr>
<td><code>os.Path.basename(path STRING)</code></td>
<td></td>
</tr>
<tr>
<td><code>RETURNS STRING</code></td>
<td>Returns the last element of a path.</td>
</tr>
<tr>
<td><code>os.Path.chDir(path STRING)</code></td>
<td></td>
</tr>
<tr>
<td><code>RETURNS INTEGER</code></td>
<td>Changes the current working directory.</td>
</tr>
<tr>
<td><code>os.Path.chOwn(path STRING, uid INTEGER, gui INTEGER)</code></td>
<td></td>
</tr>
<tr>
<td><code>RETURNS INTEGER</code></td>
<td>Changes the UNIX™ owner and group of a file.</td>
</tr>
<tr>
<td><code>os.Path.chRwx(path STRING, mode INTEGER)</code></td>
<td></td>
</tr>
<tr>
<td><code>RETURNS INTEGER</code></td>
<td>Changes the UNIX™ permissions of a file.</td>
</tr>
<tr>
<td><code>os.Path.chVolume(volume STRING)</code></td>
<td></td>
</tr>
<tr>
<td><code>RETURNS INTEGER</code></td>
<td>Changes the current working volume.</td>
</tr>
<tr>
<td><code>os.Path.copy(fromPath STRING, toPath STRING)</code></td>
<td></td>
</tr>
<tr>
<td><code>RETURNS INTEGER</code></td>
<td>Creates a new file by copying an existing file.</td>
</tr>
<tr>
<td><code>os.Path.delete(path STRING)</code></td>
<td></td>
</tr>
<tr>
<td><code>RETURNS INTEGER</code></td>
<td>Deletes a file or a directory.</td>
</tr>
<tr>
<td><code>os.Path.dirClose(dirHandle INTEGER)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Closes the directory referenced by the directory opened by <code>os.Path.diropen()</code>.</td>
</tr>
<tr>
<td><code>os.Path.dirFMask(mask INTEGER)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Defines a filter mask for <code>os.Path.dirOpen()</code>.</td>
</tr>
<tr>
<td><code>os.Path dirname(path STRING)</code></td>
<td></td>
</tr>
<tr>
<td><code>RETURNS STRING</code></td>
<td>Returns all components of a path excluding the last one.</td>
</tr>
<tr>
<td><code>os.Path.dirNext(dirHandle INTEGER)</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Reads the next entry in the directory opened with <code>os.Path.diropen()</code>.</td>
</tr>
</tbody>
</table>
**os.Path.atime**

Returns the time of the last file access.

**Syntax**

```sql
os.Path.atime(
    path STRING
) RETURNS STRING
```

1. *path* is the path to a file.

**Usage**

The function returns a string containing the last access time for the specified file, in the standard format 'YYYY-MM-DD HH:MM:SS'.

If the function fails, it returns **NULL**.

**os.Path.baseName**

Returns the last element of a path.

**Syntax**

```sql
os.Path.baseName(
    path STRING
) RETURNS STRING
```

1. *path* is the name of the file.

**Usage**

This method extracts the last component of a path provided as argument.

For example, if you pass "/root/dir1/file.ext" as the parameter, it will return "file.ext".

See [Example 1: Extracting the parts of a file name](page 2809) on page 2809 for more examples.

**os.Path.copy**

Creates a new file by copying an existing file.

**Syntax**

```sql
os.Path.copy(
    fromPath STRING, 
    toPath STRING 
) RETURNS INTEGER
```

1. *fromPath* is the name of the file to copy.
2. *toPath* is the destination filename.

**Usage**

The function returns **TRUE** if the file has been successfully copied, **FALSE** otherwise.

**Example**

```sql
IF NOT os.Path.copy("/tmp/result.txt","/tmp/result-backup.txt") THEN
    DISPLAY "Could not copy the file."
END IF
```
**os.Path.chDir**
Changes the current working directory.

**Syntax**

```plaintext
os.Path.chDir(
    path STRING,
) RETURNS INTEGER
```

1. `path` is the path to the directory to select.

**Usage**
Use this function to change the current working directory.
The function returns `TRUE` if the current directory is successfully selected, `FALSE` otherwise.

**os.Path.chRwx**
Changes the UNIX™ permissions of a file.

**Syntax**

```plaintext
os.Path.chRwx(
    path STRING,
    mode INTEGER )
RETURNS INTEGER
```

1. `path` is the name of the file.
2. `mode` is the UNIX™ permission combination in decimal (not octal!).

**Usage**
This method can only be used on UNIX™!
Function returns `TRUE` on success, `FALSE` otherwise.
The `mode` must be a decimal value which is the combination of read, write and execution bits for the user, group and other part of the UNIX™ file permission. Make sure to pass the `mode` as the decimal version of permissions, not as octal (the chrwx UNIX™ command takes an octal value as parameter). For example, to set `-rw-r--r--` permissions, you must pass `((4+2)*64) + (4*8) + 4 = 420` to this method.

**os.Path.chOwn**
Changes the UNIX™ owner and group of a file.

**Syntax**

```plaintext
os.Path.chOwn(
    path STRING,
    uid INTEGER,
    gui INTEGER )
RETURNS INTEGER
```

1. `path` is the name of the file.
2. `uid` is the user id.
3. `gui` is the group id.

**Usage**
This method can only be used on UNIX™!
Function returns TRUE on success, FALSE otherwise.

**os.Path.chVolume**
Changes the current working volume.

**Syntax**

```plaintext
os.Path.chVolume(
    volume STRING)
RETURNS INTEGER
```

1. `volume` is the name of the volume to select as the new current working volume.

**Usage**

To change the current volume to C:

```plaintext
LET result = os.Path.chVolume("C:\")
```

The function returns TRUE if the current working volume is successfully changed, FALSE otherwise.

**os.Path.delete**
Deletes a file or a directory.

**Syntax**

```plaintext
os.Path.delete(
    path STRING)
RETURNS INTEGER
```

1. `path` is the path to the file or directory to delete.

**Usage**

A directory can only be deleted if it is empty.

The function returns TRUE if the file has been successfully deleted, FALSE otherwise.

**os.Path.dirClose**
Closes the directory referenced by the directory opened by `os.Path.diropen()`.

**Syntax**

```plaintext
os.Path.dirClose(
    dirHandle INTEGER)
```

1. `dirHandle` is the directory handle of the directory to close.

**Usage**

This function closes the directory search handle opened with `os.Path.diropen()`.

**os.Path.dirFMask**
Defines a filter mask for `os.Path.diropen()`.

**Syntax**

```plaintext
os.Path.dirFMask(
    mask INTEGER)
```
1. *mask* defines the filter mask.

**Usage**

When you call this function, you define the filter mask for any subsequent `os.Path.dirOpen()` call. By default, all kinds of directory entries are selected by the `dirOpen()` function. You can restrict the number of entries by using a filter mask.

The parameter of the `os.Path.dirFMask()` function must be a combination of the following bits:

- 0x01 = Exclude hidden files (.*)
- 0x02 = Exclude directories
- 0x04 = Exclude symbolic links
- 0x08 = Exclude regular files

For example, to retrieve only regular files, you must call:

```java
CALL os.Path.dirFMask( 1 + 2 + 4 )
```

**os.Path.dirName**

Returns all components of a path excluding the last one.

**Syntax**

```java
os.Path.dirName(
    path STRING)
RETURNS STRING
```

1. *path* is the path to a file or directory.

**Usage**

This method removes the last component of a path provided as argument.

For example, if you pass "/root/dir1/file.ext" as the parameter, it will return "/root/dir1".

See [Example 1: Extracting the parts of a file name](#) on page 2809 for more examples.

**os.Path.dirNext**

Reads the next entry in the directory opened with `os.Path.dirOpen()`.

**Syntax**

```java
os.Path.dirNext(
    dirHandle INTEGER)
RETURNS STRING
```

1. *dirHandle* is the directory handle of the directory to read.

**Usage**

This function returns the next entry of the directory opened with `os.Path.dirOpen()`.

Returns NULL if all entries have been read.
**os.Path.dirOpen**

Opens a directory and returns an integer handle to this directory.

**Syntax**

```c
os.Path.dirOpen(  
    path STRING)  
RETURNS INTEGER
```

1. *path* is the name of the directory.
2. *handle* is the directory handle.

**Usage**

This function creates a handle to scan the elements of a directory.

The function returns a value of 0 if it fails to open the directory.

Before calling the `dirOpen()` method, you can define a filter with `os.Path.dirFMask()`, and a sort order with `os.Path.dirSort()`.

**os.Path.dirSort**

Defines the sort criteria and sort order for `os.Path.dirOpen()`.

**Syntax**

```c
os.Path.dirSort(  
    criteria STRING,  
    order INTEGER )
```

1. *criteria* is the sort criteria.
2. *order* defines ascending (1) or descending (-1) order.

**Usage**

When you call this function, you define the sort criteria and sort order for any subsequent `os.Path.dirOpen()` call.

The *criteria* parameter must be one of the following strings:

- "undefined" = No sort. This is the default. Entries are read as returned by the OS functions.
- "name" = Sort by file name.
- "size" = Sort by file size.
- "type" = Sort by file type (directory, link, regular file).
- "atime" = Sort by access time.
- "mtime" = Sort by modification time.
- "extension" = Sort by file extension.

When sorting by name, directory entries will be ordered based on the current locale.

When sorting by any criteria other than the file name, entries having the same value for the given criteria are ordered by name following the value of the *order* parameter.

**os.Path.executable**

Checks if a file is executable.

**Syntax**

```c
os.Path.executable(  
    path STRING)
```
1. *path* is the path to a file.

**Usage**
The function returns **TRUE** if the file is executable, **FALSE** otherwise.

*os.Path.exists*
Checks if a file exists.

**Syntax**

```python
os.Path.exists(path STRING) RETURNS INTEGER
```

1. *path* is the path to a file.

**Usage**
The function returns **TRUE** if the file exists, **FALSE** otherwise.

*os.Path.extension*
Returns the file extension.

**Syntax**

```python
os.Path.extension(path STRING) RETURNS STRING
```

1. *path* is the path to a file.

**Usage**
The function returns the string following the last dot found in *path*.
If *path* does not have an extension, the function returns **NULL**.

*os.Path.fullPath*
Returns the canonical equivalent of a path.

**Syntax**

```python
os.Path.fullPath(path STRING) RETURNS STRING
```

1. *path* is the path to complete.

**Usage**
The *os.path.fullPath()* class method takes a path as parameter and resolves extra path separator characters (/ on UNIX™, \ on Windows®), as well as references to current (.) and parent directory (..). The result is called a canonical path.

On UNIX™, symbolic links **are not** followed. Use the *os.Path.isLink()* method to identify symbolic links.
Example

```display
os.Path.fullPath("/home/usr//scott/tmp/../images")
```

Resolves to:

```
/home/usr/scott/images
```

`os.Path.gid`

Returns the UNIX™ group id of a file.

Syntax

```python
os.Path.gid(path STRING)
RETURNS INTEGER
```

1. `path` is the path to a file.

Usage

This function returns the group id of the file.

This method can only be used on UNIX™!

Function returns -1 if it fails to get the user id.

`os.Path.glob`

Returns a list of files matching the specified pattern.

Syntax

```python
os.Path.glob(pattern STRING)
RETURNS DYNAMIC ARRAY OF STRING
```

1. `pattern` is a string representing file names, using wildcard characters *, ?, and [] as the MATCHES operator.

Usage

The `glob()` method produces a DYNAMIC ARRAY OF STRING containing a list of pathnames of the file system that match the pattern provided as parameter.

If no file or directory path matches the pattern, the `glob()` method returns an empty array.

The pattern can contain wildcard as follows:

- `?`: matches any single char at this position (except the path separator).
- `*`: matches any string (not including the path separator).
- `[ set-of-chars ]`: matches any character enclosed by the brackets. Can define a range of characters with the hyphen ( `[a-z]` ). The matching is inverted when starting with a `^`: `[^0-9]` matches any non digit.

Globbing is applied on each of the components of a pathname separately. The path separator (Unix `/`, Windows `\` or `\`) splits the pattern in subpatterns. A pattern in the form `"pattern1/pattern2"` finds files matching `pattern2` in each directory matching `pattern1`. Thus `glob()` limits the result to directory-names if the pattern ends with a path separator.
Table 598: Examples of glob() patterns

<table>
<thead>
<tr>
<th>glob pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*.4gl</td>
<td>Files in current dir ending with '.4gl'</td>
</tr>
<tr>
<td>m*</td>
<td>Files in current dir starting with the character 'm'</td>
</tr>
<tr>
<td>[Mm]akefile</td>
<td>The file Makefile or makefile in current dir</td>
</tr>
<tr>
<td>[0-9]*</td>
<td>Files in current dir starting with a digit</td>
</tr>
<tr>
<td>subdir/*.4gl</td>
<td>Files ending with '.4gl' in the directory 'subdir'</td>
</tr>
<tr>
<td>subdir[12]/*.4gl</td>
<td>Files ending with '.4gl' in subdir1 or subdir2</td>
</tr>
<tr>
<td>..//*.4gl</td>
<td>Files ending with '.4gl' in the parent directory</td>
</tr>
<tr>
<td>/Users/Me/*.4gl</td>
<td>Files ending with '.4gl' in the directory 'Users/Me'</td>
</tr>
<tr>
<td>*/</td>
<td>All subdirectories (at level 1)</td>
</tr>
</tbody>
</table>

Example

```python
IMPORT os
MAIN
   DEFINE files DYNAMIC ARRAY OF STRING
   DEFINE x INTEGER
   LET files = os.Path.glob("[A-Z]*.4gl")
   FOR x=1 TO files.getLength()
      DISPLAY "File: ", files[x]
   END FOR
END MAIN
```

*os.Path.homeDir*

Returns the path to the HOME directory of the current user.

**Syntax**

```python
os.Path.homeDir()
RETURNS STRING
```

**Usage**

This function returns the home directory of the current user.

*os.Path.isDirectory*

Checks if a file is a directory.

**Syntax**

```python
os.Path.isDirectory(
   path STRING)
RETURNS BOOLEAN
```

1. *path* is the path to the directory.

**Usage**

The function returns TRUE if the file is a directory, FALSE otherwise.
**os.Path.isFile**  
Checks if a file is a regular file.

**Syntax**
```
os.Path.isFile(
    path STRING
)  
RETURNS BOOLEAN
```

1. `path` is the path to a file.

**Usage**
The function returns TRUE if the file is a regular file, FALSE otherwise.

**os.Path.isHidden**  
Checks if a file is hidden.

**Syntax**
```
os.Path.isHidden(
    path STRING
)  
RETURNS BOOLEAN
```

1. `path` is the file or directory path.

**Usage**
The function returns TRUE if the file is hidden, FALSE otherwise.

For example, on UNIX™, files starting with a dot in the file name are considered as hidden when using the `ls` command.

**os.Path.isLink**  
Checks if a file is an UNIX™ symbolic link.

**Syntax**
```
os.Path.isLink(
    path STRING
)  
RETURNS BOOLEAN
```

1. `path` is the file or directory path.

**Usage**
The function returns TRUE if the files is a symbolic link, FALSE otherwise.

This method can only be used on UNIX™!

**os.Path.isRoot**  
Checks if a file path is a root path.

**Syntax**
```
os.Path.isRoot(
    path STRING
)  
RETURNS BOOLEAN
```

1. `path` is the path to check.
**Usage**

The function returns TRUE if the path is a root path, FALSE otherwise.

On UNIX™ the root path is '/'.

On Windows® the root path matches "[a-zA-Z]:\".

**os.Path.join**

Joins two path segments adding the platform-dependent separator.

**Syntax**

```plaintext
os.Path.join(
    begin STRING,
    end STRING
) RETURNS STRING
```

1. `begin` is the beginning path segment.
2. `end` is the ending path segment.

**Usage**

Use this method to construct a path with no system-specific code to use the correct path separator:

```plaintext
LET path = os.Path.join(os.Path.homedir(), name)
```

This method returns the ending path segment if it is an absolute path.

If one of the arguments is NULL, the function returns NULL.

**os.Path.makeTempName**

Generates a temporary file name.

**Syntax**

```plaintext
os.Path.makeTempName() RETURNS STRING
```

**Usage**

This method creates a new temporary file path, with a unique file name, in the temporary directory of the process.

**Note:** The method just returns a new temporary file name. A file created with this name must be deleted by the programmer. It will not be deleted automatically when exiting the program.

The temporary directory can be defined with the DBTEMP environment variable. If the DBTEMP variable is not defined, the runtime system uses the temporary directory as defined by the operating system. For more details, see the DBTEMP environment variable.

**os.Path.mtime**

Returns the time of the last file modification.

**Syntax**

```plaintext
os.Path.mtime(
    path STRING
) RETURNS STRING
```

1. `path` is the path to a file.
Usage

The function returns a string containing the last modification time for the specified file, in the standard format 'YYYY-MM-DD HH:MM:SS'.

If the function fails, it returns NULL.

\textit{os.Path.mkDir}

Creates a new directory.

Syntax

\begin{verbatim}
\texttt{os.Path.mkDir(path STRING) RETURNS INTEGER}
\end{verbatim}

1. \textit{path} is the path of the directory to create.

Usage

The function returns TRUE if the directory has been successfully created, FALSE otherwise.

\textit{os.Path.pathSeparator}

Returns the character used in environment variables to separate path elements.

Syntax

\begin{verbatim}
\texttt{os.Path.pathSeparator() RETURNS STRING}
\end{verbatim}

Usage

You typically use this method to build a path from two components.

On UNIX™, the path separator is '\:'.

On Windows®, the path separator is '\;'.

\textit{os.Path.pathType}

Checks if a path is a relative path or an absolute path.

Syntax

\begin{verbatim}
\texttt{os.Path.pathType(path STRING) RETURNS STRING}
\end{verbatim}

1. \textit{path} is the path to check.

Usage

The function returns "absolute" if the path is an absolute path, or "relative" if the path is a relative path.

\textbf{Note:} On Unix-style platforms, an absolute path starts with a (/) slash. On Windows® platforms, an absolute path starts with a backslash (\), a slash (/), or starts with a drive letter followed by a colon (C:)

If the path is NULL, the function returns NULL.
**os.Path.pwd**  
Returns the current working directory.

**Syntax**

```
    os.Path.pwd()
    RETURNS STRING
```

**Usage**

This function returns the path of the current working directory.

On a mobile device, this front call returns the current application working directory:

- On Android™, it returns the directory where the program executes.
- On iOS, it returns the "Documents" directory under the application directory.

**os.Path.readable**  
Checks if a file is readable.

**Syntax**

```
    os.Path.readable(  
          path STRING) 
    RETURNS INTEGER
```

1. *path* is the path to a file or directory.

**Usage**

The function returns TRUE if the file is readable, FALSE otherwise.

**os.Path.rename**  
Renames a file or a directory.

**Syntax**

```
    os.Path.rename(  
          oldPath STRING,  
          newPath STRING ) 
    RETURNS INTEGER
```

1. *oldPath* is the current name of the file or directory to be renamed.
2. *newPath* is the new name to assign to the file or directory.

**Usage**

The function returns TRUE if the file or directory has been successfully renamed, FALSE otherwise.

On UNIX™ platforms, you can rename/move files and directories located on the same file system.

On Microsoft™ Windows® platforms only files can be renamed/moved. However, on Windows® you can move files across disks and directories.

**os.Path.separator**  
Returns the character used to separate path segments.

**Syntax**

```
    os.Path.separator()
```
Usage
Use this method to build a path from two components.
On UNIX™, the directory separator is '/'.
On Windows®, the directory separator is '\'.

__os.Path.size__
Returns the size of a file.

Syntax
```
__os.Path.size__(
    __path__ STRING)
RETURNS BIGINT
```

1. `path` is the path to a file.

Usage
The function returns the size in bytes for the specified file.
Large files sizes (> 2GB) are supported.

__os.Path.rootDir__
Returns the root directory of the current working path.

Syntax
```
__os.Path.rootDir__()
RETURNS STRING
```

Usage
This function returns the root directory of the current working path.
On UNIX™, it always returns "/".
On Windows® it returns the current working drive as "[a-zA-Z]:\"

__os.Path.rootName__
Returns the file path without the file extension of the last element of the file path.

Syntax
```
__os.Path.rootName__(
    __path__ STRING)
RETURNS STRING
```

1. `path` is the path to a file or directory.

Usage
This method removes the file extension from the path provided as parameter.
For example, if you pass "/root/dirl/file.ext" as the parameter it will return "/root/dirl/file".
See Example 1: Extracting the parts of a file name on page 2809 for more examples.
**os.Path.rwx**

Returns the UNIX™ file permissions of a file.

**Syntax**

```python
os.Path.rwx(
    path STRING)
RETURNS INTEGER
```

1. `path` is the path to a file or directory.

**Usage**

This function returns the combination of permissions for user, group and other.

This method can only be used on UNIX™!

Function returns -1 if it fails to get the permissions.

The `mode` is returned as a decimal value which is the combination of read, write and execution bits for the user, group and other part of the UNIX™ file permission. For example, if a file has the `-rwxr-xr-x` permissions, the method returns `((4+2+1) * 64 + (4+1) * 8) + (4+1) ) = 493`.

**os.Path.type**

Returns the file type as a string.

**Syntax**

```python
os.Path.type(
    path STRING)
RETURNS STRING
```

1. `path` is the path to file.

**Usage**

On UNIX™, this method follows symbolic links. Use the `os.Path.islink()` method to identify symbolic links.

The possible values returned by this method are:

1. `file`: the file is a regular file
2. `directory`: the file is a directory
3. `socket`: the file is a socket
4. `fifo`: the file is a fifo
5. `block`: the file is a block device
6. `char`: the file is a character device

**os.Path.uid**

Returns the UNIX™ user id of a file.

**Syntax**

```python
os.Path.uid(
    path STRING)
RETURNS INTEGER
```

1. `path` is the path to a file or directory.
Usage
This method can only be used on UNIX™!
Function returns -1 if it fails to get the user id.

*os.Path.volumes*
Returns the available volumes.

**Syntax**

```python
os.Path.volumes()
RETURNS STRING
```

Usage
The function returns the list of all available volumes separated by "|".
To display the list of available volumes, a volume is identified by its letter, followed by a colon and a backslash (:\).

```python
DISPLAY os.Path.volumes()
```

Output example:

```
C:\|E:\|F:\
```

*os.Path.writable*
Checks if a file is writable.

**Syntax**

```python
os.Path.writable(
    path STRING)
RETURNS INTEGER
```

1. *path* is the path to a file or directory.

Usage
The function returns TRUE if the file is writable, FALSE otherwise.

Examples
*os.Path usage examples.*

**Example 1: Extracting the parts of a file name**

This program uses the file functions to extract the directory name, the base name, the root name, and the file extension:

```python
IMPORT os
MAIN
    DISPLAY "Dir name  = ", os.Path.dirName(arg_val(1))
    DISPLAY "Base name  = ", os.Path.baseName(arg_val(1))
    DISPLAY "Root name  = ", os.Path.rootName(arg_val(1))
    DISPLAY "Extension = ", os.Path.extension(arg_val(1))
END MAIN
```

Example results:
Table 599: Example results

<table>
<thead>
<tr>
<th>Path</th>
<th>os.Path.dirname</th>
<th>os.Path.basename</th>
<th>os.Path.rootname</th>
<th>os.Path.extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>..</td>
<td>.</td>
<td>..</td>
<td>.</td>
<td>NULL</td>
</tr>
<tr>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>NULL</td>
</tr>
<tr>
<td>/usr/lib</td>
<td>/usr</td>
<td>lib</td>
<td>/usr/lib</td>
<td>NULL</td>
</tr>
<tr>
<td>/usr/</td>
<td>/</td>
<td>usr</td>
<td>/usr/</td>
<td>NULL</td>
</tr>
<tr>
<td>usr</td>
<td>.</td>
<td>usr</td>
<td>usr</td>
<td>NULL</td>
</tr>
<tr>
<td>file.xx</td>
<td>.</td>
<td>file.xx</td>
<td>file</td>
<td>xx</td>
</tr>
<tr>
<td>/tmp.yy/</td>
<td>/tmp.yy</td>
<td>file.xx</td>
<td>/tmp.yy/file</td>
<td>xx</td>
</tr>
<tr>
<td>/tmp.yy/</td>
<td>/tmp.yy</td>
<td>file.xx</td>
<td>/tmp.yy/file</td>
<td>/tmp.yy/file</td>
</tr>
<tr>
<td>/tmp.yy/</td>
<td>/</td>
<td>tmp.yy</td>
<td>/tmp.yy/</td>
<td>NULL</td>
</tr>
<tr>
<td>/tmp.yy/.</td>
<td>/tmp.yy</td>
<td>.</td>
<td>/tmp.yy/</td>
<td>NULL</td>
</tr>
</tbody>
</table>

These examples use UNIX™ file names. On Windows® the result would be different, as the file name separator is a backslash (\).

**Example 2: Browsing directories**

This program takes a directory path as an argument and scans the content recursively:

```
IMPORT os

MAIN
    CALL showDir(arg_val(1))
END MAIN

FUNCTION showDir(path)
    DEFINE path STRING
    DEFINE child STRING
    DEFINE h INTEGER

    IF NOT os.Path.exists(path) THEN
        RETURN
    END IF
```
IF NOT os.Path.isDirectory(path) THEN
  DISPLAY " ", os.Path.baseName(path)
  RETURN
END IF

DISPLAY "[", path, "]"
CALL os.Path.dirSort("name", 1)
LET h = os.Path.dirOpen(path)
WHILE h > 0
  LET child = os.Path.dirNext(h)
  IF child IS NULL THEN EXIT WHILE END IF
  IF child == "." OR child == ".." THEN CONTINUE WHILE END IF
  CALL showDir( os.Path.join( path, child ) )
END WHILE

CALL os.Path.dirClose(h)
END FUNCTION

The com package
The Genero Web Services com package provides classes and methods that allow you to perform tasks associated with creating Services and Clients, and managing the services.

Use the IMPORT statement at the top of the module using this library:

```
IMPORT com
```

Related concepts
Web services on page 3270
Create a Web service client or server with Genero BDL.

Web services classes
The Web services classes manage Web Services servers.

- The WebService class on page 2811
- The WebOperation class on page 2821
- The WebServiceEngine class on page 2828
- The HTTPServiceRequest class on page 2845

The WebService class
The com.WebService class provides an interface to create and manage Genero Web Services.

The com.WebServices class is used to implement a Web Service on the server.

Important: This Web Services class is not supported on GMI mobile devices.

Related concepts
Web services on page 3270
Create a Web service client or server with Genero BDL.

**com.WebServices methods**
Methods for the `com.WebService` class.

**Table 600: Class methods**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
Table 601: Object methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>createFault</code></td>
<td>Creates a global fault for a Web Service object.</td>
</tr>
<tr>
<td><code>createHeader</code></td>
<td>Defines the header for the Web Service object.</td>
</tr>
<tr>
<td><code>generateWSDL</code></td>
<td>Creates an <code>xml.DomDocument</code> object with the WSDL corresponding to the Web Service object.</td>
</tr>
<tr>
<td><code>publishOperation</code></td>
<td>Publishes a Web Operation.</td>
</tr>
<tr>
<td><code>registerInputHTTPVariable</code></td>
<td>Registers the record variable for HTTP input.</td>
</tr>
<tr>
<td><code>registerInputRequestHandler</code></td>
<td>Registers the function to be executed on incoming SOAP requests.</td>
</tr>
<tr>
<td><code>registerOutputHTTPVariable</code></td>
<td>Registers the record variable for HTTP output.</td>
</tr>
<tr>
<td><code>registerOutputRequestHandler</code></td>
<td>Registers the function to be executed just before the SOAP response is forwarded to the client.</td>
</tr>
<tr>
<td><code>registerWSDLHandler</code></td>
<td>Registers the function to be executed when a WSDL is generated.</td>
</tr>
<tr>
<td><code>saveWSDL</code></td>
<td>Writes to a file the WSDL corresponding to the Web Service object.</td>
</tr>
<tr>
<td><code>setComment</code></td>
<td>Defines the comment for the Web Service object.</td>
</tr>
<tr>
<td><code>setFeature</code></td>
<td>Defines a feature for the current Web Service object.</td>
</tr>
</tbody>
</table>

`com.WebService.createFault`
Creates a global fault for a Web Service object.

**Syntax**

```java
createFault(
    fault RECORD,
    encoded INTEGER
)
```

1. *fault* defines the Web Service fault.
2. *encoded* defines an integer value specifying the encoding mechanism.

**Usage**

The `createFault()` method creates a global fault for this Web Service object.

The *fault* parameter can be of any type that defines the SOAP fault in a SOAP response. In case of SOAP fault, the client for this Web Service will receive a variable with the same structure.

When the parameter *encoded* is true (1), the SOAP Section 5 encoding mechanism is used. False (0) indicates the XML Schema mechanism.

In case of error, the method throws an exception and sets the *STATUS* variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (*STATUS*) on page 3554.

**com.WebService.createHeader**

Defines the header for the Web Service object.

**Syntax**

```java
createHeader(
    header RECORD,
    encoded INTEGER
)
```

1. *header* defines the header for the Web Service object.
2. *encoded* defines an integer value specifying the encoding mechanism.

**Usage**

The `createHeader()` method creates a global header for the current Web Service object.

The Web Service header is defined by the first parameter. This will define SOAP headers exchanged by the client and server.

When the parameter *encoded* is true (1), the SOAP Section 5 encoding mechanism is used. False (0) indicates the XML Schema mechanism.

**Important:** Since Web Services headers are generally in Document Style, we recommend to set the *encoded* parameter to 0.

In case of error, the method throws an exception and sets the *STATUS* variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (*STATUS*) on page 3554.

**com.WebService.CreateWebService**

Creates a new object to implement a Web Service.

**Syntax**

```java
com.WebService.CreateWebService(
    name STRING,
```
ns STRING )  
RETURNS com.WebService

1. *name* defines the Web service identifier.
2. *ns* defines the namespace for the Web service.

**Usage**


The *name* and *ns* parameters must uniquely identify the Web service across the entire application, when multiple Web service programs run on the same server. In theory, the value of *ns*+*name* must be unique on the internet.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

`com.WebService.CreateStatefulWebService`

Creates a new object to implement a stateful Web service.

**Syntax**

```java
com.WebService.CreateStatefulWebService(
  name STRING,
  ns STRING ,
  state RECORD )
RETURNS com.WebService
```

1. *name* defines the Web service identifier.
2. *ns* defines the namespace for the Web service.
3. *state* defines the state between the client and server.

**Usage**


The *name* and *ns* parameters must uniquely identify the Web service across the entire application, when multiple Web service programs run on the same server. In theory, the value of *ns*+*name* must be unique on the internet.

The *state* variable is used to identify the state between the client and the server:

- For a WS-Addressing stateful service, the *state* variable must be a RECORD with the following structure, with the `W3CEndpointReference` variable attribute:

  ```java
  RECORD ATTRIBUTES(W3CEndpointReference)
    address STRING, -- The location of the Web Service (for ex: URL)
    ref RECORD
    ... (other members defining the state)
  END RECORD
  END RECORD
  ```

- For a stateful service based on HTTP cookies, the *state* variable must be a simple variable defined with a basic data type.

It is up to the programmer to manage the *state* variable and to restore the service state from a database.

When creating a stateful Web service, all published Web operations require a session in the client request except those defined as 'initiateSession'.

```java
RECORD ATTRIBUTES(W3CEndpointReference)
  address STRING, -- The location of the Web Service (for ex: URL)
  ref RECORD
  ... (other members defining the state)
END RECORD
END RECORD
```
In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**Related concepts**

Stateful SOAP Web services on page 3283
The GWS provides support for stateful service. Different options for implementing SOAP stateful services are described.

com.WebService.generateWSDL

Creates an `xml.DomDocument` object with the WSDL corresponding to the Web Service object.

**Syntax**

```java
generateWSDL(
   location STRING )
RETURNS xml.DomDocument
```

1. `location` defines the URL where the Web Service will be deployed.

**Usage**

The `generateWSDL()` method creates a new `xml.DomDocument` object containing the WSDL data of the Web Service object.

The URL where the Web Service will be deployed must be specified.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**Related concepts**

com.WebService.setComment on page 2820
Defines the comment for the Web Service object.

com.WebService.publishOperation

Publishes a Web Operation.

**Syntax**

```java
publishOperation(
   op com.WebOperation,
   role STRING )
```

1. `op` defines the Web Operation object.
2. `role` identifies uniquely the Web Operation.

**Usage**

The `publishOperation()` method publishes the Web Operation specified by the `com.WebOperation` object passed as parameter.

The `role` identifies the operation, if several operations have the same name, by setting the SOAPAction HTTP header. Usually this parameter is set to NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

com.WebService.registerInputHTTPVariable
Registers the record variable for HTTP input.

**Syntax**

```plaintext
registerInputHTTPVariable(
  headers RECORD )
```

1. *headers* defines the HTTP input record variable with the following structure:

```plaintext
RECORD
  verb STRING,
  url STRING,
  headers DYNAMIC ARRAY OF RECORD
    name STRING,
    value STRING
  END RECORD
END RECORD
```

**Usage**

The `registerInputHTTPVariable()` method registers a program variable with a specific structure, that will be filled with the HTTP request headers when a Web Operation arrives.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**Related concepts**

- [com.WebService.registerOutputHTTPVariable](#) on page 2818
  Registers the record variable for HTTP output.

- [com.WebService.registerInputRequestHandler](#)
  Registers the function to be executed on incoming SOAP requests.

**Syntax**

```plaintext
registerInputRequestHandler(
  function STRING )
```

1. *function* has the name of a program function.

**Usage**

The `registerInputRequestHandler()` method registers a function to be called when an incoming SOAP request is received and before the SOAP engine has processed it.

The callback function must be defined with a unique parameter of type `xml.DomDocument`, and must return the reference to this object, or `NULL`:

```plaintext
FUNCTION myRequestInputHandler( in )
  DEFINE in xml.DomDocument
  ...
  RETURN in
END FUNCTION
```

The input callback function typically modifies the content of the SOAP input request DOM document object passed as parameter.

When returning `NULL` from the input callback function, the output callback function will be called with the default SOAP fault node, which can then be modified.
In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**Related concepts**

*Server handlers* on page 3344  
Create and register your callback handlers (request, and response) to modify the WSDL.

**com.WebService.registerOutputRequestHandler**  
Registers the function to be executed just before the SOAP response is forwarded to the client.

**com.WebService.registerOutputHTTPVariable**  
Registers the record variable for HTTP output.

### Syntax

```
registerOutputHTTPVariable(  
    headers RECORD  
)
```

1. *headers* defines the HTTP input record variable with the following structure:

```
RECORD  
    code INTEGER,  
    desc STRING,  
    headers DYNAMIC ARRAY OF RECORD  
        name STRING,  
        value STRING  
    END RECORD  
END RECORD
```

**Usage**

The `registerOutputHTTPVariable()` method registers a program variable with a specific structure, that will be used to fill the HTTP response headers when a Web Operation is completed.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**Related concepts**

*com.WebService.registerInputHTTPVariable* on page 2816  
Registers the record variable for HTTP input.

**com.WebService.registerOutputRequestHandler**  
Registers the function to be executed just before the SOAP response is forwarded to the client.

### Syntax

```
registerOutputRequestHandler(  
    function STRING  
)
```

1. *function* has the name of a program function.

**Usage**

The `registerOutputRequestHandler()` method registers a function to be called just after the SOAP engine has processed the request and before the SOAP response is forwarded to the client.
The output callback function must be defined with a unique parameter of type `xml.DomDocument`, and must return the reference to this object:

```
FUNCTION myRequestOutputHandler( out )
    DEFINE out xml.DomDocument
    ...
    RETURN out
END FUNCTION
```

The output callback function typically modifies the content of the SOAP output request DOM document object passed as parameter.

If `NULL` was returned from the input callback function, the output callback function will be called with the default SOAP fault node.

When WS-Addressing is enabled, and the server side callback is triggered, the SOAP engine has already created the SOAP:Header node with the correct WS-Addressing entries. If other headers need to be added (for WS-Security for instance), it is recommended to add them as child nodes of the existing SOAP header node, instead of creating a new header node. Otherwise, you may get two SOAP headers in the same response, which is not allowed.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**Related concepts**

- **Server handlers** on page 3344
  Create and register your callback handlers (request, and response) to modify the WSDL.

  - `com.WebService.registerInputRequestHandler` on page 2817
    Registers the function to be executed on incoming SOAP requests.

- `com.WebService.registerWSDLHandler`
  Registers the function to be executed when a WSDL is generated.

**Syntax**

```
registerWSDLHandler( 
    function STRING )
```

1. `function` has the name of a program function.

**Usage**

The `registerWSDLHandler()` method registers a function to be called when the WSDL of the current Web Service object is generated.

The callback function must be defined with a unique parameter of type `xml.DomDocument`, and must return the reference to this object:

```
FUNCTION myWSDLHandler( wsdl )
    DEFINE wsdl xml.DomDocument
    ...
    RETURN wsdl
END FUNCTION
```

The callback function typically modifies the content of the WSDL DOM document object passed as parameter.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.
**Related concepts**

**Server handlers** on page 3344
Create and register your callback handlers (request, and response) to modify the WSDL.

**com.WebService.saveWSDL**
Writes to a file the WSDL corresponding to the Web Service object.

**Syntax**

```java
saveWSDL(
    location STRING )
RETURNS INTEGER
```

1. *location* defines the URL where the Web Service will be deployed.

**Usage**
The `saveWSDL()` method writes the WSDL data corresponding to the Web Service object.

The URL where the Web Service will be deployed must be specified.

The name of the file will be the name of the Web Service defined by the *name* parameter passed to the `createWebService()` or `createStatefulWebService()` methods.

The method returns 0 if the file was saved, -1 in case of error.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See **Error handling in GWS calls (STATUS)** on page 3554.

**Related concepts**

**com.WebService.setComment** on page 2820
Defines the comment for the Web Service object.

**Syntax**

```java
setComment (
    comment STRING )
```

1. *comment* defines the comment to be set.

**Usage**
The `setComment()` method defines the comment associated with a `com.WebService` object.

The comment will be used when generating the WSDL file, as defined by the standard.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See **Error handling in GWS calls (STATUS)** on page 3554.

**com.WebService.setFeature**
Defines a feature for the current Web Service object.

**Syntax**

```java
setFeature(
    feature STRING,
    ...)
```

1. *feature* is a string that defines the feature to be set.
1. `feature` defines the name of the Web Service feature.
2. `value` defines the value of the feature.

**Usage**

The `setFeature()` method defines a feature for the current Web Service object by specifying a feature name and a value.

The features names are predefined. The second parameter must have a valid value for the specified feature.

**Table 602: Support Web Service features for the `setFeature()` method**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Soap1.1</td>
<td>Defines whether the Web Service supports the SOAP 1.1 protocol. Default value is <code>FALSE</code>.</td>
</tr>
<tr>
<td>Soap1.2</td>
<td>Defines whether the Web Service supports the SOAP 1.2 protocol. Default value is <code>FALSE</code>.</td>
</tr>
<tr>
<td>WS-Addressing1.0</td>
<td>Defines whether the Web Service supports WS-Addressing 1.0. Valid values include:</td>
</tr>
<tr>
<td></td>
<td>• <code>TRUE</code> - The service supports WS-Addressing 1.0 and accepts requests without WS-Addressing.</td>
</tr>
<tr>
<td></td>
<td>• <code>REQUIRED</code> - The service supports WS-Addressing 1.0 and accepts only requests with WS-Addressing.</td>
</tr>
<tr>
<td></td>
<td>• <code>FALSE</code> - WS-Addressing 1.0 is disabled (Default).</td>
</tr>
</tbody>
</table>

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**The `WebOperation` class**

The `com.WebOperation` class provides an interface to create and manage the operations of a Genero Web Service.

**Important:** This Web Services class is not supported on GMI mobile devices.

The Web Operation can be created as RPC Style or Document Style. Both RPC/Literal and Doc/Literal Styles are WS-I compliant (standards set by the Web Services Interoperability organization).

RPC Style Service (RPC/Literal) is generally used to execute a function, such as a service that returns a stock option. Document Style Service (Doc/Literal) is generally used for more sophisticated operations that exchange complex data structures, such as a service that sends an invoice to an application, or exchanges a Word document; this is the MS.Net default. The input or output `RECORD` cannot have `XMLNamespace` attributes set on their members.

Calling the appropriate function to create the desired style is the only difference in your Genero code that creates the service. The remainder of the code that describes the service is the same, regardless of whether you want to create an RPC or Document style of service.

Do not use the `setInputEncoded()` and `setOutputEncoded()` methods, as they will specify the RPC/Encoded Style, which is not recommended (see Choosing a Web Service Style).

Since release 2.0 GWS allows you to create RPC Style and Document Style operations in the same Web service. However, we do not recommend this, as it is not WS-I compliant.
**com.WebOperation methods**
Methods for the `com.WebOperation` class.

**Table 603: Class methods**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>com.WebOperation.CreateRPCStyle( function STRING, operation STRING, inputVar RECORD , outputVar RECORD )</code></td>
<td>Creates a new Web Operation object with RPC style.</td>
</tr>
</tbody>
</table>
Table 604: Object methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addFault (</td>
<td>Adds a fault to the current Web Operation definition.</td>
</tr>
<tr>
<td>fault RECORD ,</td>
<td></td>
</tr>
<tr>
<td>wsaaction STRING</td>
<td></td>
</tr>
<tr>
<td>addInputHeader (</td>
<td>Adds an input header for the current Web Operation definition.</td>
</tr>
<tr>
<td>header RECORD )</td>
<td></td>
</tr>
<tr>
<td>addOutputHeader (</td>
<td>Adds an output header for the current Web Operation definition.</td>
</tr>
<tr>
<td>header RECORD )</td>
<td></td>
</tr>
<tr>
<td>initiateSession (</td>
<td>Defines the Web Operation as session initiator.</td>
</tr>
<tr>
<td>ok INTEGER )</td>
<td></td>
</tr>
<tr>
<td>setComment (</td>
<td>Sets the comment for the Web Operation object.</td>
</tr>
<tr>
<td>comment STRING )</td>
<td></td>
</tr>
<tr>
<td>setInputAction (</td>
<td>Sets the WS-Addressing action identifier of the input operation.</td>
</tr>
<tr>
<td>action STRING )</td>
<td></td>
</tr>
<tr>
<td>setInputEncoded (</td>
<td>Defines the encoding mechanism for Web Operation input parameters.</td>
</tr>
<tr>
<td>val INTEGER )</td>
<td></td>
</tr>
<tr>
<td>setOutputAction (</td>
<td>Sets the WS-Addressing action identifier of the output operation.</td>
</tr>
<tr>
<td>action STRING )</td>
<td></td>
</tr>
<tr>
<td>setOutputEncoded (</td>
<td>Defines the encoding mechanism for Web Operation output parameters.</td>
</tr>
<tr>
<td>val INTEGER )</td>
<td></td>
</tr>
</tbody>
</table>

com.WebOperation.addFault
Adds a fault to the current Web Operation definition.

Syntax
addFault (  
    fault RECORD ,  
    wsaaction STRING )

1. fault defines the program variable defining the fault.
2. wsaaction defines the type of action.

Usage
Adds a fault the Web Operation can throw during operation processing, where fault is any variable previously created as fault of the com.WebService object, and wsaaction the WS-Addressing action identifier if WS-Addressing is supported. If WS-Addressing is not supported, pass NULL as second parameter.
In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

com.WebOperation.addInputHeader

Adds an input header for the current Web Operation definition.

**Syntax**

```java
addInputHeader (  
    header RECORD )
```

1. `header` defines the program variable defining the header.

**Usage**

This method adds a header to the Web Operation object for input parameters.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

com.WebOperation.addOutputHeader

Adds an output header for the current Web Operation definition.

**Syntax**

```java
addOutputHeader (  
    header RECORD )
```

1. `header` defines the program variable defining the header.

**Usage**

This method adds a header to the Web Operation object for input parameters.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

com.WebOperation.CreateDOCStyle

Creates a new Web Operation object with Document style.

**Syntax**

```java
com.WebOperation.CreateDOCStyle (  
    function STRING,  
    operation STRING,  
    inputVar RECORD ,  
    outputVar RECORD )  
RETURNS com.WebOperation
```

1. `function` defines the program function to be called to process the XML operation.
2. `operation` defines the XML operation.
3. `inputVar` defines the variable defining the input parameters of the operation (or NULL if there is none).
4. `outputVar` defines the variable defining the output parameters of the operation (or NULL if there is none).
Usage

This method creates a Request-Response Document style `com.WebOperation` object, where `function` is the name of the program function that is executed to process the XML operation.

The function name must be a string literal, not a string variable, due to operation publication restrictions.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

`com.WebOperation.CreateRPCStyle`

Creates a new Web Operation object with RPC style.

Syntax

```java
com.WebOperation.CreateRPCStyle(
    function STRING,
    operation STRING,
    inputVar RECORD,
    outputVar RECORD)
RETURNS com.WebOperation
```

1. `function` defines the program function to be called to process the XML operation.
2. `operation` defines the XML operation.
3. `inputVar` defines the variable defining the input parameters of the operation (or NULL if there is none).
4. `outputVar` defines the variable defining the output parameters of the operation (or NULL if there is none).

Usage

This method creates a Request-Response RPC style `com.WebOperation` object, where `function` is the name of the program function that is executed to process the XML operation.

The function name must be a string literal, not a string variable, due to operation publication restrictions.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

`com.WebOperation.CreateOneWayDOCStyle`

Creates a new Web Operation object with One-Way Document style.

Syntax

```java
com.WebOperation.CreateOneWayDOCStyle(
    function STRING,
    operation STRING,
    inputVar RECORD)
RETURNS com.WebOperation
```

1. `function` defines the program function to be called to process the XML operation.
2. `operation` defines the XML operation.
3. `inputVar` defines the variable defining the input parameters of the operation (or NULL if there is none).

Usage

This method creates a One-Way DOC style `com.WebOperation` object, where `function` is the name of the program function that is executed to process the XML operation.

The function name must be a string literal, not a string variable, due to operation publication restrictions.
There is no output parameter to be returned to the client.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

com.WebOperation.CreateOneWayRPCStyle

Creates a new Web Operation object with One-Way RPC style.

Syntax

```java
com.WebOperation.CreateOneWayRPCStyle( 
  function STRING, 
  operation STRING, 
  inputVar RECORD 
) 
RETURNS com.WebOperation
```

1. `function` defines the program function to be called to process the XML operation.
2. `operation` defines the XML operation.
3. `inputVar` defines the variable defining the input parameters of the operation (or NULL if there is none).

Usage

This method creates a One-Way RPC Style `com.WebOperation` object, where `function` is the name of the program function that is executed to process the XML operation.

The function name must be a string literal, not a string variable, due to operation publication restrictions.

There is no output parameter to be returned to the client.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

com.WebOperation.initiateSession

Defines the Web Operation as session initiator.

Syntax

```java
initiateSession( 
  ok INTEGER 
)
```

1. `ok` must have an integer value of 1 to define a session initiator.

Usage

Pass the parameter with the value of 1 to `initiateSession()` in order to define the current Web Operation as a session initiator.

A new session must be instantiated in this operation, and must be returned to the client via the `state` variable defined at service creation.

This method works only for stateful web services.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

com.WebOperation.setComment
Sets the comment for the Web Operation object.

**Syntax**

```java
setComment(
    comment STRING )
```

1. *comment* defines the comment to be set.

**Usage**

The `setComment()` method defines a comment for the current Web Operation object. The comment will appear in the WSDL of the service.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (`STATUS`) on page 3554.

**com.WebOperation.setInputAction**

Sets the WS-Addressing action identifier of the input operation.

**Syntax**

```java
setInputAction(
    action STRING )
```

1. *action* defines the WSA action identifier.

**Usage**

When WS-Addressing is enabled, this method defines the WS-Addressing action identifier of the input operation.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (`STATUS`) on page 3554.

**com.WebOperation.setInputEncoded**

Defines the encoding mechanism for Web Operation input parameters.

**Syntax**

```java
setInputEncoded(
    val INTEGER )
```

1. *val* defines an integer value defining the encoding mechanism to be used.

**Usage**

When the parameter `encoded` is true (1), the SOAP Section 5 encoding mechanism is used. False (0) indicates the XML Schema mechanism.

The XML Schema mechanism (FALSE) is not recommended.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (`STATUS`) on page 3554.

**com.WebOperation.setOutputAction**
Sets the WS-Addressing action identifier of the output operation.

**Syntax**

```java
setOutputAction(
    action STRING )
```

1. `action` defines the WSA action identifier.

**Usage**

When WS-Addressing is enabled, this method defines the WS-Addressing action identifier of the output operation. In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**com.WebOperation.setOutputEncoded**

Defines the encoding mechanism for Web Operation output parameters.

**Syntax**

```java
setOutputEncoded(
    val INTEGER  )
```

1. `val` defines an integer value defining the encoding mechanism to be used.

**Usage**

When the parameter `encoded` is true (1), the SOAP Section 5 encoding mechanism is used. False (0) indicates the XML Schema mechanism.

The XML Schema mechanism (FALSE) is not recommended.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**The WebServiceEngine class**

The `com.WebServiceEngine` class provides an interface to manage the Web Services engine.

**Important**: This Web Services class is not supported on GMI mobile devices.
com.WebServiceEngine methods
Methods for the com.WebServiceEngine class.
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>com.WebServiceEngine.Flush()</code></td>
<td>Forces the Web Service engine to immediately flush the response of the web service operation.</td>
</tr>
<tr>
<td><code>com.WebServiceEngine.HandleRequest(timeout INTEGER, status RECORD)</code></td>
<td>Get a handle for an incoming HTTP service request.</td>
</tr>
<tr>
<td><code>com.WebServiceEngine.ProcessServices(timeout INTEGER)</code></td>
<td>Specifies the wait period for an HTTP input request, to process an operation of one of the registered Web Services.</td>
</tr>
<tr>
<td><code>com.WebServiceEngine.RegisterRestService on page 2835(module STRING, basePath STRING)</code></td>
<td>Registers a REST service in the engine.</td>
</tr>
<tr>
<td><code>com.WebServiceEngine.SetFaultCode(code STRING, ns STRING)</code></td>
<td>Get a handle for an incoming HTTP service request.</td>
</tr>
<tr>
<td><code>com.WebServiceEngine.SetOption(optionName STRING, optionValue STRING)</code></td>
<td>Sets an option for the Web Service engine.</td>
</tr>
</tbody>
</table>
com.WebServiceEngine.Flush

Forces the Web Service engine to immediately flush the response of the web service operation.

Syntax

```markdown
com.WebServiceEngine.Flush()
RETURNS INTEGER
```

Usage

The `com.WebServiceEngine.flush()` class method allows for the return of the response inside a high-level web service operation, before the end of the web service function.

When this method is used, any other web operation output parameter changes are ignored.

The `status` returned by the method provides information about the execution of the last web operation. A return status of zero means OK. For a complete list of error codes, see Error codes of `com.WebServicesEngine` on page 2843

**Note:** The return status of the `com.WebServiceEngine.flush()` method has the same meaning as a status returned by `com.WebServiceEngine.ProcessServices()`, with the additional status code -32, meaning that the flush method was called outside of a web operation execution context.

**Note:** `com.WebServiceEngine.ProcessServices()` and `com.WebServiceEngine.HandleRequest()` can return the status code of -31, meaning that the flush function has been called in the last executed web operation.

Example

In this code example, the `flush()` method is used to force the response of the web service operation.

```ada
DEFINE echoBoolean_in, echoBoolean_out RECORD
  a_boolean BOOLEAN ATTRIBUTES(XMLName="Boolean")
END RECORD

MAIN
  DEFINE ret INTEGER
  ...
  WHILE true
    LET ret = com.WebServiceEngine.ProcessServices(-1)
    CASE ret
      WHEN 0
        DISPLAY "Request automatically processed."
      WHEN -31
        DISPLAY "Operation has been flushed."
      ...
    END CASE
  END WHILE
  ...
END MAIN

FUNCTION echoBoolean()
  DEFINE ret INTEGER
  -- Assign output parameter with input parameter
  LET echoBoolean_out.a_boolean = echoBoolean_in.a_boolean
  -- Immediate flush of web operation
  LET ret = com.WebServiceEngine.flush()
  IF ret != 0 THEN
    DISPLAY "ERROR Code : ",ret
    EXIT PROGRAM (1)
  END IF
  -- Changing the output parameters after flush() would have no effect.
END FUNCTION
```
**Related concepts**

- `com.WebServiceEngine.HandleRequest` on page 2833
  - Get a handle for an incoming HTTP service request.

- `com.WebServiceEngine.ProcessServices` on page 2834
  - Specifies the wait period for an HTTP input request, to process an operation of one of the registered Web Services.

**com.WebServiceEngine.GetHTTPServiceRequest**

- Get a handle for an incoming HTTP service request.

**Syntax**

```java
com.WebServiceEngine.GetHTTPServiceRequest (timeout INTEGER)
RETURNS com.HTTPServiceRequest
```

1. `timeout` defines the timeout in seconds.

**Usage**

The `com.WebServiceEngine.GetHTTPServiceRequest()` class method returns a `com.HTTPServiceRequest` object to handle an incoming HTTP request, or `NULL` if there was no request during the given period of time.

The `timeout` parameter defines the time in seconds to wait for an incoming request. A value of `-1` means infinite wait. When the timeout occurs, the method returns `NULL`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See [Error handling in GWS calls (STATUS)](page 3554).

Any new call to this function will raise an error until the previous HTTP request is handled by sending a response back to the client, or destroyed.

The error `-15565` can be thrown if an invalid or unsupported HTTP request is sent. Supported methods are GET, PUT, POST, HEAD, and DELETE.

Even if the exception is trapped, the GWS HTTP server is not in a valid state anymore, and you must close the application properly before exiting the program.

**Note:** This is not an issue in production environments as the Genero Application Server (GAS) and GWSProxy will detect the ended DVM, return an HTTP error code to the client app, and any new request will start a new DVM in a clean state via GWS proxy and the pool configuration.

The error `-15575` can be thrown if the GAS disconnects the Web Services program.

URLs are sent in UTF-8 on the network, if the web services server is not able to convert UTF-8 URLs back to fglrun locale charset, error `-15552` will be thrown. As a general advice, run you WS server program in UTF-8.

**Example**

```java
DEFINE req com.HTTPServiceRequest
...
TRY
  WHILE TRUE
    LET req = com.WebServiceEngine.getHTTPServiceRequest(-1)
    IF req IS NULL THEN
      DISPLAY "HTTP request timeout...: ", CURRENT YEAR TO FRACTION
    ELSE
      CALL req.sendTextResponse(200,NULL,"It works")
    END IF
  END WHILE
CATCH
```
IF status == -15575 THEN
  DISPLAY "Disconnected : ",SQLCA.SQLERRM
ELSE
  DISPLAY "ERROR : ",status,SQLCA.SQLERRM
END IF
END TRY

Related concepts

com.WebServiceEngine.ProcessServices on page 2834
Specifies the wait period for an HTTP input request, to process an operation of one of the registered Web Services.

com.WebServiceEngine.HandleRequest on page 2833
Get a handle for an incoming HTTP service request.

com.WebServiceEngine.GetOption
Returns the value of a Web Service engine option.

Syntax

com.WebServiceEngine.GetOption (  
  option STRING  )  
RETURNS STRING

1.  option defines the option to be queried.

Usage

The com.WebServiceEngine.GetOption() class method returns the current value of the given Web Services engine option.

See WebServiceEngine options on page 2840 for the supported options.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

com.WebServiceEngine.HandleRequest
Get a handle for an incoming HTTP service request.

Syntax

com.WebServiceEngine.HandleRequest (  
  timeout INTEGER,  
  status RECORD  )  
RETURNS com.HTTPServiceRequest

1.  timeout defines the timeout in seconds.
2.  status defines the variable receiving the method execution status.

Usage

The com.WebServiceEngine.HandleRequest() class method returns a com.HTTPServiceRequest object to handle an incoming HTTP request, or NULL if there was no request during the given period of time.

The timeout parameter defines the time in seconds to wait for an incoming request. A value of -1 means infinite wait.

• If there is no request in the given period of time, or if there is an error, the status code is updated by reference, and a NULL object is returned.
• If the request is intended for a registered web service, it is processed automatically. The status code is updated by reference and a NULL object is returned.
• If the request isn't dedicated to a registered web service, a status code of value 1 is returned by reference, and a valid instance of a `com.HTTPServiceRequest` object, immediately usable to handle the incoming request, is returned.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

Any new call to this function will raise an error until the previous HTTP request was handled by sending a response back to the client, or destroyed.

The `status` returned by the method provides information about the execution of the last web operation. A return status of zero means OK. For a complete list of error codes, see Error codes of `com.WebServicesEngine` on page 2843.

A status of -8 or -10 is returned if an invalid or unsupported HTTP request is sent. Supported methods are GET, PUT, POST, HEAD, and DELETE.

Even if the exception is trapped, the GWS HTTP server is not in a valid state anymore, and you must close the application properly before exiting the program.

**Note:** This is not an issue in production environments as the Genero Application Server (GAS) and GWSProxy will detect the ended DVM, return an HTTP error code to the client app, and any new request will start a new DVM in a clean state via GWS proxy and the pool configuration.

**Note:** If the `com.WebServiceEngine.Flush()` method is used, the return status handling must be done in the web operation function, while `com.WebServiceEngine.HandleRequest()` will return the code -31, to indicated that a flush was done.

The error -15575 can be thrown if the GAS disconnects the Web Services program.

URLs are sent in UTF-8 on the network, if the web services server is not able to convert UTF-8 URLs back to fglrun locale charset, error -15552 will be thrown. As a general advice, run you WS server program in UTF-8.

**Related concepts**

- `com.WebServiceEngine.ProcessServices` on page 2834
  Specifies the wait period for an HTTP input request, to process an operation of one of the registered Web Services.

- `com.WebServiceEngine.GetHTTPServiceRequest` on page 2832
  Get a handle for an incoming HTTP service request.

- `com.WebServiceEngine.ProcessServices` specifies the wait period for an HTTP input request, to process an operation of one of the registered Web Services.

**Syntax**

```c
    timeout INTEGER
) RETURNS INTEGER
```

1. `timeout` defines the timeout in seconds.

**Usage**

The `com.WebServiceEngine.ProcessServices()` class method specifies the wait period for an HTTP input request, to process an operation of one of the registered Web Services.

The `timeout` parameter defines the wait period for an HTTP input request, to process an operation of one of the registered Web Services. The value -1 specifies an infinite waiting time.

The `status` returned by the method provides information about the execution of the last web operation. A return status of zero means OK. For a complete list of error codes, see Error codes of `com.WebServicesEngine` on page 2843.

A status of -8 or -10 is returned if an invalid or unsupported HTTP request is sent.
Even if the exception is trapped, the GWS HTTP server is not in a valid state anymore, and you must close the application properly before exiting the program.

**Note:** This is not an issue in production environments as the Genero Application Server (GAS) and GWSProxy will detect the ended DVM, return an HTTP error code to the client app, and any new request will start a new DVM in a clean state via GWS proxy and the pool configuration.

The execution status is typically handled in a `CASE / END CASE` block, to treat all possible execution cases. For a complete example of execution status handling, see [Process the requests](#) on page 3351.

**Note:** If the `com.WebServiceEngine.Flush()` method is used, the return status handling must be done in the web operation function, while `com.WebServiceEngine.ProcessServices()` will return the code `-31`, to indicated that a flush was done.

**Related concepts**

- `com.WebServiceEngine.HandleRequest` on page 2833
  Get a handle for an incoming HTTP service request.

- `com.WebServiceEngine.GetHTTPServiceRequest` on page 2832
  Get a handle for an incoming HTTP service request.

- `com.WebServiceEngine.RegisterService`
  Registers a service in the engine.

  **Syntax**

  ```java
  com.WebServiceEngine.RegisterService(
      ws com.WebService 
  )
  ```

  1. `ws` defines the name of the service object to register.

  **Usage**

  The `com.WebServiceEngine.RegisterService()` class method registers the `com.WebService` object passed as parameter.

  In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See [Error handling in GWS calls (STATUS)](#) on page 3554.

- `com.WebServiceEngine.RegisterRestService`
  Registers a REST service in the engine.

  **Syntax**

  ```java
  com.WebServiceEngine.RegisterRestService on page 2835(
      module STRING
      basePath STRING
  )
  ```

  1. `module` registers the module of the REST service.

  2. `basePath` defines the base path to the Web service as it appears in the URL.

  **Usage**

  You use this method to register all the public functions in the `module` of your Web service with the Genero Web Service engine and publish them for users on the internet.

  It is recommended to import the Rest service module with the `IMPORT FGL` statement. Alternatively, you can link `.42r` files using the `fgllink` command.
The `basePath` sets the base path to the Web service in the Genero Web Service engine as it appears in the URL for the service. For instance, if you want your 4GL module called "accounts" to be published as a Rest service with the name "Accounts" (uppercase "A") you must call the function as shown:

```java
CALL com.WebServiceEngine.RegisterRestService("accounts","Accounts")
```

Then the URL is: `http://host:port/gas/ws/r/group/xcf/Accounts`

**Note:** Typically, you deploy the service behind a Genero Application Server, and therefore you have also the `group` name (if not the default) and the service `xcf` file name in the URL.

In direct mode on a localhost, the URL is just: `http://localhost:port/ws/r/Accounts`

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See [Error handling in GWS calls](STATUS) on page 3554.

**com.WebServiceEngine.SetFaultCode**

Get a handle for an incoming HTTP service request.

**Syntax**

```java
com.WebServiceEngine.SetFaultCode(
    code STRING,
    ns STRING
)
```

1. `code` defines the fault code.
2. `ns` defines the namespace of the fault code.

**Usage**

The `com.WebServiceEngine.SetFaultCode()` class method defines a user SOAP Fault code to be returned to the client, where `code` is the mandatory SOAP Fault code and `ns` is the mandatory code namespace.

This method must be called inside a Web Service operation.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See [Error handling in GWS calls](STATUS) on page 3554.

**com.WebServiceEngine.SetFaultDetail**

Defines the published SOAP Fault.

**Syntax**

```java
com.WebServiceEngine.SetFaultDetail(
    fault RECORD
)
```

1. `fault` defines the published fault.

**Usage**

The `com.WebServiceEngine.SetFaultDetail()` class method defines the published SOAP Fault to be returned to the client when operation has finished, where `fault` is one of the published variables defined as Fault for that operation.

This method must be called inside a Web Service operation.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See [Error handling in GWS calls](STATUS) on page 3554.
com.WebServiceEngine.SetFaultString
Defines the description of a SOAP Fault.

Syntax

```java
com.WebServiceEngine.SetFaultString(
    str STRING )
```

1. `str` defines the description of the fault.

Usage
The `com.WebServiceEngine.SetFaultString()` class method defines a user SOAP Fault description to be returned to the client, where `str` contains the description of the fault.

This method must be called inside a Web Service function.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (`STATUS`) on page 3554.

com.WebServiceEngine.SetOption
Sets an option for the Web Service engine.

Syntax

```java
com.WebServiceEngine.SetOption(
    optionName STRING,
    optionValue STRING )
```

1. `optionName` defines the option to set.
2. `optionValue` defines the value of the option to set.

Usage
The `com.WebServiceEngine.SetOption()` class method configures the Web Services engine with options. See WebServiceEngine options on page 2840 for the supported options.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (`STATUS`) on page 3554.

com.WebServiceEngine.SetRestError
Manages error handling for a REST high-level Web Service function.

Syntax

```java
com.WebServiceEngine.SetRestError on page 2837(
    code INTEGER
    error RECORD )
```

1. `code` defines a standard HTTP response status code (400 - 599) as defined in RFC 2616 Section 10.4 and RFC 2616 Section 10.5
2. `error` defines details of the error. It can reference a variable of a suitable Genero BDL type; for example, simple type, or record. It can return a NULL value.
**Usage**

You use this method to set the HTTP status code and details of the error to be returned to the client when the request to the resource has finished. `SetRestError` accepts the HTTP code, a Genero BDL variable (if any), or a text description to be returned. This method must be called inside a Web Service REST high-level function.

If the `SetRestError()` method `error` parameter references a variable, the variable must be specified in the `WSThrows` on page 3484 declaration of the function, otherwise the method fails and returns an error code of -15570.

Variables referenced by `WSThrows`, must be defined as `PUBLIC` at the modular level and specified with the `WSError` on page 3483 attribute.

If `WSThrows` does not reference a variable, just a message string (for example "404:my error message"), you must call the method with the NULL value (`SetRestError(404,NULL)`) to send the "my error message" as the HTTP reason phrase. Ultimately, this will depend on whether the web server allows you to change the standard protocol message or not.

It is recommended that if you need to ensure an error description other than the standard protocol message is returned to the client, you must use `WSThrows` with "code:@variable" option and call the method with a reference to the variable, for example `SetRestError(400,myError)`.

In the examples Example 1: managing expected errors on page 2838 and Example 2: managing unexpected errors on page 2838 different uses of the method are shown.

**Example 1: managing expected errors**

In this example, common application level type errors, like resource not found, are expected. Instead of the standard HTTP status code (400 bad request), the call to `SetRestError(400,myerror)` will send the description set on the `myError.message` variable as the reason for the error.

```plaintext
PUBLIC DEFINE myError RECORD ATTRIBUTE(WSError="user error")
message STRING
END RECORD

PUBLIC
FUNCTION queryAccountById(id VARCHAR(10) ATTRIBUTE(WSParam))

  ATTRIBUTES(WSGet,
  WSPATH="/accounts/{id}",
  WSThrows="400:@myError,500:Internal Server Error")
  RETURNS accountType ATTRIBUTES(WSName="body")

  DEFINE thisAccount RECORD LIKE accounts.*
  # ... retrieve user id from database ...
  CASE sqlca.sqlcode
  WHEN 0
    EXIT CASE
  WHEN NOTFOUND
    LET myError.message = SFMT("Could not find account id :%1",id)
    CALL com.WebServiceEngine.SetRestError(400,myError)
  OTHERWISE
    CALL com.WebServiceEngine.SetRestError(500,NULL)
  END CASE
  RETURN thisAccount.*
END FUNCTION
```

**Example 2: managing unexpected errors**

**Warning:** In general, the recommended option is to list all the expected errors as shown in Example 1, in order to get them generated in the OpenAPI specification file and trapped on the client side.
In this example you code to trap a status code not defined in the WSThrows list. In the call to the SetRestError() method the status code and optionally the error variable is returned.

PUBLIC DEFINE myError RECORD ATTRIBUTE(WSError="user error")
   message STRING
END RECORD

PUBLIC
FUNCTION queryAccountById(id VARCHAR(10) ATTRIBUTE(WSParam))
   ATTRIBUTES(WSGet,
      WSPath="/accounts/{id}",
      WSThrows="400:@myError,500:Internal Server Error")
   RETURNS accountType ATTRIBUTES(WSName="body")

DEFINE thisAccount RECORD LIKE accounts.*
# ... retrieve user id from database ...

CASE sqlca.sqlcode
WHEN 0 # success
   EXIT CASE
WHEN NOTFOUND
   LET myError.message = SFMT("Could not find account id :%1",id)
   CALL com.WebServiceEngine.SetRestError(400,myError)
WHEN < 0
   CALL com.WebServiceEngine.SetRestError(500,NULL)
otherwise
   CALL com.WebServiceEngine.SetRestError(505,NULL)
END CASE
RETURN thisAccount.*
END FUNCTION

Related concepts
Handling application level errors on page 3399
There are many situations in which you need to notify an error to a client using your REST Web service.

Define functions in a module on page 3376
A GWS REST service is defined in a single module.

Related reference
High-level RESTful Web service attributes on page 3463

com.WebServiceEngine.Start
Starts the Web Service engine.

Syntax

com.WebServiceEngine.Start()

Usage
The com.WebServiceEngine.Start() class method starts the engine for all registered Web Services.

If you run the Web Services server program in standalone mode, port 80 is used unless FGLAPPSERVER is set. We recommend that you set FGLAPPSERVER, as port 80 may already be in use by other Web servers. If you run the Web Services server program through the Genero Application Server, the FGLAPPSERVER variable is automatically set by the Genero Application Server. Do NOT manually set FGLAPPSERVER in this case.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.
Related concepts
FGLAPPSERVER on page 274
Defines the listening TCP port of the Web service in development context.

WSDL generation options notes
If you are planning to work with WSDL generation, you are advised to review these notes.

1. For the DECIMAL(5, 2) data type, when wsdl_decimalsize is TRUE, the generated WSDL file contains the total size and the size of the fractional part of the decimal:

```xml
<types>
    <schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.mycompany.com/types/">
        <simpleType name="echoDecimal5_2_a_dec5_2_out_FGLDecimal">
            <restriction base="decimal">
                <totalDigits value="5" />
                <fractionDigits value="2" />
            </restriction>
        </simpleType>
    </schema>
</types>
```

When wsdl_decimalsize is FALSE, the total size and the size of the fractional part are not mentioned:

```xml
<message name="echoDecimal5_2">
    <part name="dec5_2" type="f:echoDecimal5_2_a_dec5_2_in_FGLDecimal" />
</message>
```

2. If the WSDL file does not contain the size, the client application has no way of knowing the size. In this case, a default value for the size is generated. For example, the exported server type DECIMAL(5, 2) becomes a DECIMAL(32) on the client side.

3. It is better to keep the options wsdl_arraysize, wsdl_stringsize and wsdl_decimalsize set to TRUE so that the client program can do exact type mapping. The default for all three options is TRUE.

4. When setting a facet constraint attribute on a simple data type, the generation of the WSDL will take this attribute into account even if an option has been set to perform the opposite.

5. When setting one facet constraint attribute, all of the default constraint attributes won’t be generated anymore unless you specify them as facet constraint attributes.

WebServiceEngine options

Table 606: Options for the com.WebServiceEngine

<table>
<thead>
<tr>
<th>Flag</th>
<th>Client or Server</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>autocookiesmanagement</td>
<td>Client</td>
<td>If set to TRUE, it activates the automatic cookies management for any HTTPRequest. It is the same as calling <code>req.setAutoCookies(TRUE)</code> for all HTTPRequest to a single instance of fglrun. By setting to TRUE, the program does not need to call <code>req.setAutoCookies(true)</code> on any HTTPRequest in its code. The global option is called once at the start of the program, and all cookies received from a server are automatically sent back according to the cookie definition (path, expiration date, domain). Default value is FALSE.</td>
</tr>
<tr>
<td>Flag</td>
<td>Client or Server</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>connectiontimeout</td>
<td>Client</td>
<td>Defines the default maximum time in seconds a client, an HTTPRequest and a TCPRequest have to wait for the establishment of a connection with a server.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A value of -1 means infinite wait.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The default is 30 seconds for non-Windows, 5 seconds for Windows®.</td>
</tr>
<tr>
<td>http_invoketimeout</td>
<td>Client</td>
<td>Defines the default maximum time in seconds a client has to wait before the client connection raises an error because the server is not responding.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A value of -1 means that it has to wait until the server responds.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The default is -1.</td>
</tr>
<tr>
<td>http_invoketimeout</td>
<td>Client</td>
<td>Important: This feature is deprecated, its use is discouraged although not prohibited.</td>
</tr>
<tr>
<td>maximumpersistentcookies</td>
<td>Client</td>
<td>Specify the maximum number of persistent cookies that can be handled by a single fglrun process. If the limit is reached, the older cookies will be deleted to create room for the newer cookies.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This option applies to the client side only.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default value is 50.</td>
</tr>
<tr>
<td>maximumresponselength</td>
<td>Both</td>
<td>Defines the maximum authorized size in KBytes for a client, server, HTTP or TCP response, before a break (when it stops and returns from the function because the amount of data surpassed the maximumresponselength.)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A value of -1 means no limit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The default is -1.</td>
</tr>
<tr>
<td>readwritetimeout</td>
<td>Client</td>
<td>Defines the default maximum time in seconds a client, a HTTP request/response and a TCP request/response have to wait before raising an error that the server doesn't return or accept data.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A value of -1 means infinite wait.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The default is -1.</td>
</tr>
<tr>
<td>server_readwritetimeout</td>
<td>Server</td>
<td>Defines how long a socket read or write operation can wait before raising an error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The default value is 5 seconds.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Note: Before this option was introduced, the default value was -1 (infinite) and was configurable with the accept timeout parameter via ProcessServices() method.</td>
</tr>
<tr>
<td>Flag</td>
<td>Client or Server</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>server_restdefaultformat</td>
<td>Server</td>
<td>Defines runtime support for mime types in REST operations. By default the supported formats are XML and JSON. If the mime type is not set by the WSMedia attribute on the record for instance, by calling com.webserviceEngine.SetOption(&quot;server_restdefaultformat&quot;, &quot;json&quot;) you can specify the XML format at runtime. Other options are &quot;json&quot; or &quot;both&quot;. Then you can also retrieve the value with GetOption.</td>
</tr>
<tr>
<td>SoapModuleURI</td>
<td>Both</td>
<td>Defines the SOAP role of a Genero application with an URI to identify it along a SOAP message path. The default value is NULL.</td>
</tr>
<tr>
<td>tcp_connectiontimeout</td>
<td>Client</td>
<td>Defines the default maximum time in seconds a client has to wait for the establishment of a TCP connection with a server. A value of -1 means infinite wait. The default is 30 seconds for non-Windows, 5 seconds for Windows®. Important: Deprecated - use connectiontimeout instead.</td>
</tr>
<tr>
<td>wsdl_arraysize</td>
<td>Server</td>
<td>Defines whether the size of a BDL array will be taken into account during the WSDL generation. See WSDL generation options notes on page 2840. A value of zero means FALSE. The default is TRUE.</td>
</tr>
<tr>
<td>wsdl_decimalsize</td>
<td>Server</td>
<td>Defines whether the precision and scale of a DECIMAL variable will be taken into account during the WSDL generation. See WSDL generation options notes on page 2840. A value of zero means FALSE. The default is TRUE.</td>
</tr>
<tr>
<td>wsdl_stringsize</td>
<td>Server</td>
<td>Defines whether the size of a CHAR or VARCHAR variable will be taken into account during the WSDL generation. See WSDL generation options notes on page 2840. A value of zero means FALSE. The default is TRUE.</td>
</tr>
</tbody>
</table>

**Related concepts**

- com.WebServiceEngine.GetOption on page 2833
  Returns the value of a Web Service engine option.
- com.WebServiceEngine.SetOption on page 2837
Sets an option for the Web Service engine.

**Error codes of com.WebServicesEngine**
Error codes returned by com.WebServiceEngine methods.

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Timeout</td>
</tr>
<tr>
<td></td>
<td>com.WebServiceEngine.ProcessServices(x) timeout is reached. No requests to process during x seconds.</td>
</tr>
<tr>
<td>-2</td>
<td>AsCloseCommand</td>
</tr>
<tr>
<td></td>
<td>GAS tells the DVM to shutdown. You must exit your application.</td>
</tr>
<tr>
<td>-3</td>
<td>ConnectionBroken</td>
</tr>
<tr>
<td></td>
<td>Client has closed the connection in standalone GWS (without GAS).</td>
</tr>
<tr>
<td>-4</td>
<td>ConnectionInterrupted</td>
</tr>
<tr>
<td></td>
<td>Ctrl-C received. Interruption received by DVM. You must exit your application.</td>
</tr>
<tr>
<td>-5</td>
<td>BadHTTPHeader</td>
</tr>
<tr>
<td></td>
<td>Check the message with FGLWSDEBUG or display SQLCA.SQLERRM.</td>
</tr>
<tr>
<td>-6</td>
<td>MalformedSOAPEnvelope</td>
</tr>
<tr>
<td></td>
<td>Check the message with FGLWSDEBUG or display SQLCA.SQLERRM.</td>
</tr>
<tr>
<td>-7</td>
<td>MalformedXMLDocument</td>
</tr>
<tr>
<td></td>
<td>Check the message with FGLWSDEBUG or display SQLCA.SQLERRM.</td>
</tr>
<tr>
<td>-8</td>
<td>Internal HTTP Error</td>
</tr>
<tr>
<td></td>
<td>There is a communication issue with application server or client. Check with FGLWSDEBUG.</td>
</tr>
<tr>
<td>-9</td>
<td>Unsupported operation</td>
</tr>
<tr>
<td></td>
<td>The URL of the operation requested is unknown. Check the message with FGLWSDEBUG or display SQLCA.SQLERRM.</td>
</tr>
<tr>
<td>-10</td>
<td>Unknown Error</td>
</tr>
<tr>
<td></td>
<td>This is an internal error, contact the support team. You must exit your application.</td>
</tr>
<tr>
<td>-11</td>
<td>WSDL generation failed</td>
</tr>
<tr>
<td></td>
<td>You need to debug your application.</td>
</tr>
<tr>
<td>-12</td>
<td>WSDL Service not found</td>
</tr>
<tr>
<td></td>
<td>Check the message with FGLWSDEBUG or display SQLCA.SQLERRM.</td>
</tr>
<tr>
<td>-13</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>No need to exit the application. A new request might not have the issue.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| -14    | Incoming request overflow  
         You exceeded the data maximum length allowed by  
| -15    | Server was not started  
         Call to `com.WebServiceEngine.Start()` failed. You must exit your application. |
| -16    | Request still in progress  
         With RESTful service, this error occurs when you are currently processing a request that has  
         not yet send the response and you try to process another request. You need to debug your  
         application. It depends, you might not need to stop your application. |
| -17    | Stax response error  
         You need to debug your application. Check the message with FGLWSDEBUG or display  
         `SQLCA.SQLERRM`. |
| -18    | Input request handler error  
         You need to debug your application. Check the message with FGLWSDEBUG or display  
         `SQLCA.SQLERRM`. |
| -19    | Output request handler error  
         You need to debug your application. Check the message with FGLWSDEBUG or display  
         `SQLCA.SQLERRM`. |
| -20    | WSDL handler error  
         You need to debug your application. Check the message with FGLWSDEBUG or display  
         `SQLCA.SQLERRM`. |
| -21    | SOAP Version mismatch  
         Your client SOAP version does not match your server SOAP version, amend either your client  
         or your server code. |
| -22    | SOAP header not understood  
         Modify your server code to handle the `mustUnderstand` attribute. Use the incoming request  
         handler. |
| -23    | Deserialization error  
         Check the message with FGLWSDEBUG or display `SQLCA.SQLERRM`. |
| -24    | Reserved error code  
         This error code is reserved for future use. |
| -25    | Web Services Addressing action is mandatory  
         Check that the WSA action is specified in the SOAP message. |
| -26    | Web Services Addressing message header is invalid  
         Check that the WSA header is correct in the SOAP message. |
| -27    | Web Services Addressing message header is mandatory  
         Check that the WSA header is specified in the SOAP message. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -28    | Web Services Addressing message protocol does not match  
Check that the WSA message uses the protocol version of the client matches the version expected by the server. |
| -29    | Cookie error  
Check that the HTTP request contains a valid cookie. |
| -30    | No active web operation  
The method was called outside the context of a web operation process. |
| -31    | Web Operation was flushed  
This code is returned by the `ProcessServices()` or the `HandlerRequest()` method, to indicated that the `Flush()` method was called during the last web operation execution. |
| -32    | Serialization error  
Check the message with FGLWSDEBUG or display SQLCA.SQLERRM. |
| -33    | XOP Not Supported  
The Web service does not implement the XML-binary Optimized Packaging (XOP) packaging specification for serializing binary data. |
| -34    | XOP Error  
Handing of XML-Binary Optimized Packaging failed, check the error with FGLWSDEBUG. |
| -35    | No Matching Rest Operation Found  
Check the URL of the resource and the REST web service function parameters. |
| -36    | Unexpected Rest Parameter  
Check that the number and type of input and return parameters match the service function. |
| -37    | WADL Error  
Generation of Web Application Description Language (WADL) failed, check error with FGLWSDEBUG. |
| -38    | Open API Error  
Generation of OpenAPI specification failed, check error with FGLWSDEBUG. |
| -39    | Content Type Ctx Incompatible  
Runtime Content-Type set via `WSContext` does not match the `WSMedia` of the REST operation, check `WSMedia` and `WSContext` values. |
| -40    | Scope Missing  
Service is secured, and a user token is required to authenticate access. Check that you are providing a scope. |

**The HTTPServiceRequest class**

The `com.HTTPServiceRequest` class provides an interface to process incoming XML and TEXT requests over HTTP on the server side, with an access to the HTTP layer and additional XML streaming possibilities.

**Important:** This Web Services class is not supported on GMI mobile devices.
**Table 608: Object methods: Request URLs**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getURL()</code></td>
<td>Returns the URL of the HTTP service request. The URL may consists of a host name, a port number, a path, and a query.</td>
</tr>
<tr>
<td><code>getURLHost()</code></td>
<td>Returns the host name contained in the URL of an HTTP service request.</td>
</tr>
<tr>
<td><code>getURLPath()</code></td>
<td>Returns the path contained in the URL of an HTTP service request.</td>
</tr>
<tr>
<td><code>getURLPort()</code></td>
<td>Returns the port number contained in the URL of an HTTP service request.</td>
</tr>
<tr>
<td><code>getURLQuery(query RECORD)</code></td>
<td>Takes a dynamic array of RECORD of two strings and fills that array with the decoded query string of an HTTP service request.</td>
</tr>
</tbody>
</table>

**Table 609: Object methods: Request headers**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getRequestHeader(name STRING)</code></td>
<td>Returns the value of an HTTP header.</td>
</tr>
<tr>
<td><code>getRequestHeaderCount()</code></td>
<td>Returns the number of request headers.</td>
</tr>
<tr>
<td><code>getRequestHeaderName(ind INTEGER)</code></td>
<td>Returns a request header name by position.</td>
</tr>
<tr>
<td><code>getRequestHeaderValue(ind INTEGER)</code></td>
<td>Returns a request header value by position.</td>
</tr>
<tr>
<td><code>getRequestVersion()</code></td>
<td>Returns the HTTP version of the service request.</td>
</tr>
</tbody>
</table>
### Table 610: Object methods: Request body

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>beginXmlRequest()</code></td>
<td>Starts an HTTP streaming request.</td>
</tr>
<tr>
<td>RETURNS <code>xml.StaxReader</code></td>
<td></td>
</tr>
<tr>
<td><code>endXmlRequest(stax xml.StaxReader)</code></td>
<td>Terminates an HTTP streaming request.</td>
</tr>
<tr>
<td><code>getMethod()</code></td>
<td>Returns the HTTP method of the service request.</td>
</tr>
<tr>
<td>RETURNS <code>STRING</code></td>
<td></td>
</tr>
<tr>
<td><code>hasRequestKeepConnection()</code></td>
<td>Returns TRUE if the connection remains open after</td>
</tr>
<tr>
<td>RETURNS <code>INTEGER</code></td>
<td>sending a response.</td>
</tr>
<tr>
<td><code>readDataRequest(b BYTE)</code></td>
<td>Returns the body of a request in a BYTE.</td>
</tr>
<tr>
<td><code>readFileRequest()</code></td>
<td>Returns the body of a request in a file.</td>
</tr>
<tr>
<td>RETURNS <code>STRING</code></td>
<td></td>
</tr>
<tr>
<td><code>readFormEncodedRequest(utf8 INTEGER)</code></td>
<td>Returns the string of a GET request with UTF-8</td>
</tr>
<tr>
<td>RETURNS <code>STRING</code></td>
<td>conversion option.</td>
</tr>
<tr>
<td><code>readTextRequest()</code></td>
<td>Returns the request body as a plain string.</td>
</tr>
<tr>
<td>RETURNS <code>STRING</code></td>
<td></td>
</tr>
<tr>
<td><code>readXmlRequest()</code></td>
<td>Returns the request body as an XML document.</td>
</tr>
<tr>
<td>RETURNS <code>xml.DomDocument</code></td>
<td></td>
</tr>
</tbody>
</table>
## Table 611: Object methods: Request multipart

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getRequestMultipartType()</code></td>
<td>Returns the multipart type of an incoming request.</td>
</tr>
<tr>
<td><code>getRequestPart(pos INTEGER)</code></td>
<td>Returns the HTTPPart object at the specified index position.</td>
</tr>
<tr>
<td><code>getRequestPartCount()</code></td>
<td>Returns the number of additional multipart elements.</td>
</tr>
<tr>
<td><code>getRequestPartFromID(id STRING)</code></td>
<td>Returns the HTTPPart object of the given Content-ID value.</td>
</tr>
</tbody>
</table>

## Table 612: Object methods: Response header

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setResponseCharset(charset STRING)</code></td>
<td>Defines the HTTP response character set.</td>
</tr>
<tr>
<td><code>setResponseHeader(name STRING, value STRING)</code></td>
<td>Defines a header for the HTTP response.</td>
</tr>
<tr>
<td><code>setResponseVersion(version STRING)</code></td>
<td>Defines the HTTP response version.</td>
</tr>
</tbody>
</table>
### Table 613: Object methods: Response body

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>beginXmlResponse(code INTEGER, description STRING)</code></td>
<td>Starts an HTTP streaming response.</td>
</tr>
<tr>
<td><code>endXmlResponse(stax xml.StaxWriter)</code></td>
<td>Terminates an HTTP streaming response.</td>
</tr>
<tr>
<td><code>sendResponse(code INTEGER, description STRING)</code></td>
<td>Sends an HTTP response without body.</td>
</tr>
<tr>
<td><code>sendDataResponse(code INTEGER, description STRING, b BYTE)</code></td>
<td>Sends an HTTP response with data of a BYTE variable.</td>
</tr>
<tr>
<td><code>sendFileResponse(code INTEGER, description STRING, filename STRING)</code></td>
<td>Sends an HTTP response with the data contained in a file.</td>
</tr>
<tr>
<td><code>sendTextResponse(code INTEGER, description STRING, txt STRING)</code></td>
<td>Sends an HTTP response with data from a plain string.</td>
</tr>
</tbody>
</table>

### Table 614: Object methods: Response multipart

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addResponsePart(part com.HTTPPart)</code></td>
<td>Adds a new part to the HTTP root part response.</td>
</tr>
<tr>
<td><code>setResponseMultipartType(type STRING, start STRING, boundary STRING)</code></td>
<td>Sets HTTP response in multipart mode of given type.</td>
</tr>
</tbody>
</table>
Table 615: Object methods: Cookie management

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>findRequestCookie</code> (name STRING) RETURNS STRING</td>
<td>Enables the server to retrieve a cookie sent by the client by name.</td>
</tr>
<tr>
<td><code>setResponseCookies</code> (cookies RECORD)</td>
<td>Allows the server to return cookies to be set on the client application sending the request.</td>
</tr>
</tbody>
</table>

**com.HTTPServiceRequest.addResponsePart**

Adds a new part to the HTTP root part response.

**Syntax**

```java
addResponsePart(
    part com.HTTPPart
)
```

1. `part` defines `com.HTTPPart` used to manage the HTTP attachment.

**Usage**

Adds a new part to the HTTP root part response. It will be sent after root part has been processed.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**Related concepts**

Examples using `com.HTTPServiceRequest` methods on page 2870

These examples use methods of the `com.HTTPServiceRequest` class.

Examples: Using the `com.HTTPPart` class on page 2910

Examples using methods of the `com.HTTPPart` class.

**com.HTTPServiceRequest.beginXmlRequest**

Starts an HTTP streaming request.

**Syntax**

```java
beginXmlRequest() RETURNS xml.StaxReader
```

**Usage**

The `beginXmlRequest()` method starts the streaming of HTTP request and returns a `xml.StaxReader` object ready to read the XML from the client.

Supported methods are PUT and POST.

The request Content-Type header must be of the form */xml or */+xml. For example: application/xhtml+xml.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.
The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 3555

Related concepts
Examples using com.HTTPServiceRequest methods on page 2870
These examples use methods of the com.HTTPServiceRequest class.
Examples: Using the com.HTTPPart class on page 2910
Examples using methods of the com.HTTPPart class.

com.HTTPServiceRequest.beginTransaction
Starts an HTTP streaming response.

Syntax

```
beginTransaction(
    code INTEGER,
    description STRING )
RETURNS xml.StaxWriter
```

1. `code` defines the status code of the response.
2. `description` defines the description of the response.

Usage
The `beginTransaction()` method starts an HTTP streaming response by sending a status (`code`) and description (`desc`), followed by the headers previously set, and returns a `xml.StaxWriter` object ready to send XML as the HTTP body.

If the request fails to be read, its content will be discarded; for example, when a request is not well formatted.

The default Content-Type header is `text/xml`, but it can be changed if of the form `/*xml` or `*/*+xml`. For example: `application/xhtml+xml`.

In HTTP 1.1, if the body size is greater than 32k, the response will be sent in several chunks of the same size.

If the description is `NULL`, a default description based on the status code is sent.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 3555

Related concepts
Examples using com.HTTPServiceRequest methods on page 2870
These examples use methods of the com.HTTPServiceRequest class.
Examples: Using the com.HTTPPart class on page 2910
Examples using methods of the com.HTTPPart class.

com.HTTPServiceRequest.endRequest
Terminates an HTTP streaming request.

Syntax

```
endRequest(
    stax xml.StaxReader )
```

1. `stax` defines the `xml.StaxReader` object used for streaming.
Usage

The `endXmlRequest()` method ends the streaming HTTP request by closing the `xml.StaxReader` object passed as parameter.

The `stax` object must be created with the `beginXmlRequest()` method.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

The `INT_FLAG` variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 3555

Related concepts

Examples using `com.HTTPServiceRequest` methods on page 2870
These examples use methods of the `com.HTTPServiceRequest` class.

Examples: Using the `com.HTTPPart` class on page 2910
Examples using methods of the `com.HTTPPart` class.

com.HTTPServiceRequest.endXmlResponse

Terminates an HTTP streaming response.

Syntax

```
endXmlResponse(
    stax xml.StaxWriter
)
```

1. `stax` defines `xml.StaxWriter` used to write the HTTP body.

Usage

The `endXmlResponse()` method terminates the HTTP streaming response by closing the `xml.StaxWriter` object created by `beginXmlResponse`.

The body of the request is discarded.

New incoming requests can be retrieved again with the `com.WebServiceEngine.GetHTTPServiceRequest()` method.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

The `INT_FLAG` variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 3555

Related concepts

Examples using `com.HTTPServiceRequest` methods on page 2870
These examples use methods of the `com.HTTPServiceRequest` class.

Examples: Using the `com.HTTPPart` class on page 2910
Examples using methods of the `com.HTTPPart` class.

com.HTTPServiceRequest.findRequestCookie

Enables the server to retrieve a cookie sent by the client by name.

Syntax

```
findRequestCookie(
    name STRING
)
```
**Usage**

Enables the server to retrieve a cookie sent by the client by name.

The method returns the value of the cookie, or NULL if the named cookie does not exist.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**Related concepts**

- Examples using com.HTTPServiceRequest methods on page 2870
  These examples use methods of the com.HTTPServiceRequest class.
- Examples: Using the com.HTTPPart class on page 2910
  Examples using methods of the com.HTTPPart class.

com.HTTPServiceRequest.getMethod

Returns the HTTP method of the service request.

**Syntax**

```java
getMethod()
  RETURNS STRING
```

**Usage**

The `getMethod()` method returns the HTTP method of the request, Supported methods are GET, PUT, POST, HEAD, and DELETE.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**Related concepts**

- Examples using com.HTTPServiceRequest methods on page 2870
  These examples use methods of the com.HTTPServiceRequest class.
- Examples: Using the com.HTTPPart class on page 2910
  Examples using methods of the com.HTTPPart class.

com.HTTPServiceRequest.getRequestVersion

Returns the HTTP version of the service request.

**Syntax**

```java
getRequestVersion()
  RETURNS STRING
```

**Usage**

The `getRequestVersion()` method returns the HTTP version of the request (1.0 or 1.1).

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.
Related concepts

Examples using com.HTTPServiceRequest methods on page 2870
These examples use methods of the com.HTTPServiceRequest class.

Examples: Using the com.HTTPPart class on page 2910
Examples using methods of the com.HTTPPart class.

**com.HTTPServiceRequest.getRequestHeader**

Returns the value of an HTTP header.

**Syntax**

```
getRequestHeader (  
   name STRING  
)  
RETURNS STRING
```

1. *name* defines an HTTP header.

**Usage**

The `getRequestHeader()` method returns the value of the HTTP header specified by the *name* parameter, or NULL if there is none found.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

Related concepts

Examples using com.HTTPServiceRequest methods on page 2870
These examples use methods of the com.HTTPServiceRequest class.

Examples: Using the com.HTTPPart class on page 2910
Examples using methods of the com.HTTPPart class.

**com.HTTPServiceRequest.getRequestHeaderCount**

Returns the number of request headers.

**Syntax**

```
getRequestHeaderCount ()  
RETURNS INTEGER
```

**Usage**

The `getRequestHeaderCount()` method returns the entire URL request containing the host, port, document and query string.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

Related concepts

Examples using com.HTTPServiceRequest methods on page 2870
These examples use methods of the com.HTTPServiceRequest class.

Examples: Using the com.HTTPPart class on page 2910
Examples using methods of the com.HTTPPart class.

**com.HTTPServiceRequest.getRequestHeaderName**
Returns a request header name by position.

Syntax

```
getRequestHeaderName (  
    ind INTEGER  
)  
RETURNS STRING
```

1. `ind` defines the ordinal position of the header.

Usage

The `getRequestHeaderName()` method returns the name of the header at the given position.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

Related concepts

Examples using com.HTTPServiceRequest methods on page 2870
These examples use methods of the `com.HTTPServiceRequest` class.

Examples: Using the `com.HTTPPart` class on page 2910
Examples using methods of the `com.HTTPPart` class.

com.HTTPServiceRequest.getRequestHeaderValue
Returns a request header value by position.

Syntax

```
getRequestHeaderValue (  
    ind INTEGER  
)  
RETURNS STRING
```

1. `ind` defines the ordinal position of the header.

Usage

The `getRequestHeaderValue()` method returns the value of the header at the given position.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

Related concepts

Examples using com.HTTPServiceRequest methods on page 2870
These examples use methods of the `com.HTTPServiceRequest` class.

Examples: Using the `com.HTTPPart` class on page 2910
Examples using methods of the `com.HTTPPart` class.

com.HTTPServiceRequest.getRequestMultipartType
Returns the multipart type of an incoming request.

Syntax

```
getRequestMultipartType (  
)  
RETURNS STRING
```
Usage

Returns the multipart type of an incoming request, returns NULL if not a multipart request.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

Related concepts

Examples using com.HTTPServiceRequest methods on page 2870
These examples use methods of the com.HTTPServiceRequest class.

Examples: Using the com.HTTPPart class on page 2910
Examples using methods of the com.HTTPPart class.

com.HTTPServiceRequest.getRequestPart

Returns the HTTPPart object at the specified index position.

Syntax

```java
getRequestPart (  
    pos INTEGER)  
RETURNS com.HTTPPart
```

1. pos defines the index position.

Usage

Returns the HTTPPart object at the specified index position. The file part is stored in the temporary directory as defined by the DVM environment or the system. If the DVM DBTEMP on page 273 environment variable is set, this is used. Otherwise the temporary directory as defined by the system is used. On Windows® the system temporary directory is set by TEMP. On UNIX® the /tmp directory is used.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

Can raise error -15554 (Index is out of bounds).

Related concepts

Examples using com.HTTPServiceRequest methods on page 2870
These examples use methods of the com.HTTPServiceRequest class.

Examples: Using the com.HTTPPart class on page 2910
Examples using methods of the com.HTTPPart class.

com.HTTPServiceRequest.getRequestPartCount

Returns the number of additional multipart elements.

Syntax

```java
getRequestPartCount ()  
RETURNS INTEGER
```

Usage

The root multipart is handled via standard readTextRequest(), readXmlRequest(), readDataRequest() and beginXmlRequest().

The number of parts is only available when the entire request has been read.
In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**Related concepts**

Examples using com.HTTPServiceRequest methods on page 2870
These examples use methods of the com.HTTPServiceRequest class.

Examples: Using the com.HTTPPart class on page 2910
Examples using methods of the com.HTTPPart class.

**com.HTTPServiceRequest.getRequestPartFromID**

Returns the HTTPPart object of the given Content-ID value.

**Syntax**

```java
getRequestPartFromID(
    id STRING)
RETURNS com.HTTPPart
```

1. *id* defines the Content-ID value.

**Usage**

Returns the HTTPPart object of the given Content-ID value, returns NULL if there is none.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**Related concepts**

Examples using com.HTTPServiceRequest methods on page 2870
These examples use methods of the com.HTTPServiceRequest class.

Examples: Using the com.HTTPPart class on page 2910
Examples using methods of the com.HTTPPart class.

**com.HTTPServiceRequest.getURL**

Returns the URL of the HTTP service request. The URL may consists of a host name, a port number, a path, and a query.

**Syntax**

```java
getURL()
RETURNS STRING
```

**Usage**

The `getURL()` method returns the entire URL request containing the host, port, document and query string.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

URLs are sent in UTF-8 on the network. If the query part of the URL cannot be converted from UTF-8 to the fglrun locale charset, STATUS will be set to -15552. Is this case, the document part of the URL is available, but the query string must be retrieved through `HTTPServiceRequest.readFormEncodedRequest()`. As a general advice, run your WS server program in UTF-8.
Related concepts
Examples using com.HTTPServiceRequest methods on page 2870
These examples use methods of the com.HTTPServiceRequest class.
Examples: Using the com.HTTPPart class on page 2910
Examples using methods of the com.HTTPPart class.

com.HTTPServiceRequest.getURLHost
Returns the host name contained in the URL of an HTTP service request.

Syntax

```
getURLHost ()
RETURNS STRING
```

Usage
Returns the host name contained in the URL of an HTTP service request.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

Related concepts
Examples using com.HTTPServiceRequest methods on page 2870
These examples use methods of the com.HTTPServiceRequest class.
Examples: Using the com.HTTPPart class on page 2910
Examples using methods of the com.HTTPPart class.

com.HTTPServiceRequest.getURLPath
Returns the path contained in the URL of an HTTP service request.

Syntax

```
getURLPath ()
RETURNS STRING
```

Usage
Returns the path contained in the URL of an HTTP service request.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

Related concepts
Examples using com.HTTPServiceRequest methods on page 2870
These examples use methods of the com.HTTPServiceRequest class.
Examples: Using the com.HTTPPart class on page 2910
Examples using methods of the com.HTTPPart class.

com.HTTPServiceRequest.getURLPort
Returns the port number contained in the URL of an HTTP service request.

Syntax

```
getURLPort ()
```
**Usage**

Returns the port number contained in the URL of an HTTP service request.

The method returns NULL if there is no port.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**Related concepts**

- Examples using com.HTTPServiceRequest methods on page 2870
- These examples use methods of the com.HTTPServiceRequest class.
- Examples: Using the com.HTTPPart class on page 2910
- Examples using methods of the com.HTTPPart class.

### com.HTTPServiceRequest.getURLQuery

Takes a dynamic array of RECORD of two strings and fills that array with the decoded query string of an HTTP service request.

**Syntax**

```plaintext
getURLQuery(
    query RECORD)
```

1. `query` defines a record with the following structure:

   ```plaintext
   DEFINE query DYNAMIC ARRAY OF RECORD
   name STRING,
   value STRING
   END RECORD
   ```

   The WSHelper library provides the WSHelper.WSQueryType variable for your use. See WSHelper variables and records on page 3500.

**Usage**

Takes a dynamic array of RECORD of two strings and fills that array with the decoded query string of an HTTP service request. If there is no query string, the dynamic array size will be zero (0).

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

If the given array is not of the expected type, it raises exception -15535.

If there is an encoding issue, the STATUS is set to -15552. No exception is raised. You can process the query parts that do not have any UTF-8 conversion issues.

**Related concepts**

- Examples using com.HTTPServiceRequest methods on page 2870
- These examples use methods of the com.HTTPServiceRequest class.
- Examples: Using the com.HTTPPart class on page 2910
- Examples using methods of the com.HTTPPart class.

### com.HTTPServiceRequest.hasRequestKeepConnection

...
Returns TRUE if the connection remains open after sending a response.

**Syntax**

```java
hasRequestKeepConnection ()
RETURNS INTEGER
```

**Usage**

The `hasRequestKeepConnection()` method returns if the request expects the connection to stay open after sending the response.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**Related concepts**

- Examples using com.HTTPServiceRequest methods on page 2870
  These examples use methods of the com.HTTPServiceRequest class.

- Examples: Using the com.HTTPPart class on page 2910
  Examples using methods of the com.HTTPPart class.

---

**com.HTTPServiceRequest.readDataRequest**

Returns the body of a request in a `BYTE`.

**Syntax**

```java
readDataRequest (b BYTE)
```

1. `b` defines a variable of type `BYTE` that will be filled with the request body.

**Usage**

The `readDataRequest()` method returns the body of the request in a `BYTE`.

Supported methods are PUT and POST.

The `BYTE` variable must be located in memory, and will be filled with the request body. The existing content of the `BYTE` will be discarded.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

The `INT_FLAG` variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 3555.

**Related concepts**

- Examples using com.HTTPServiceRequest methods on page 2870
  These examples use methods of the com.HTTPServiceRequest class.

- Examples: Using the com.HTTPPart class on page 2910
  Examples using methods of the com.HTTPPart class.

---

**com.HTTPServiceRequest.readFileRequest**
Returns the body of a request in a file.

Syntax

```java
readFileRequest()
RETURNS STRING
```

Usage

The `readFileRequest()` method returns the body of the request to a file on disk.

The file is created in the temporary directory used by the runtime system (DBTEMP). The name of the file is the basename found in the HTTP Content-Disposition Header. If this basename is not specified, the file name is created with a UUID. If a file with the same name already exists in the temporary directory, the API prefixes the new file with a number. It is then of the form: `/tmp/ABC/filename_index.ext`, where `index` represents the number of files with the same name on disk.

Supported methods are PUT and POST.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 3555

Related concepts

Examples using com.HTTPServiceRequest methods on page 2870
These examples use methods of the com.HTTPServiceRequest class.

Examples: Using the com.HTTPPart class on page 2910
Examples using methods of the com.HTTPPart class.

com.HTTPServiceRequest.readFormEncodedRequest

Returns the string of a GET request with UTF-8 conversion option.

Syntax

```java
readFormEncodedRequest(
    utf8 INTEGER)
RETURNS STRING
```

1. `utf8` specifies if the string must be decoded to the current charset.

Usage

The `readFormEncodedRequest()` method returns the query of a POST "application/x-www-form-urlencoded" request or the query string of a GET request, decoded depending on HTML4 or XFORM if `utf8` is TRUE.

Note: If the result string contains & or = XForms special characters, these are doubled as follows:

na&me=va=ue

If the `utf8` parameter is TRUE, the decoded query string is translated from UTF-8 to the current character set. This may lead to a conversion error.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 3555
Related concepts
Examples using com.HTTPServiceRequest methods on page 2870
These examples use methods of the com.HTTPServiceRequest class.
Examples: Using the com.HTTPPart class on page 2910
Examples using methods of the com.HTTPPart class.

com.HTTPServiceRequest.readTextRequest
Returns the request body as a plain string.

Syntax

```java
readTextRequest ()
RETURNS STRING
```

Usage

The `readTextRequest()` method returns the body of the request as a string.

Supported methods are PUT and POST.

The request Content-Type header can be of any form */*. For example: `application/json`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

The `INT_FLAG` variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 3555.

Related concepts
Examples using com.HTTPServiceRequest methods on page 2870
These examples use methods of the com.HTTPServiceRequest class.
Examples: Using the com.HTTPPart class on page 2910
Examples using methods of the com.HTTPPart class.

com.HTTPServiceRequest.readXmlRequest
Returns the request body as an XML document.

Syntax

```java
readXmlRequest ()
RETURNS xml.DomDocument
```

Usage

The `readXmlRequest()` method returns the request as an entire XML document, contained in a `xml.DomDocument` object.

Supported methods are PUT and POST.

The request Content-Type header must be of the form */xml or */+xml. For example: `application/xhtml+xml`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

The `INT_FLAG` variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 3555.
Related concepts

Examples using com.HTTPServiceRequest methods on page 2870
These examples use methods of the com.HTTPServiceRequest class.

Examples: Using the com.HTTPPart class on page 2910
Examples using methods of the com.HTTPPart class.

com.HTTPServiceRequest.sendDataResponse
Sends an HTTP response with data of a BYTE variable.

Syntax

```java
sendDataResponse(
    code INTEGER,
    description STRING,
    b BYTE )
```

1. `code` specifies the status code of the response.
2. `description` specifies the description of the response.
3. `b` defines a BYTE type containing the data to be sent.

Usage

The `sendDataResponse()` method performs the HTTP response by sending the status (`code`) and description (`description`), followed by the headers previously set and binary data contained in the `b` program variable as body.

It is important for the server to return a correct status `code`, following the HTTP standards, otherwise the client may fail to interpret the response. For instance, if the request is malformed, the server is expected to send an HTTP response with the code of 400 (Bad Request). See HTTP status codes (wikipedia) for more details about common HTTP response codes.

The `b` parameter is defined as a BYTE and must be located in memory and not NULL, otherwise the operation fails.

The default Content-Type header is `application/octet-stream`, but it can be changed to any other mime type. For example: `image/jpeg`.

In HTTP 1.1, if the body size is greater than 32k, the response will be sent in several chunks of the same size.

If the description is NULL, a default description according to the status code is sent.

New incoming requests can be retrieved again with the `com.WebServiceEngine.GetHTTPServiceRequest()` method.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 3555.

Related concepts

Examples using com.HTTPServiceRequest methods on page 2870
These examples use methods of the com.HTTPServiceRequest class.

Examples: Using the com.HTTPPart class on page 2910
Examples using methods of the com.HTTPPart class.

com.HTTPServiceRequest.sendFileResponse
Sends an HTTP response with the data contained in a file.

**Syntax**

```java
sendFileResponse(
    code INTEGER,
    description STRING,
    filename STRING )
```

1. `code` specifies the status code of the response.
2. `description` specifies the description of the response.
3. `filename` specifies the path of the file containing the data to be send.

**Usage**

The `sendFileResponse()` method performs the HTTP response by sending the status (`code`) and description (`description`), followed by the headers previously set and the data contained in the specified file as body.

It is important for the server to return a correct status `code`, following the HTTP standards, otherwise the client may fail to interpret the response. For instance, if the request is malformed, the server is expected to send an HTTP response with the code of 400 (Bad Request). See HTTP status codes (wikipedia) for more details about common HTTP response codes.

If not defined by the programmer, the HTTP headers are automatically set as follows:

- **Content-Type** is defined based on the file name extension. If the file extension is not recognized, `Content-Type` defaults to `application/octet-stream`.
  
  **Note:** File extensions to `Content-Type` mapping can be customized in the file `$FGLDIR/lib/wse/mime.cfg`.

- **Content-Disposition** is set with the base name of the given `filename` as follows: `attachment; filename="basename"`.

For example, when calling the method as follows:

```java
CALL server.sendFileResponse( 200, NULL, "/opt/myapp/resources/logo.jpg" )
```

The resulting HTTP headers of the response will look like:

```text
Content-Type: image/jpeg
Content-Disposition: attachment; filename="logo.jpg"
```

In HTTP 1.1, if the body size is greater than 32k, the response will be sent in several chunks of the same size.

If the description is `NULL`, a default description according to the status code is sent.

New incoming requests can be retrieved again with the `com.WebServiceEngine.GetHTTPServiceRequest()` method.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

The `INT_FLAG` variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 3555.

**Related concepts**

- Examples using `com.HTTPServiceRequest` methods on page 2870
  
  These examples use methods of the `com.HTTPServiceRequest` class.

- Examples: Using the `com.HTTPPart` class on page 2910
Examples using methods of the `com.HTTPPart` class.

`com.HTTPServiceRequest.sendResponse`
Sends an HTTP response without body.

**Syntax**

```java
sendResponse(
    code INTEGER,
    description STRING )
```

1. `code` specifies the status code of the response.
2. `description` specifies the description of the response.

**Usage**

The `sendResponse()` method performs the HTTP response by sending a status (`code`) and description (`description`), followed by the headers previously set, without a body.

It is important for the server to return a correct status `code`, following the HTTP standards, otherwise the client may fail to interpret the response. For instance, if the request is malformed, the server is expected to send an HTTP response with the code of 400 (Bad Request). See [HTTP status codes](https://en.wikipedia.org/wiki/List_of_HTTP_status_codes) for more details about common HTTP response codes.

If the description is `NULL`, a default description based on the status code is sent.

New incoming requests can be retrieved again with the `com.WebServiceEngine.GetHTTPServiceRequest()` method.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See [Error handling in GWS calls (STATUS)](https://www.ibm.com/support/knowledgecenter/SSSHWH_14.1.0/com.ibm.db2.luw.ur.doc/doc/docx/erradm.htm) on page 3554.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see [ Interruption handling in GWS calls (INT_FLAG) ](https://www.ibm.com/support/knowledgecenter/SSSHWH_14.1.0/com.ibm.db2.luw.ur.doc/doc/docx/erradm.htm) on page 3555.

**Related concepts**

- Examples using `com.HTTPServiceRequest` methods on page 2870
- These examples use methods of the `com.HTTPServiceRequest` class.

- Examples: Using the `com.HTTPPart` class on page 2910
- Examples using methods of the `com.HTTPPart` class.

`com.HTTPServiceRequest.sendTextResponse`
Sends an HTTP response with data from a plain string.

**Syntax**

```java
sendTextResponse(
    code INTEGER,
    description STRING,
    txt STRING )
```

1. `code` specifies the status code of the response.
2. `description` specifies the description of the response.
3. `txt` defines the string containing the data to be sent.
Usage

The `sendTextResponse()` method performs the HTTP response by sending the a status (`code`) and description (`description`), followed by the headers previously set, and text data contained in the string as body.

It is important for the server to return a correct status `code`, following the HTTP standards, otherwise the client may fail to interpret the response. For instance, if the request is malformed, the server is expected to send an HTTP response with the code of 400 (Bad Request). See [HTTP status codes (wikipedia)](https://en.wikipedia.org/wiki/List_of_HTTP_status_codes) for more details about common HTTP response codes.

The default Content-Type header is `text/plain`, but it can be changed if of the form `*/*`. For example: `application/json`.

Automatic conversion from locale to user-defined charset is performed when possible, otherwise throws an exception.

In HTTP 1.1, if the body size is greater than 32k, the response will be sent in several chunks of the same size.

If the description is `NULL`, a default description based on the status code is sent.

New incoming requests can be retrieved again with the `com.WebServiceEngine.GetHTTPServiceRequest()` method.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See [Error handling in GWS calls](page 3554) on page 3554.

The `INT_FLAG` variable is checked during GWS API call to handle program interruptions, for more details, see [Interruption handling in GWS calls (INT_FLAG)](page 3555) on page 3555.

### Related concepts

- [Examples using com.HTTPServiceRequest methods](page 2870)
  
  These examples use methods of the `com.HTTPServiceRequest` class.

- [Examples: Using the com.HTTPPart class](page 2910)
  
  Examples using methods of the `com.HTTPPart` class.

### Syntax

```java
sendTextResponse(
    code INTEGER,
    desc STRING,
    doc xml.DomDocument)
```

1. `code` specifies the status code of the response.
2. `description` specifies the description of the response.
3. `doc` specifies the XML document containing the data to be sent.

Usage

The `sendTextResponse()` method performs the HTTP response by sending a status (`code`) and description (`description`), followed by the headers previously set, and the XML data contained in the passed `xml.DomDocument` object as body.

It is important for the server to return a correct status `code`, following the HTTP standards, otherwise the client may fail to interpret the response. For instance, if the request is malformed, the server is expected to send an HTTP response with the code of 400 (Bad Request). See [HTTP status codes (wikipedia)](https://en.wikipedia.org/wiki/List_of_HTTP_status_codes) for more details about common HTTP response codes.

The default Content-Type header is `text/xml`, but it can be changed if of the form `*/xml` or `*/+xml`. For example: `application/xhtml+xml`.
In HTTP 1.1, if the body size is greater than 32k, the response will be sent in several chunks of the same size.

If the description is NULL, a default description based on the status code is sent.

New incoming requests can be retrieved again with the `com.WebServiceEngine.GetHTTPServiceRequest()` method.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 3555.

**Related concepts**

- Examples using com.HTTPServiceRequest methods on page 2870
  - These examples use methods of the `com.HTTPServiceRequest` class.
- Examples: Using the `com.HTTPPart` class on page 2910
  - Examples using methods of the `com.HTTPPart` class.

**Syntax**

```java
com.HTTPServiceRequest.setResponseCharset
```

Defines the HTTP response character set.

**Usage**

The `setResponseCharset()` method defines the character set to use when sending an HTTP response.

The server must send a response in a character set that the client understands.

If the response character set is not defined by `setResponseCharset()`, the same character set as the client request is used, or the implicit ISO-8859-1 charset is used if the character is not defined by the client request.

The method must be called before sending the response with one of `sendResponse`, `sendTextResponse`, `sendXMLResponse`, or `beginXMLResponse` and `endXMLResponse` methods.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**Related concepts**

- Localization on page 512
  - Localization support allows you to implement programs that follow specific language and cultural rules.
- Examples using com.HTTPServiceRequest methods on page 2870
  - These examples use methods of the `com.HTTPServiceRequest` class.

**Syntax**

```java
com.HTTPServiceRequest.setResponseCookies
```

Allows the server to return cookies to be set on the client application sending the request.

**Usage**

The `setResponseCookies()` method allows the server to return cookies to be set on the client application sending the request.

The method must be called before sending the response with one of `sendResponse`, `sendTextResponse`, `sendXMLResponse`, or `beginXMLResponse` and `endXMLResponse` methods.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.
1. *cookies* defines a dynamic array for the cookies to be set. See WSHelper variables and records on page 3500 for more information regarding WSServerCookiesType.

**Usage**

Allows the server to return cookies to be set on the client application sending the request.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**Setting cookies example**

```plaintext
IMPORT FGL WSHelper

DEFINE cookies WSHelper.WSServerCookiesType

# Set first cookie named 'CookieName'
LET cookies[1].name = "CookieName"
LET cookies[1].value = "AnyValue"
LET cookies[1].expires = now + INTERVAL (5) MINUTE TO MINUTRE

# Set second cookie named 'SecondCookie'
LET cookies[2].name = "SecondCookie"
LET cookies[2].value = "AnotherValue"

# Set all cookies defined in the cookie array
CALL req.setResponseCookies(cookies)
CALL req.sendTextResponse(200, NULL, "Hello world")
```

**Related concepts**

Examples using com.HTTPServiceRequest methods on page 2870
These examples use methods of the com.HTTPServiceRequest class.

Examples: Using the com.HTTPPart class on page 2910
Examples using methods of the com.HTTPPart class.

**Syntax**

```plaintext
setResponseHeader (  
    name STRING,  
    value STRING )
```

1. *name* specifies the name of a header.
2. *value* defines the value of a header.

**Usage**

The setResponseVersion() method sets (or replaces) the name and value of an HTTP response header.

The Content-Length header cannot be set, because it is computed internally according to the body size.

The method must be called before sending the response with one of sendResponse, sendTextResponse, sendXmlResponse, or beginXmlResponse and endXmlResponse methods.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.
Related concepts

Examples using com.HTTPServiceRequest methods on page 2870
These examples use methods of the com.HTTPServiceRequest class.

Examples: Using the com.HTTPPart class on page 2910
Examples using methods of the com.HTTPPart class.

com.HTTPServiceRequest.setResponseMultipartType
Sets HTTP response in multipart mode of given type.

Syntax

```java
setResponseMultipartType(
    type STRING,
    start STRING,
    boundary STRING )
```

1. `type` defines one of the following:
   - form-data: Browser Xform with attachment
   - mixed: Parts are independent
   - related: Parts are dependent (Required for SOAP)
   - alternative: Parts are different types of the same document
   - *or any other type*
   - NULL: switch multipart mode off

2. `start` specifies the Content-ID value of root multipart document. Must be ASCII (optional)
3. `boundary` specifies the string used as multipart boundary. Must be ASCII (optional)

Usage

Sets HTTP response in multipart mode of given type. Calling one of the standard request methods will send the HTTP response as the given multipart type, even if no other part has been set.

The root HTTP part must be handled via the standard HTTPServiceRequest methods such as sendTextRequest(), sendXmlRequest(), sendDataRequest() and BeginXmlResponse().

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

Related concepts

Examples using com.HTTPServiceRequest methods on page 2870
These examples use methods of the com.HTTPServiceRequest class.

Examples: Using the com.HTTPPart class on page 2910
Examples using methods of the com.HTTPPart class.

com.HTTPServiceRequest.setResponseVersion
Defines the HTTP response version.

Syntax

```java
setResponseVersion(
    version STRING )
```

1. `version` specifies the HTTP response version.
Usage

The `setResponseVersion()` method defines the HTML response version (1.0 or 1.1). If not set, the same version as the request is used.

The method must be called before sending the response with one of `sendResponse`, `sendTextResponse`, `sendXmlResponse`, or `beginXmlResponse` and `endXmlResponse` methods.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the $\text{SQLCA.SQLERRM}$ register. See Error handling in GWS calls (STATUS) on page 3554.

Related concepts

Examples using com.HTTPServiceRequest methods on page 2870

These examples use methods of the com.HTTPServiceRequest class.

Examples: Using the com.HTTPPart class on page 2910

Examples using methods of the com.HTTPPart class.

Examples using com.HTTPServiceRequest methods

These examples use methods of the com.HTTPServiceRequest class.

Server-side example

```java
IMPORT COM
IMPORT Security
IMPORT FGL WSHelper

CONSTANT C_5_MINUTES = INTERVAL(5) MINUTE TO MINUTE

MAIN
  DEFINE expires DATETIME DAY TO SECOND
  DEFINE req com.HTTPServiceRequest
  DEFINE query WSHelper.WSQueryType
  DEFINE ind INTEGER
  DEFINE url, host, port, path, myCookie STRING
  DEFINE cookies WSHelper.WSServerCookiesType

  CALL com.WebServiceEngine.Start()
  WHILE true
    LET req = com.WebServiceEngine.GetHTTPServiceRequest(-1)
    LET url = req.getURL()
    DISPLAY "URL is ":url
    LET host = req.getURLHost()
    DISPLAY "Host is ":host
    LET port = req.getURLPort()
    DISPLAY "Port is ":port
    LET path = req.getURLPath()
    DISPLAY "Path is ":path
    CALL req.getUrlQuery(query)
    FOR ind = 1 TO query.getLength()
      DISPLAY "Query ":ind
      DISPLAY " key ":query[ind].name
      DISPLAY " val ":query[ind].value
    END FOR

    LET myCookie = req.findRequestCookie("UserIDCookie")
    IF myCookie IS NULL THEN
      DISPLAY "New user"
    ELSE
      DISPLAY "User id is ":myCookie
    END IF
```
# Compute Set-Cookies
DISPLAY "Now is ":, CURRENT
LET expires = CURRENT + C_5_MINUTES

LET cookies[1].name = "AnotherCookie"
LET cookies[1].value = "security.RandomGenerator.CreateUUIDString()"
LET cookies[1].domain = ".strasbourg.4js.com"
LET cookies[1].expires = expires
LET cookies[1].httpOnly = TRUE
CALL req.setResponseCookies(cookies)
# Send response with cookie
CALL req.sendTextResponse(200, NULL, "IT WORKS.")
END WHILE
END MAIN

HTTP classes
The HTTP classes manage HTTP client network operations on the client side.

The HTTPRequest class
The com.HTTPRequest class provides an interface to perform asynchronous XML and TEXT requests over HTTP for a specified URL, with additional XML streaming possibilities, on the client side.

The STATUS variable is set to zero after a successful method call.

Related reference
HTTP configuration on page 3495

com.HTTPRequest methods
Methods for the com.HTTPRequest class.

Table 616: Class methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.HTTPRequest.Create(</td>
<td>Creates a new HTTPRequest object from an URL.</td>
</tr>
<tr>
<td>url STRING )</td>
<td>RETURNS com.HTTPRequest</td>
</tr>
</tbody>
</table>
### Table 617: Object methods: Configuration methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clearAuthentication()</td>
<td>Removes user-defined authentication.</td>
</tr>
<tr>
<td>clearHeaders()</td>
<td>Removes all user-defined HTTP request headers.</td>
</tr>
<tr>
<td>removeHeader(name STRING)</td>
<td>Removes a named HTTP request header.</td>
</tr>
<tr>
<td>setAuthentication(login STRING, pass STRING, scheme STRING, realm STRING)</td>
<td>Defines the user login and password to authenticate to the server.</td>
</tr>
<tr>
<td>setAutoReply(val INTEGER)</td>
<td>Defines the auto reply option for response methods.</td>
</tr>
<tr>
<td>setCharset(charset STRING)</td>
<td>Defines the charset used when sending text or XML.</td>
</tr>
<tr>
<td>setConnectionTimeOut(timeout INTEGER)</td>
<td>Defines the timeout for the establishment of the connection.</td>
</tr>
<tr>
<td>setHeader(name STRING, value STRING)</td>
<td>Sets an HTTP header for the request.</td>
</tr>
<tr>
<td>setMethod(method STRING)</td>
<td>Sets the HTTP method of the request.</td>
</tr>
<tr>
<td>setKeepConnection(keep INTEGER)</td>
<td>Defines whether a connection is kept open if a new request occurs.</td>
</tr>
<tr>
<td>setMaximumResponseLength(length INTEGER)</td>
<td>Defines the maximum size in Kbytes of a response.</td>
</tr>
<tr>
<td>setTimeOut(timeout INTEGER)</td>
<td>Defines the timeout for a reading or writing operation.</td>
</tr>
<tr>
<td>setVersion(version STRING)</td>
<td>Sets the HTTP version of the request.</td>
</tr>
</tbody>
</table>
### Table 618: Object methods: Sending methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>beginXmlRequest()</code></td>
<td>Starts a streaming HTTP request.</td>
</tr>
<tr>
<td><code>endXmlRequest (stax xml.StaxWriter)</code></td>
<td>Terminates a streaming HTTP request.</td>
</tr>
<tr>
<td><code>doDataRequest (b BYTE)</code></td>
<td>Performs the request by sending binary data.</td>
</tr>
<tr>
<td><code>doFileRequest (filename STRING)</code></td>
<td>Performs the request by sending data contained in a file.</td>
</tr>
<tr>
<td><code>doFormEncodedRequest (query STRING, utf8 INTEGER)</code></td>
<td>Performs an &quot;application/x-www-form-urlencoded forms&quot; encoded query.</td>
</tr>
<tr>
<td><code>doRequest()</code></td>
<td>Performs the HTTP request.</td>
</tr>
<tr>
<td><code>doTextRequest (str STRING)</code></td>
<td>Performs the request by sending an entire string at once.</td>
</tr>
<tr>
<td><code>doXmlRequest (doc xml.DomDocument)</code></td>
<td>Performs the request by sending an entire XML document at once.</td>
</tr>
</tbody>
</table>

### Table 619: Object methods : Response methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getAsyncResponse()</code></td>
<td>When available, returns the response produced by one of request methods.</td>
</tr>
<tr>
<td><code>getResponse()</code></td>
<td>Waits for and returns the response produced by one of request methods.</td>
</tr>
</tbody>
</table>
### Table 620: Object methods of com.HTTPRequest: Multipart methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addPart ( part com.HTTPPart )</td>
<td>Adds a new part to the HTTP root part request.</td>
</tr>
<tr>
<td>setMultipartType ( type STRING, start STRING, boundary STRING )</td>
<td>Switch HTTPRequest in multipart mode of a given type.</td>
</tr>
</tbody>
</table>

### Table 621: Object methods of com.HTTPRequest: Cookie management

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setAutoCookies ( val INTEGER )</td>
<td>Enables automatic cookie management for a given request.</td>
</tr>
</tbody>
</table>

### Table 622: Object methods of com.HTTPRequest: Proxy management

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setProxy ( host STRING, port INTEGER )</td>
<td>Configure the proxy URL.</td>
</tr>
<tr>
<td>setProxyAuthentication ( login STRING, password STRING, scheme STRING, realm STRING )</td>
<td>Define the login and password to use for proxy authentication.</td>
</tr>
</tbody>
</table>

`com.HTTPRequest.addPart`  
Adds a new part to the HTTP root part request.

**Syntax**

```plaintext
addPart ( part com.HTTPPart )
```

1. `part` defines the HTTPPart object.

**Usage**

Adds a new part to the HTTP root part request. This part is sent after root part has been processed.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (`STATUS`) on page 3554.

**Related concepts**

Examples using com.HTTPRequest methods on page 2891  
These examples use methods of the `com.HTTPRequest` class.

Examples: Using the com.HTTPPart class on page 2910
Examples using methods of the `com.HTTPRequest` class.

`com.HTTPRequest.beginXmlRequest`
Starts a streaming HTTP request.

**Syntax**

```java
beginXmlRequest ()
RETURNS xml.StaxWriter
```

**Usage**

The `beginXmlRequest()` starts a streaming HTTP request and returns an `xml.StaxWriter` object ready to send to the server.

Supported methods are PUT and POST.

The default Content-Type header is `text/xml`, but it can be changed if of the form `*/xml` or `*/+xml`. For example: `application/xhtml+xml`.

In HTTP 1.1, if the body size is greater than 32 KB, the request will be sent in several chunks of the same size.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

The `INT_FLAG` variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 3555.

**Related concepts**

- Examples using `com.HTTPRequest` methods on page 2891
- These examples use methods of the `com.HTTPRequest` class.

`com.HTTPRequest.clearAuthentication`
Removes user-defined authentication.

**Syntax**

```java
clearAuthentication()
```

**Usage**

Removes user-defined authentication.

If an `authenticate` entry exists in the FGLPROFILE file, it will be used for authentication, even if the user-defined authentication was removed.

**Important:** The iOS HTTP stack doesn't provide a simple way to handle authentication. The GMI front-end uses the global iOS credential management system, that keeps credential values of previous request based on host and realm, until the keep-alive session is closed. Therefore, doing a `clearAuthentication()` on iOS devices does not take effect immediately.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**Related concepts**

- Examples using `com.HTTPRequest` methods on page 2891
These examples use methods of the `com.HTTPRequest` class.

**Examples: Using the `com.HTTPPart` class** on page 2910
Examples using methods of the `com.HTTPPart` class.

**com.HTTPRequest.clearHeaders**
Removes all user-defined HTTP request headers.

**Syntax**

```java
clearHeaders()
```

**Usage**
Removes all user-defined HTTP request headers defined with the `setHeader()` method.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (`STATUS`) on page 3554.

**Related concepts**

- Examples using `com.HTTPRequest` methods on page 2891
- These examples use methods of the `com.HTTPRequest` class.

**Examples: Using the `com.HTTPPart` class** on page 2910
Examples using methods of the `com.HTTPPart` class.

**com.HTTPRequest.Create**
Creates a new `HTTPRequest` object from an URL.

**Syntax**

```java
com.HTTPRequest.Create(
    url STRING )
RETURNS com.HTTPRequest
```

1. `url` defines the URL for the HTTP request.

**Usage**
Creates a `com.HTTPRequest` object by providing a mandatory URL with HTTP or HTTPS as protocol.

The `url` parameter can be an identifier of an URL mapping with an optional `alias://` prefix. See FGLPROFILE Configuration for more details about URL mapping with aliases, and for proxy and security configuration.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (`STATUS`) on page 3554.

**Related concepts**

- Examples using `com.HTTPRequest` methods on page 2891
- These examples use methods of the `com.HTTPRequest` class.

**Examples: Using the `com.HTTPPart` class** on page 2910
Examples using methods of the `com.HTTPPart` class.

**com.HTTPRequest.doDataRequest**
Performs the request by sending binary data.

**Syntax**

```
doDataRequest (  
   b BYTE  )
```

1. `b` defines the binary data.

**Usage**

Performs the request by sending binary data contained in the `b` variable.

Supported methods are PUT and POST.

The `b` must be located in memory and not NULL otherwise the operation fails.

The default Content-Type header is `application/octet-stream`, but it can be changed to any other mime type. For example: `image/jpeg`.

In HTTP 1.1, if the body size is greater than 32k, the request will be sent in several chunks of the same size.

This HTTP request method is non-blocking. It returns immediately after the call. Use the `com.HTTPRequest.getResponse` on page 2882 method, to perform a synchronous HTTP request, suspending the program flow until the response returns from the server. If the program must keep going, use the `com.HTTPRequest.getAsyncResponse` on page 2881 method, to check if a response is available.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

The `INT_FLAG` variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 3555

**Related concepts**

- Examples using `com.HTTPRequest` methods on page 2891
  These examples use methods of the `com.HTTPRequest` class.
- Examples: Using the `com.HTTPPart` class on page 2910
  Examples using methods of the `com.HTTPPart` class.

`com.HTTPRequest.doFileRequest`

Performs the request by sending data contained in a file.

**Syntax**

```
doFileRequest (  
   filename STRING  )
```

1. `filename` defines the file containing the data to be send.

**Usage**

Performs the request by sending data contained in the file passed as parameter. The data is sent as is without any further conversion.

Supported methods are PUT and POST.

If not defined by programmer, the HTTP headers are automatically set as follows:

- `Content-Type` is defined based on the file name extension. If the file extension is not recognized, `Content-Type` defaults to `application/octet-stream`. 
Note: File extensions to Content-Type mapping can be customized in the file $FGGLDIR/lib/wse/mime.cfg.

- Content-Disposition is set with the base name of the given filename as follows: attachment; filename="basename".

For example, when calling the method as follows:

```java
CALL request.doFileRequest( "/opt/myapp/resources/logo.jpg" )
```

The resulting HTTP headers of the POST or PUT will look like:

```
Content-Type: image/jpeg
Content-Disposition: attachment; filename="logo.jpg"
```

In HTTP 1.1, if the body size is greater than 32k, the request will be sent in several chunks of the same size.

This HTTP request method is non-blocking. It returns immediately after the call. Use the `com.HTTPRequest.getResponse` on page 2882 method, to perform a synchronous HTTP request, suspending the program flow until the response returns from the server. If the program must keep going, use the `com.HTTPRequest.getAsyncResponse` on page 2881 method, to check if a response is available.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 3555

Related concepts

Examples using com.HTTPRequest methods on page 2891
These examples use methods of the com.HTTPRequest class.

Examples: Using the com.HTTPPart class on page 2910
Examples using methods of the com.HTTPPart class.

`com.HTTPRequest.doFormEncodedRequest`
Performs an "application/x-www-form-urlencoded forms" encoded query.

**Syntax**

```java
doFormEncodedRequest ( 
    query STRING,
    utf8 INTEGER )
```

1. `query` defines a list of name/value pairs separated by an &.
2. `utf8` defines if the query string is UTF-8 encoded.

**Usage**

The `doFormEncodedRequest()` method performs a request with an "application/x-www-form-urlencoded forms" encoded query.

Supported methods are GET and POST.

The `query` string is a list of name/value pairs separated by an ampersand (&). For example:

```java
name1=value1&name2=value2&name3=value3
```

**Note:** If you need to URL-encode the separator characters & and =, double them as following: na&&me=va==lue.
If the `utf8` parameter is `TRUE`, the query string is encoded in UTF-8 as specified in `XForms1.0`, otherwise in ASCII as specified in `HTML4`.

This HTTP request method is non-blocking. It returns immediately after the call. Use the `com.HTTPRequest.getResponse` on page 2882 method, to perform a synchronous HTTP request, suspending the program flow until the response returns from the server. If the program must keep going, use the `com.HTTPRequest.getAsyncResponse` on page 2881 method, to check if a response is available.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See `Error handling in GWS calls (STATUS)` on page 3554.

The `INT_FLAG` variable is checked during GWS API call to handle program interruptions, for more details, see `Interruption handling in GWS calls (INT_FLAG)` on page 3555

**Related concepts**

Examples using `com.HTTPRequest` methods on page 2891
These examples use methods of the `com.HTTPRequest` class.

Examples: Using the `com.HTTPPart` class on page 2910
Examples using methods of the `com.HTTPPart` class.

**com.HTTPRequest.doRequest**
Performs the HTTP request.

### Syntax

```java
doRequest()
```

### Usage

The `doRequest()` method performs the HTTP request.

Supported methods are GET, HEAD, POST, PUT and DELETE.

**Note:** For POST and PUT requests, the body will be empty.

This HTTP request method is non-blocking. It returns immediately after the call. Use the `com.HTTPRequest.getResponse` on page 2882 method, to perform a synchronous HTTP request, suspending the program flow until the response returns from the server. If the program must keep going, use the `com.HTTPRequest.getAsyncResponse` on page 2881 method, to check if a response is available.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See `Error handling in GWS calls (STATUS)` on page 3554.

The `INT_FLAG` variable is checked during GWS API call to handle program interruptions, for more details, see `Interruption handling in GWS calls (INT_FLAG)` on page 3555

**Related concepts**

Examples using `com.HTTPRequest` methods on page 2891
These examples use methods of the `com.HTTPRequest` class.

Examples: Using the `com.HTTPPart` class on page 2910
Examples using methods of the `com.HTTPPart` class.

**com.HTTPRequest.doTextRequest**
Performs the request by sending an entire string at once.

### Syntax

```java
doTextRequest()
```
1. *str* defines a string containing the request.

**Usage**

Performs the request by sending an entire string at once.

Supported methods are PUT and POST.

The default Content-Type header is `text/plain`, but it can be changed if of the form `*/*`. For example: `application/json`.

Automatic character set conversion from the application locale to the user-defined charset is performed. In case of conversion error, the method throws an exception.

**Note:** To avoid character conversion problems when sending text over HTTP, consider setting the same user-defined character set as the program defined by the application locale (assuming that the server understands the client application character set).

In HTTP 1.1, if the body size is greater than 32 KB, the request will be sent in several chunks of the same size.

This HTTP request method is non-blocking. It returns immediately after the call. Use the `com.HTTPRequest.getResponse` method, to perform a synchronous HTTP request, suspending the program flow until the response returns from the server. If the program must keep going, use the `com.HTTPRequest.getAsyncResponse` method, to check if a response is available.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

The `INT_FLAG` variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 3555.

**Related concepts**

Examples using com.HTTPRequest methods on page 2891

These examples use methods of the `com.HTTPRequest` class.

Examples: Using the com.HTTPPart class on page 2910

Examples using methods of the `com.HTTPPart` class.

**Syntax**

```java
doXmlRequest (doc xml.DomDocument )
```

1. *doc* defines the XML document containing the data to be sent.

**Usage**

The `doXmlRequest()` method performs the request by sending the entire passed `xml.DomDocument` at once.

Supported methods are PUT and POST.

The default Content-Type header is `text/xml`, but it can be changed if of the form `*/xml` or `*//*.xml`. For example: `application/xhtml+xml`.

In HTTP 1.1, if the body size is greater than 32 KB, the request will be sent in several chunks of the same size.
The character set used to send the XML data is defined by the encoding attribute in the XML document prolog. It is recommended that you define the HTTP request character set to NULL with the `setCharSet()` method, or that you use the same character set that was set in the XML Document.

This HTTP request method is non-blocking. It returns immediately after the call. Use the `com.HTTPRequest.getResponse` on page 2882 method, to perform a synchronous HTTP request, suspending the program flow until the response returns from the server. If the program must keep going, use the `com.HTTPRequest.getAsyncResponse` on page 2881 method, to check if a response is available.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 3555

**Related concepts**

Examples using com.HTTPRequest methods on page 2891
These examples use methods of the `com.HTTPRequest` class.

Examples: Using the `com.HTTPPart` class on page 2910
Examples using methods of the `com.HTTPPart` class.

```java
com.HTTPRequest.endXmlRequest
Terminates a streaming HTTP request.

Syntax

```
endXmlRequest(
    stax xml.StaxWriter
)
```

1. `stax` defines the `xml.StaxWriter` object used to write the HTTP request.

**Usage**

The `endXmlRequest()` method terminates a streaming HTTP request by closing the `xml.StaxWriter` object that was created with the `beginXmlRequest()` method.

This HTTP request method is non-blocking. It returns immediately after the call. Use the `com.HTTPRequest.getResponse` on page 2882 method, to perform a synchronous HTTP request, suspending the program flow until the response returns from the server. If the program must keep going, use the `com.HTTPRequest.getAsyncResponse` on page 2881 method, to check if a response is available.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 3555

**Related concepts**

Examples using com.HTTPRequest methods on page 2891
These examples use methods of the `com.HTTPRequest` class.

Examples: Using the `com.HTTPPart` class on page 2910
Examples using methods of the `com.HTTPPart` class.
When available, returns the response produced by one of request methods.

Syntax

```
getAsyncResponse()
RETURNS com.HTTPResponse
```

**Usage**

If a response is available, the `getAsyncResponse()` method returns a `com.HTTPResponse` object corresponding to the response that was produced by a call to one of the request methods: `doRequest()`, `doTextRequest()`, `doXmlRequest()`, `doFormEncodedRequest()`, or `beginXmlRequest()` and `endXmlRequest()`.

Unlike `getResponse()`, the `getAsyncResponse()` method is non-blocking: it returns immediately and does not stop the program flow when waiting for a response.

The method returns `null` if the HTTP response was not yet received from the server.

This method is typically called just after a `do*Request()` call, and if the returned value is `null`, it is called again after a short period of time, to check for a response. Within a dialog, use an `ON IDLE` block to issue a `getAsyncRequest()` every seconds for example.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (`STATUS`) on page 3554.

The `INT_FLAG` variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (`INT_FLAG`) on page 3555.

**Related concepts**

- `com.HTTPRequest.getResponse` on page 2882
  Waits for and returns the response produced by one of request methods.
  
  **Examples using `com.HTTPRequest` methods** on page 2891
  These examples use methods of the `com.HTTPRequest` class.

**Syntax**

```
getResponse()
RETURNS com.HTTPResponse
```

**Usage**

The `getResponse()` method waits for a response from the server and returns a `com.HTTPResponse` object corresponding to the response that was produced by a call to one of the request methods: `doRequest()`, `doTextRequest()`, `doXmlRequest()`, `doFormEncodedRequest()`, or `beginXmlRequest()` and `endXmlRequest()`.

**Note:** On iOS, a long running HTTP request will display a message box, to allow the user to cancel the request. If the user cancels the HTTP request, the error `-15578` will be raised. This error can be trapped with `TRY/CATCH`.

Unlike `getAsyncResponse()`, the `getResponse()` method is blocking: it stops program flow until an HTTP response is received from the server.

Define a response timeout with the `com.HTTPRequest.setTimeOut` on page 2890 method.
**Note:** On iOS devices, when using this method, it is not possible to distinguish different timeouts for the connection and for read/write operation, defined respectively by the `setConnectionTimeOut()` and `setTimeOut()` methods. If both timeouts are defined, the longest timeout will be used for the connection and read/write operations.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See *Error handling in GWS calls (STATUS)* on page 3554.

The `INT_FLAG` variable is checked during GWS API call to handle program interruptions, for more details, see * Interruption handling in GWS calls (INT_FLAG)* on page 3555.

**Related concepts**

- [com.HTTPRequest.getAsyncResponse](#) on page 2881
  - When available, returns the response produced by one of request methods.

- [Examples using com.HTTPRequest methods](#) on page 2891
  - These examples use methods of the `com.HTTPRequest` class.

**Syntax**

```java
removeHeader(
    name STRING
)
```

1. `name` defines the HTTP header name.

**Usage**

The `removeHeader()` method deletes an HTTP header identified by `name`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See *Error handling in GWS calls (STATUS)* on page 3554.

**Related concepts**

- [Examples using com.HTTPRequest methods](#) on page 2891
  - These examples use methods of the `com.HTTPRequest` class.

**Examples: Using the com.HTTPPart class** on page 2910
- Examples using methods of the `com.HTTPPart` class.

**Syntax**

```java
setAuthentication(
    login STRING,
    pass STRING,
    scheme STRING,
    realm STRING
)
```

1. `login` defines the login to use for authentication.
2. `pass` specifies the password to use for authentication.
3. `scheme` defines the method to be used during authentication. This is optional; it can be set to NULL.
4. `realm` defines the realm. This is optional; it can be set to NULL.
**Usage**

The `setAuthentication()` method defines the mandatory user login and password to authenticate to the server.

**Note:** If the server uses the Windows® NT (New Technology) LAN Manager (NTLM) protocol, authentication requires the request to keep the connection open via a call to `setKeepConnection(TRUE)`, otherwise authentication will fail.

The `scheme` parameter defines the method to be used during authentication. The supported values for the `scheme` parameter are `Anonymous`, `Basic`, `Digest`, and `NTLM`. The default is `Anonymous`.

An optional `realm` can be specified.

With `Anonymous`, `Digest`, or `NTLM` authentication, you must re-send the request if you get a 401 or 407 HTTP return code (authorization required). In the case of NTLM, the request must be re-sent one more time.

If a user-defined authentication is set and there is an `authenticate` entry for this URL in the FGLPROFILE file, the user-defined authentication has priority.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**Related concepts**

Examples using `com.HTTPRequest` methods on page 2891

These examples use methods of the `com.HTTPRequest` class.

Examples: Using the `com.HTTPPart` class on page 2910

Examples using methods of the `com.HTTPPart` class.

`com.HTTPRequest.setAutoCookies`

Enables automatic cookie management for a given request.

**Syntax**

```java
setAutoCookies(val INTEGER)
```

1. `val` defines the cookie management flag.

**Usage**

Set to `TRUE` to activate automatic cookie management.

If set to `TRUE`, it activates the automatic cookies management for that request. For the `com.HTTPRequest` instance:

- If the server returns session cookies (with no expiration date), they will be automatically sent again for the next request.
- If the server returns persistent cookies (with an expiration date), those cookies will be registered globally for the current `fglrn` process, and any other `HTTPRequest` (including the current one) will automatically send those cookies according to the request path and domain, as long as the expiration date has not expired.

The `autocookiesmanagement` option of the `com.WebServiceEngine.SetOption` on page 2837 method activates the automatic cookies management for any `HTTPRequest`.

The `maximumpersistentcookies` option of the `com.WebServiceEngine.SetOption` on page 2837 method sets the maximum number of cookies that can be handled by an `fglrn` process.

Default value is `FALSE`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.
com.HTTPRequest.setAutoReply

Defines the auto reply option for response methods.

Syntax

```plaintext
setAutoReply( 
  val INTEGER )
```

1. `val` defines auto-reply when `TRUE`.

Usage

The `setAutoReply()` method defines whether `getResponse()` or `getAsyncResponse()` will automatically perform another HTTP GET request if the response contains HTTP Authentication, Proxy Authentication or HTTP redirect data.

Available for GET method and the HTTP HEAD method.

The default is `TRUE`.

**Important:** On iOS devices, `setAutoReply()` is ignored for redirection in synchronous requests. The iOS HTTP stack does not allow you to set an auto reply option when doing synchronous requests.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

Related concepts

- `com.HTTPRequest.getResponse` on page 2882
  Waits for and returns the response produced by one of request methods.
- `com.HTTPRequest.getAsyncResponse` on page 2881
  When available, returns the response produced by one of request methods.
- Examples using `com.HTTPRequest methods` on page 2891
  These examples use methods of the `com.HTTPRequest` class.

com.HTTPRequest.setCharset

Defines the charset used when sending text or XML.

Syntax

```plaintext
setCharset( 
  charset STRING )
```

1. `charset` defines the character set to use.

Usage

Defines the character set used when sending an HTTP request.

By default, no character set information will be transmitted in the HTTP header. This is also the case when specifying `NULL` as parameter for this method.

If no character set is specified in HTTP headers, ISO8859-1 will implicitly be used as defined by the HTTP standards.
In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

Related concepts

Localization on page 512
Localization support allows you to implement programs that follow specific language and cultural rules.

Examples using com.HTTPRequest methods on page 2891
These examples use methods of the com.HTTPRequest class.

com.HTTPRequest.setConnectionTimeOut
Defines the timeout for the establishment of the connection.

Syntax

```
setConnectionTimeOut(
    timeout INTEGER
)
```

1. `timeout` defines the number of seconds.

Usage

The `setConnectionTimeOut()` method sets the time value in seconds to wait for the establishment of the connection, before a break.

The value of -1 means infinite wait.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

Related concepts

Examples using com.HTTPRequest methods on page 2891
These examples use methods of the com.HTTPRequest class.

Examples: Using the com.HTTPPart class on page 2910
Examples using methods of the com.HTTPPart class.

com.HTTPRequest.setHeader
Sets an HTTP header for the request.

Syntax

```
setHeader(
    name STRING,
    value STRING
)
```

1. `name` defines the HTTP header name.
2. `value` defines the HTTP header value.

Usage

The `setHeader()` method defines an HTTP header with a `name` and `value` for the request.

If a header exists with the same name, it is replaced with the new value.

Setting a header after the body has been sent, or if a streaming operation has been started, will not take effect. It will only be set when a new request is reissued.
In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**Related concepts**
- Examples using `com.HTTPRequest` methods on page 2891
  These examples use methods of the `com.HTTPRequest` class.
- Examples: Using the `com.HTTPPart` class on page 2910
  Examples using methods of the `com.HTTPPart` class.

**com.HTTPRequest.setKeepConnection**
Defines whether a connection is kept open if a new request occurs.

**Syntax**

```java
setKeepConnection(
  keep INTEGER )
```

1. `keep` defines if the connection is kept.

**Usage**
The `setKeepConnection()` method defines whether the connection stays open when a new HTTP request occurs.

The default is `FALSE`.

**Note:** If the server uses the Windows® NT (New Technology) LAN Manager (NTLM) protocol, authentication requires the request to keep the connection open via a call to `setKeepConnection(TRUE)`, otherwise authentication will fail.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**Related concepts**
- Examples using `com.HTTPRequest` methods on page 2891
  These examples use methods of the `com.HTTPRequest` class.
- Examples: Using the `com.HTTPPart` class on page 2910
  Examples using methods of the `com.HTTPPart` class.

**com.HTTPRequest.setMaximumResponseLength**
Defines the maximum size in Kbytes of a response.

**Syntax**

```java
setMaximumResponseLength(
  length INTEGER )
```

1. `length` defines the maximum size in Kbytes.

**Usage**
The `setMaximumResponseLength()` method sets the maximum authorized size in Kbytes of the whole response (including headers, body and all control characters), before a break.

The value of -1 means no limit.
Note: Setting the maximum response length is ignored for synchronous requests in a Genero Mobile for iOS (GMI) app. The iOS HTTP stack does not allow you to set a maximum response length when doing synchronous requests.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

Related concepts
Examples using com.HTTPRequest methods on page 2891
These examples use methods of the com.HTTPRequest class.
Examples: Using the com.HTTPPart class on page 2910
Examples using methods of the com.HTTPPart class.

com.HTTPRequest.setMethod
Sets the HTTP method of the request.

Syntax

```
setMethod(
    method STRING
)
```

1. `method` defines the HTTP method of the request.

Usage

The `setMethod()` method defines the HTTP method of the request.

Supported methods are GET, PUT, POST, HEAD, and DELETE.

The default is GET.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

Related concepts
Examples using com.HTTPRequest methods on page 2891
These examples use methods of the com.HTTPRequest class.
Examples: Using the com.HTTPPart class on page 2910
Examples using methods of the com.HTTPPart class.

com.HTTPRequest.setMultipartType
Switch HTTPRequest in multipart mode of a given type.

Syntax

```
setMultipartType(
    type STRING,
    start STRING,
    boundary STRING
)
```

1. `type` defines one of the following:
   - form-data: Browser Xform with attachment
   - mixed: Parts are independent
   - related: Parts are dependent (Required for SOAP)
   - alternative: Parts are different types of the same document
   - or any other type
   - NULL: switch multipart mode off
2. *start* defines the Content-ID value of root multipart document. (optional)
3. *boundary* defines a string used as multipart boundary. (optional)

**Usage**

Switch HTTPRequest in multipart mode of a given type. Calling one of the standard request methods will send the HTTP request as the given multipart type, even if no other part has been set.

**Important:** Multipart HTTP requests are not supported on GMI mobile devices.

The root HTTP part is the part handled via the standard HTTPRequest methods such as `doTextRequest()`, `doXmlRequest()`, `doDataRequest()` and `beginXmlRequest()`.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**Related concepts**

Examples using com.HTTPRequest methods on page 2891
These examples use methods of the com.HTTPRequest class.

Examples: Using the com.HTTPPart class on page 2910
Examples using methods of the com.HTTPPart class.

**Syntax**

```java
setProxy(
    host STRING,
    port INTEGER )
```

1. *host* defines the host of the proxy.
2. *port* defines the port number of the proxy.

**Usage**

Defines the proxy URL to be used for the current HTTPRequest request. Even if a proxy URL is configured in FGLPROFILE, the proxy set by the `setProxy()` method will be used.

If `proxy_host` is NULL or `proxy_port` is < 0, error -15535 (Invalid parameter) is raised.

If `proxy_host` is unreachable, error -15579 (COM_PROXY_ERROR) is raised.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**Related concepts**

Examples using com.HTTPRequest methods on page 2891
These examples use methods of the com.HTTPRequest class.

Examples: Using the com.HTTPPart class on page 2910
Examples using methods of the com.HTTPPart class.

com.HTTPRequest.setProxyAuthentication
Define the login and password to use for proxy authentication.

Syntax

```
setProxyAuthentication (  
  login STRING,  
  password STRING,  
  scheme STRING,  
  realm STRING )
```

1. `login` defines the login to use for authentication.
2. `password` specifies the password to use for authentication.
3. `scheme` defines the method to be used during authentication. This is optional; it can be set to NULL.
4. `realm` defines the realm. This is optional; it can be set to NULL.

Usage

Defines the login and password to use for proxy authentication for the current `HTTPRequest` request.

The `scheme` parameter defines the method to be used during authentication. The supported values for the `scheme` parameter are `Anonymous`, `Basic`, `Digest`, and `NTLM`. The default is `Anonymous`.

If `setProxy()` is called, the login and password will be used to authenticate against the proxy set by this API, regardless of whether a proxy is configured in FGLPROFILE.

If `setProxy()` has not been called and a proxy is configured in FGLPROFILE, the login and password set by this method will be used to authenticate against the proxy defined by FGLPROFILE.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

Related concepts

Examples using `com.HTTPRequest` methods on page 2891
These examples use methods of the `com.HTTPRequest` class.

Examples: Using the `com.HTTPPart` class on page 2910
Examples using methods of the `com.HTTPPart` class.

c.com.HTTPRequest.setTimeOut
Defines the timeout for a reading or writing operation.

Syntax

```
setTimeOut (  
  timeout INTEGER )
```

1. `timeout` defines the number of seconds.

Usage

The `setTimeOut()` method defines a delay in seconds, to wait for an HTTP request read or write operation. If the operation is not terminated after the timeout, it returns immediately with an error.

Use the value of -1 to define an infinite timeout.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.
Related concepts
Examples using com.HTTPRequest methods on page 2891
These examples use methods of the com.HTTPRequest class.
Examples: Using the com.HTTPPart class on page 2910
Examples using methods of the com.HTTPPart class.

com.HTTPRequest.setVersion
Sets the HTTP version of the request.

Syntax

```plaintext
define
      setVersion ( version STRING )
```

1. `version` defines the HTTP version of the request.

Usage
The `setVersion()` method defines the HTTP version of the request.

Accepted versions are 1.0 and 1.1 (only these two versions are supported).

The default is 1.1.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

Related concepts
Examples using com.HTTPRequest methods on page 2891
These examples use methods of the com.HTTPRequest class.
Examples: Using the com.HTTPPart class on page 2910
Examples using methods of the com.HTTPPart class.

Examples using com.HTTPRequest methods
These examples use methods of the com.HTTPRequest class.

These examples are provided:

- HTTP GET request on page 2891
- XForms HTTP POST request on page 2892
- Streaming HTTP PUT request on page 2892
- Asynchronous HTTP DELETE request on page 2893
- HTTP request cookie management on page 2894

HTTP GET request

```plaintext
import com
main
define req com.HTTPRequest
define resp com.HTTPResponse
try
   let req = req.com.HTTPRequest.Create("http://localhost:8090/MyPage")
   # Set additional HTTP header with name 'MyHeader', and value 'High Priority'
   call req.setHeader("MyHeader","High Priority")
   call req.doRequest()
   let resp = req.getResponse()
   if resp.getStatusCode() != 200 then
```
DISPLAY "HTTP Error ("||resp.getStatusCode()||") ",
    resp.getStatusDescription()
ELSE
    DISPLAY "HTTP Response is : ",resp.getTextResponse()
END IF
CATCH
    DISPLAY "ERROR ":"||STATUS||" ("||SQLCA.SQLERRM||")"
END TRY
END MAIN

XForms HTTP POST request

IMPORT com
IMPORT xml

MAIN
    DEFINE req com.HTTPRequest
    DEFINE resp com.HTTPResponse
    DEFINE doc xml.DomDocument
    TRY
        LET req = com.HTTPRequest.Create("http://localhost:8090/MyProcess")
        CALL req.setMethod("POST") # Perform an HTTP POST method
        # Param1 value is 'hello', Param2 value is 'how are you ?'
        CALL req.doFormEncodedRequest("Param1=hello&Param2=how are you ?",FALSE)
        LET resp = req.getResponse()
        IF resp.getStatusCode() != 200 THEN
            DISPLAY  "HTTP Error ("||resp.getStatusCode()||") ",
                resp.getStatusDescription()
        ELSE
            # Expect a returned content type of the form */xml
            LET doc = resp.getXmlResponse()
            DISPLAY   "HTTP XML Response is : ",doc.saveToString()
        END IF
    CATCH
        DISPLAY "ERROR ":"||STATUS||" ("||SQLCA.SQLERRM||")"
    END TRY
END MAIN

Streaming HTTP PUT request

IMPORT com
IMPORT xml

MAIN
    DEFINE req com.HTTPRequest
    DEFINE resp com.HTTPResponse
    DEFINE writer xml.StaxWriter
    TRY
        LET req = com.HTTPRequest.Create("http://localhost:8090/MyXmlProcess")
        CALL req.setMethod("PUT") # Perform an HTTP PUT method
        CALL req.setHeader("MyHeader","Value of my header")
        # Retrieve an xml.StaxWriter to start xml streaming
        LET writer = req.beginXmlRequest()
        CALL writer.startDocument("utf-8","1.0",true)
        CALL writer.comment("My first XML document sent in streaming with genero")
        CALL writer.startElement("root")
        CALL writer.attribute("attr1","value1")
        CALL writer.endElement()
        CALL writer.endDocument()
        CALL req.endXmlRequest(writer) # End streaming request
        LET resp = req.getResponse()
IF resp.getStatusCode() != 201 OR resp.getStatusCode() != 204 THEN
  DISPLAY "HTTP Error ("||resp.getStatusCode()||") ",
  resp.getStatusDescription()
ELSE
  DISPLAY "XML document was correctly put on the server"
END IF
CATCH
  DISPLAY "ERROR ":",STATUS||" ("||SQLCA.SQLERRM||")"
END TRY
END MAIN

Asynchronous HTTP DELETE request

IMPORT com

MAIN
  DEFINE req com.HTTPRequest
  DEFINE resp com.HTTPResponse
  DEFINE url STRING
  DEFINE quit CHAR(1)
  DEFINE questionStr STRING
  DEFINE timeout INTEGER
  TRY
    WHILE TRUE
      PROMPT "Enter http url you want to delete ? "
      FOR url ATTRIBUTES (CANCEL=FALSE)
        LET req = com.HTTPRequest.Create(url)
        CALL req.setMethod("DELETE")
        CALL req.doRequest()
        # Retrieve asynchronous response for the first time
        LET resp = req.getAsyncResponse()
        CALL Update(resp) RETURNING questionStr,timeout
        WHILE quit IS NULL OR ( quit != "Y" AND quit != "N" )
          PROMPT questionStr FOR CHAR quit
          ATTRIBUTES (CANCEL=FALSE,ACCEPT=FALSE,SHIFT="up")
          ON IDLE timeout
            IF resp IS NULL THEN # If no response at first try,
              # retrieve it again
              LET resp = req.getAsyncResponse() # as we now have time
              CALL Update(resp) RETURNING questionStr,timeout
            END IF
          END IF
        END WHILE
        IF quit == "Y" THEN
          EXIT PROGRAM
        ELSE
          LET quit = NULL
        END IF
      END WHILE
    CATCH
      DISPLAY "ERROR ":",STATUS,SQLCA.SQLERRM
    END TRY
  END MAIN

FUNCTION Update(resp)
  DEFINE resp com.HTTPResponse
  DEFINE ret STRING
  IF resp IS NOT NULL THEN
    IF resp.getStatusCode() != 204 THEN
      LET ret = "HTTP Error ("||resp.getStatusCode()||") :") ",
        resp.getStatusDescription()||". Do you want to quit ? "
    ELSE
      LET ret = "HTTP Page deleted. Do you want to quit ? "
    END IF
  END IF
  RETURN ret
END FUNCTION
HTTP request cookie management

The com.HTTPResponse class provides an interface to perform XML and TEXT responses over HTTP, with additional XML streaming possibilities, on the client side.

The STATUS variable is set to zero after a successful method call.
*com.HTTPResponse methods*
Methods for the `com.HTTPResponse` class.

**Table 623: Object methods**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>beginXmlResponse()</code></td>
<td>Starts a streaming HTTP response.</td>
</tr>
<tr>
<td><code>endXmlResponse(stax xml.StaxReader)</code></td>
<td>Performs the HTTP request.</td>
</tr>
<tr>
<td><code>getDataResponse(b BYTE)</code></td>
<td>Returns the entire HTTP response in a BYTE.</td>
</tr>
<tr>
<td><code>getFileResponse()</code></td>
<td>Returns the entire HTTP response to a file on the disk.</td>
</tr>
<tr>
<td><code>getHeader(name STRING)</code></td>
<td>Returns the value of an HTTP header.</td>
</tr>
<tr>
<td><code>getHeaderCount()</code></td>
<td>Returns the number of headers.</td>
</tr>
<tr>
<td><code>getHeaderName(pos INTEGER)</code></td>
<td>Returns the name of a header by position.</td>
</tr>
<tr>
<td><code>getHeaderValue(pos INTEGER)</code></td>
<td>Returns the value of a header by position.</td>
</tr>
<tr>
<td><code>getStatusCode()</code></td>
<td>Returns the HTTP status code.</td>
</tr>
<tr>
<td><code>getStatusDescription()</code></td>
<td>Returns the HTTP status description.</td>
</tr>
<tr>
<td><code>getTextResponse()</code></td>
<td>Returns the entire HTTP response in a string.</td>
</tr>
<tr>
<td><code>getXmlResponse()</code></td>
<td>Returns the entire HTTP response in a DOM document.</td>
</tr>
</tbody>
</table>
Table 624: Object methods: Multipart methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getMultipartType()</code></td>
<td>Returns whether a response is multipart or not, and the kind of multipart if any.</td>
</tr>
<tr>
<td><code>getPart(pos INTEGER)</code></td>
<td>Returns the HTTP part object at the specified index of the current HTTP response.</td>
</tr>
<tr>
<td><code>getPartCount()</code></td>
<td>Returns the number of additional parts in the HTTP response.</td>
</tr>
<tr>
<td><code>getPartFromContentID(id STRING)</code></td>
<td>Returns the HTTP part object marked with the given Content-ID value as identifier, or NULL if none.</td>
</tr>
</tbody>
</table>

Table 625: Object methods of com.HTTPResponse: Cookie management

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getServerCookies(cookies RECORD)</code></td>
<td>Returns all cookies set as response from a server.</td>
</tr>
</tbody>
</table>

com.HTTPResponse.beginXmlResponse
Starts a streaming HTTP response.

**Syntax**

```java
beginXmlResponse()
RETURNS xml.StaxReader
```

**Usage**

The `beginXmlResponse()` method starts a streaming HTTP response and returns a `xml.StaxReader` object ready to read XML from the server.

The Content-Type header must be of the form */xml or */+xml. For example: application/xhtml+xml. In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 3555.

com.HTTPResponse.endXmlResponse
Performs the HTTP request.

**Syntax**

```java
endXmlResponse()
```
1. *stax* defines an *xml.StaxReader* object used to read the HTTP response.

**Usage**

The `endXmlResponse()` method ends the streaming HTTP response by closing the *xml.StaxReader* object that was created with the `beginXmlResponse()` method.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See [Error handling in GWS calls (STATUS)](page 3554).

The `INT_FLAG` variable is checked during GWS API call to handle program interruptions, for more details, see [Interruption handling in GWS calls (INT_FLAG)](page 3555).

**com.HTTPResponse.getDataResponse**

- Returns the entire HTTP response in a BYTE.

**Syntax**

```java
getDataResponse ( 
    b BYTE )
```

1. `b` defines a BYTE variable receiving the HTTP response data.

**Usage**

The `getDataResponse()` method returns the body of an HTTP response into a BYTE variable.

The BYTE variable must be located in memory, otherwise operation fails.

Returns binary data as response from a server into a BYTE.

Previous content is discarded.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See [Error handling in GWS calls (STATUS)](page 3554).

The `INT_FLAG` variable is checked during GWS API call to handle program interruptions, for more details, see [Interruption handling in GWS calls (INT_FLAG)](page 3555).

**com.HTTPResponse.getFileResponse**

- Returns the entire HTTP response to a file on the disk.

**Syntax**

```java
getFileResponse ( )
RETURNS STRING
```

**Usage**

Reads an HTTP response and creates a file from it.

The method returns the absolute path to the file containing the HTTP response.

The file is created in the temporary directory used by the runtime system (DBTEMP). The name of the file is the basename found in the HTTP Content-Disposition Header. If this basename is not specified, the file name is created with a UUID. If a file with the same name already exists in the temporary directory, the API prefixes the new file with a number. It is then of the form: `/tmp/ABC/filename_index.ext`, where `index` represents the number of files with the same name on disk.
In case of error, the method throws an exception and sets the \texttt{STATUS} variable. Depending on the error, a human-readable description of the problem is available in the \texttt{SQLCA.SQLERRM} register. See Error handling in GWS calls (\texttt{STATUS}) on page 3554.

The \texttt{INT\_FLAG} variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (\texttt{INT\_FLAG}) on page 3555.

\begin{verbatim}
com.HTTPResponse.getHeader
Returns the value of an HTTP header.

\textbf{Syntax}

\begin{verbatim}
getHeader ( name STRING )
  RETURNS STRING
\end{verbatim}

1. \texttt{name} is the name of the parameter that defines the HTTP header.

\textbf{Usage}

The \texttt{getHeader()} method returns the value of the HTTP header specified by the \texttt{name} parameter, or \texttt{NULL} if not found.

In case of error, the method throws an exception and sets the \texttt{STATUS} variable. Depending on the error, a human-readable description of the problem is available in the \texttt{SQLCA.SQLERRM} register. See Error handling in GWS calls (\texttt{STATUS}) on page 3554.

com.HTTPResponse.getHeaderCount
Returns the number of headers.

\textbf{Syntax}

\begin{verbatim}
getHeaderCount ()
  RETURNS INTEGER
\end{verbatim}

\textbf{Usage}

The \texttt{getHeaderCount()} method returns the number of headers of the HTTP response.

In case of error, the method throws an exception and sets the \texttt{STATUS} variable. Depending on the error, a human-readable description of the problem is available in the \texttt{SQLCA.SQLERRM} register. See Error handling in GWS calls (\texttt{STATUS}) on page 3554.

com.HTTPResponse.getHeaderName
Returns the name of a header by position.

\textbf{Syntax}

\begin{verbatim}
getHeaderName ( pos INTEGER )
  RETURNS STRING
\end{verbatim}

1. \texttt{pos} specifies the ordinal position of the header.

\textbf{Usage}

The \texttt{getHeaderName()} method returns the name of the HTTP response header depending on the position passed as parameter.
In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

com.HTTPResponse.getHeaderValue
Returns the value of a header by position.

**Syntax**

```java
getHeaderValue ( 
   pos INTEGER )
RETURNS STRING
```

1. *pos* specifies the ordinal position of the header.

**Usage**

The `getHeaderValue()` method returns the value of the HTTP response header based on the position passed as parameter.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

com.HTTPResponse.getMultipartType
Returns whether a response is multipart or not, and the kind of multipart if any.

**Syntax**

```java
getMultipartType ()
RETURNS STRING
```

**Usage**

Returns whether a response is multipart or not, and the kind of multipart if any.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**Related concepts**

Examples: Using the com.HTTPPart class on page 2910
Examples using methods of the com.HTTPPart class.

com.HTTPResponse.getPartCount
Returns the number of additional parts in the HTTP response.

**Syntax**

```java
getPartCount ()
RETURNS INTEGER
```

**Usage**

Returns the number of additional parts in the HTTP response. The root part element must be handled via `getXmlResponse().getTextResponse().getDataResponse()` and `beginXmlResponse()`. In other words, there are `getPartCount() +1` parts if `getMultipartType()` does not return NULL.
In case of error, the method throws an exception and sets the \texttt{STATUS} variable. Depending on the error, a human-readable description of the problem is available in the \texttt{SQLCA.SQLERRM} register. See Error handling in GWS calls (\texttt{STATUS}) on page 3554.

**Related concepts**
- Examples: Using the \texttt{com.HTTPPart} class on page 2910
- Examples using methods of the \texttt{com.HTTPPart} class.

\texttt{com.HTTPResponse.getPart}

Returns the HTTP part object at the specified index of the current HTTP response.

**Syntax**

\begin{verbatim}
getPart (  
pos INTEGER )  
RETURNS \texttt{com.HTTPPart}
\end{verbatim}

1. \texttt{pos} is the input parameter that specifies the index number.

**Usage**

Returns the HTTP part object at the specified index of the current HTTP response.

In case of error, the method throws an exception and sets the \texttt{STATUS} variable. Depending on the error, a human-readable description of the problem is available in the \texttt{SQLCA.SQLERRM} register. See Error handling in GWS calls (\texttt{STATUS}) on page 3554.

Can raise error -15554 (Index is out of bounds).

**Related concepts**
- Examples: Using the \texttt{com.HTTPPart} class on page 2910
- Examples using methods of the \texttt{com.HTTPPart} class.

\texttt{com.HTTPResponse.getPartFromContentID}

Returns the HTTP part object marked with the given Content-ID value as identifier, or NULL if none.

**Syntax**

\begin{verbatim}
getPartFromContentID (  
id STRING )  
RETURNS \texttt{com.HTTPPart}
\end{verbatim}

1. \texttt{id} defines the HTTP header.

**Usage**

Returns the HTTP part object marked with the given Content-ID value as identifier, or NULL if none.

In case of error, the method throws an exception and sets the \texttt{STATUS} variable. Depending on the error, a human-readable description of the problem is available in the \texttt{SQLCA.SQLERRM} register. See Error handling in GWS calls (\texttt{STATUS}) on page 3554.

\texttt{com.HTTPResponse.getServerCookies}

Returns all cookies set as response from a server.

**Syntax**

\begin{verbatim}
getServerCookies (  
cookies RECORD )
\end{verbatim}
1. *cookies* defines the dynamic array of all cookies sent as a response from a server. See WSHelper variables and records on page 3500 for more information regarding WSServerCookiesType.

**Usage**

This method returns all cookies set as response from a server, in a dynamic array. All cookies are returned, even those that have already expired.

If the method `setAutoCookies()` is enabled, cookies are automatically send back if they have not expired. In other words, the `setAutoCookies(true)` handles all cookies for you; however, you can still check them with the `getServerCookies()` method if needed.

If `setAutoCookies(false)` (the default), you must handle the cookies and not send the expired ones.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (`STATUS`) on page 3554.

This method may raise exception -15535 if the given dynamic array is not a dynamic array of RECORD with seven (7) members as defined by WSHelper.WSServerCookiesType.

**Related concepts**

- com.HTTPRequest.setAutoCookies on page 2884
  Enables automatic cookie management for a given request.

- com.HTTPResponse.getStatusCode
  Returns the HTTP status code.

  **Syntax**

  ```java
  getStatusCode()
  RETURNS INTEGER
  ```

  **Usage**

  The `getStatusCode()` method returns the status code for the HTTP response.

  When the returned HTTP status code is 401 or 407, authorization is required.

  In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (`STATUS`) on page 3554.

  **Related concepts**

  - com.HTTPRequest.setAuthentication on page 2883
    Defines the user login and password to authenticate to the server.

- com.HTTPResponse.getStatusDescription
  Returns the HTTP status description.

  **Syntax**

  ```java
  getStatusDescription()
  RETURNS STRING
  ```

  **Usage**

  The `getStatusDescription()` method returns a description of the HTTP response status.
In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

com.HTTPResponse.getTextResponse
Returns the entire HTTP response in a string.

**Syntax**

```java
getTextResponse ()
   RETURNS STRING
```

**Usage**

The `getTextResponse()` method returns a HTTP response as an entire string.

- The Content-Type header can be of the form */* or */*. For example: application/json.
- Automatic conversion to the locale charset is performed when possible, otherwise throws an exception.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 3555.

com.HTTPResponse.getXmlResponse
Returns the entire HTTP response in a DOM document.

**Syntax**

```java
getXmlResponse ()
   RETURNS xml.DomDocument
```

**Usage**

The `getXmlResponse()` method returns an HTTP response in a `xml.DomDocument` object.

The Content-Type header must be of the form */xml or *//*+xml. For example: application/xhtml+xml.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 3555.

**Examples**

Examples using methods of the `com.HTTPResponse` class.

For examples, see Examples using `com.HTTPRequest` methods on page 2891.

**The HTTPPart class**

The `com.HTTPPart` class provides an interface to manage the HTTP attachment sent or received in HTTP.

The STATUS variable is set to zero after a successful method call.
**com.HTTPPart methods**
Methods for the `com.HTTPPart` class.

**Table 626: Class methods of com.HTTPPart**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CreateAttachment (filename STRING)</code></td>
<td>Creates a new HTTPPart object based on a given file name located on disk.</td>
</tr>
<tr>
<td></td>
<td>RETURNS <code>com.HTTPPart</code></td>
</tr>
<tr>
<td><code>CreateFromData (b BYTE)</code></td>
<td>Creates a new HTTPPart object based on given BYTE located in memory.</td>
</tr>
<tr>
<td></td>
<td>RETURNS <code>com.HTTPPart</code></td>
</tr>
<tr>
<td></td>
<td>RETUNS <code>com.HTTPPart</code></td>
</tr>
<tr>
<td><code>CreateFromString (str STRING)</code></td>
<td>Creates a new HTTPPart object based on given string.</td>
</tr>
<tr>
<td></td>
<td>RETURNS <code>com.HTTPPart</code></td>
</tr>
</tbody>
</table>
Table 627: Object methods of com.HTTPPart

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clearHeaders()</td>
<td>Remove all headers from the HTTP part.</td>
</tr>
<tr>
<td>getAttachment()</td>
<td>Returns the absolute path to the HTTP part.</td>
</tr>
<tr>
<td>getContentAsData( b BYTE )</td>
<td>Returns the HTTP part as a BYTE.</td>
</tr>
<tr>
<td>getContentAsDomDocument()</td>
<td>Returns the HTTP part as an XML document.</td>
</tr>
<tr>
<td>getHeader( name STRING )</td>
<td>Setter to handle HTTP multipart headers.</td>
</tr>
<tr>
<td>getContentAsString()</td>
<td>Returns the HTTP part as a string.</td>
</tr>
<tr>
<td>removeHeader( name STRING )</td>
<td>Remove the header of given name from the current HTTPPart object.</td>
</tr>
<tr>
<td>setHeader( name STRING, value STRING )</td>
<td>Setter to handle HTTP multipart headers.</td>
</tr>
</tbody>
</table>

com.HTTPPart.CreateFromString

Creates a new HTTPPart object based on given string.

**Syntax**

```
CreateFromString(
    str STRING )
RETURNS com.HTTPPart
```

1. `str` specifies a string value.

**Usage**

Creates a new HTTPPart object based on given string. To be used via the `addPart()` method.

Defaults HTTP multipart headers:

- Content-Type: text/plain
- Content-Transfer-Encoding: 8bits

Notice that the string will be converted during request sending into ISO-8859-1 by default, unless a different charset has been set via `setHeader("Content-Type","text/plain; charset=UTF-8")` for instance.
In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

com.HTTPPart.CreateFromDomDocument
Creates a new HTTPPart object based on given XML document.

**Syntax**

```java
CreateFromDomDocument (doc xml.DomDocument )
RETURNS com.HTTPPart
```

1. `doc` specifies an XML document.

**Usage**

Creates a new HTTPPart object based on given XML document. To be used via the `addPart()` method.

Defaults HTTP multipart headers:

- Content-Type: text/xml; charset=UTF-8
- Content-Transfer-Encoding: 8bits

A different charset can be set with the `setHeader` method. For example, `setHeader("Content-Type","text/plain; charset=ISO-8859-1")` sets the charset to ISO-8859-1.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**Related concepts**

- **com.HTTPPart.setHeader** on page 2909
  Setter to handle HTTP multipart headers.

com.HTTPPart.CreateFromData
Creates a new HTTPPart object based on given BYTE located in memory.

**Syntax**

```java
CreateFromData (b BYTE )
RETURNS com.HTTPPart
```

1. `b` defines a BYTE object located in memory.

**Usage**

Creates a new HTTPPart object based on given BYTE located in memory. To be used via the `addPart()` method.

Defaults HTTP headers:

- Content-Type: application/octet-stream
- Content-Transfer-Encoding: binary

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

com.HTTPPart.CreateAttachment
Creates a new HTTPPart object based on a given file name located on disk.

**Syntax**

```java
CreateAttachment(
    filename STRING )
RETURNS com.HTTPPart
```

1. *filename* specifies the name of a file.

**Usage**

Creates a new HTTPPart object based on a given file name located on disk. To be used via the `addPart()` method.

The `com.HTTPPart.CreateAttachment()` method automatically sets the following headers for the created HTTPPart object:

- **Content-Type** is defined based on the file name extension. If the file extension is not recognized, `Content-Type` defaults to `application/octet-stream`.
  
  **Note:** File extensions to `Content-Type` mapping can be customized in the file `$FGLDIR/lib/wse/mime.cfg`.

- **Content-Transfer-Encoding** is set to "binary".

- **Content-Disposition** is set with the base name of the given `filename` as follows: `attachment; filename="basename"`.

For example, when calling the method as follows:

```java
LET part = com.HTTPPart.CreateAttachment( "/opt/myapp/resources/logo.jpg" )
```

The resulting HTTP part headers will look like:

```
Content-Type: image/jpeg
Content-Transfer-Encoding: binary
Content-Disposition: attachment; filename="logo.jpg"
```

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**Related concepts**

- Examples: Using the `com.HTTPPart` class on page 2910
- Examples using methods of the `com.HTTPPart` class.

**clearHeaders**

Remove all headers from the HTTP part.

**Syntax**

```java
clearHeaders()
```

**Usage**

This method removes all headers from an HTTP multipart object.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.
Example

```call
CALL req.clearHeaders()
```

Related concepts

- com.HTTPPart.removeHeader on page 2909
  - Remove the header of given name from the current HTTPPart object.

- com.HTTPPart.getAttachment
  - Returns the absolute path to the HTTP part.

Syntax

```call
getAttachment ()
RETURNS STRING
```

Usage

Returns the absolute path location of the received part file.

The file is created in the temporary directory used by the runtime system (DBTEMP). The name of the file is the basename found in the HTTP Content-Disposition Header. If this basename is not specified, the file name is created with a UUID. If a file with the same name already exists in the temporary directory, the API prefixes the new file with a number. It is then of the form: `/tmp/ABC/filename_index.ext`, where `index` represents the number of files with the same name on disk.

If the file is encoded in base64, you can use the Genero Web Services `fglpass -dec64` command to convert it back to binary.

It is up to the programmer to remove the file from the disk when it is no longer needed.

To be used via methods: com.HTTPResponse.getPart on page 2900, com.HTTPResponse.getPartCount on page 2899, and com.HTTPResponse.getPartFromID on page 2900

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

- com.HTTPPart.getContentAsData
  - Returns the HTTP part as a BYTE.

Syntax

```param
getContentAsData ( 
  b BYTE )
```

1. `b` defines a variable holding the BYTE data.

Usage

Returns the HTTP part as a BYTE. BYTE data cannot be returned from a function with a RETURN statement. Therefore, the `b` parameter must be handled by reference.

To be used via methods: com.HTTPResponse.getPart on page 2900, com.HTTPResponse.getPartCount on page 2899, and com.HTTPResponse.getPartFromID on page 2900

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

The `error -15573` is raised if the part cannot be converted to a Genero BYTE.
com.HTTPPart.getContentAsDomDocument
Returns the HTTP part as an XML document.

Syntax
```
getContentAsDomDocument ()
  RETURNS xml.DomDocument
```

Usage
Returns the HTTP part as an XML document.

To be used via methods: com.HTTPResponse.getPart on page 2900, com.HTTPResponse.getPartCount on page 2899, and com.HTTPResponse.getPartFromID on page 2900

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

The error -15573 is raised if the part cannot be converted to a XML DomDocument.

com.HTTPPart.getContentAsString
Returns the HTTP part as a string.

Syntax
```
getContentAsString ()
  RETURNS STRING
```

Usage
Returns the HTTP part as a string.

To be used via methods: com.HTTPResponse.getPart on page 2900, com.HTTPResponse.getPartCount on page 2899, and com.HTTPResponse.getPartFromID on page 2900

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

The error -15573 is raised if the part cannot be converted to a Genero string or if the charset is not supported.

com.HTTPPart.getHeader
Setter to handle HTTP multipart headers.

Syntax
```
getHeader ()
  name STRING
  RETURNS STRING
```

1.  name specifies the name of the header part.

Usage
Getter to handle HTTP multipart headers.

The method returns the value for the header part specified by name.

Note: In case of related multipart (i.e., the part is multipart/related and set via the com.HTTPRequest.setMultipartType("related",NULL,NULL)), it is mandatory
to set a unique Content-ID header. To set up a unique Content-ID header, you can use the
security.RandomGenerator.CreateUUIDString on page 3128 method for that.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

Example

```
LET val = req.getHeader("MyClientHeader")
```

Related concepts

com.HTTPPart.setHeader on page 2909
Setter to handle HTTP multipart headers.

com.HTTPPart.removeHeader
Remove the header of given name from the current HTTPPart object.

Syntax

```
removeHeader (  
    name STRING  )
```

1. `name` is the name of the header to remove.

Usage

With an HTTP multipart object, the "Content-Type" header may be optional for each part:

```
https://tools.abc.org/html/rfc1234#section-1.1
```

A request might be rejected by some providers if the "Content-Type" header is set for a part. This method allows you to remove headers from an HTTPPart object based on the header name.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

Example

```
CALL req.removeHeader("Content-Type")
```

Related concepts

com.HTTPPart.clearHeaders on page 2906
Remove all headers from the HTTP part.

com.HTTPPart.setHeader
Setter to handle HTTP multipart headers.

Syntax

```
setHeader (  
    name STRING,  
    value STRING  )
```

1. `name` specifies the multipart header name.
2. `value` specifies the multipart header value (such as HTTP headers).
Usage
Setter to handle HTTP multipart headers.

For instance, when you send a multipart image, it is recommended that you specify the image mime type with this header method. If the image is a .png, you have to do `part.setHeader("Content-Type","image/png")`, which allows the peer to know the format of the attached file it has to process.

**Note:** In case of related multipart (i.e., the part is multipart/related and set via the `com.HTTPRequest.setMultipartType("related",NULL,NULL)`), it is mandatory to set a unique Content-ID header. To set up a unique Content-ID header, you can use the `security.RandomGenerator.CreateUUIDString` on page 3128 method for that.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

Example

```java
CALL req.setHeader("MyClientHeader","Hello")
```

Related concepts

`com.HTTPPart.getHeader` on page 2908
Setter to handle HTTP multipart headers.

**Examples: Using the `com.HTTPPart` class**
Examples using methods of the `com.HTTPPart` class.

This example consists of two applications: a client and server exchanging an XML document in multipart with an image as an attachment.

**Client Application**

```java
IMPORT com
IMPORT xml

CONSTANT SERVER_URL = "http://localhost:8090/MultipartMixed/Sample"

MAIN

DEFINE req com.HTTPRequest
DEFINE resp com.HTTPResponse
DEFINE doc xml.DomDocument
DEFINE p com.HTTPPart
DEFINE type STRING
DEFINE ind INTEGER

LET req = com.HTTPRequest.Create(SERVER_URL)
CALL req.setMethod("POST")
CALL req.setHeader("MyClientHeader","Hello")
TRY
  # Set multipart type
  CALL req.setMultipartType("mixed",NULL,NULL)
  # Add filename as part
  LET p = com.HTTPPart.CreateAttachment("my_picture.png")
  # Set attachment Content-Type
  CALL p.setHeader("Content-Type","image/png")
  # Add part to the request
  CALL req.addPart(p)
  # Perform XML request
```
CALL req.doXmlRequest(doc)
# Check response
LET resp=req.getResponse()
IF resp.getStatusCode() != 200 THEN
  DISPLAY "HTTP Error ("||resp.getStatusCode()||") ",
  resp.getStatusDescription()
  EXIT PROGRAM (-1)
END IF
IF resp.getStatusDescription() != "OK" THEN
  DISPLAY "HTTP Error ("||resp.getStatusCode()||") ",
  resp.getStatusDescription()
  EXIT PROGRAM (-1)
END IF
# Check whether multipart response or not
LET type = resp.getMultipartType()
IF type IS NULL THEN
  DISPLAY "Failed : Expected multipart in response"
  EXIT PROGRAM (-1)
ELSE
  DISPLAY "Response is multipart of ":,type
END IF
# Check response
LET doc = resp.getXmlResponse()
IF doc IS NULL THEN
  DISPLAY "Expected XML document as response"
  EXIT PROGRAM (-1)
ELSE
  DISPLAY "Response is ":,doc.saveToString()
END IF
# Process additional parts
FOR ind = 1 TO resp.getPartCount()
  LET p = resp.getPart(ind)
  IF p.getAttachment() IS NOT NULL THEN
    DISPLAY "Attached file at ":,p.getAttachment()
  ELSE
    DISPLAY "Attached part is ":,p.getContentAsString()
  END IF
END FOR
CATCH
  DISPLAY "unexpected exception ":,STATUS," ("||SQLCA.SQLERRM||")"
END TRY
END MAIN

Server Application

IMPORT com
IMPORT xml
MAIN
DEFINE req com.HTTPServiceRequest
DEFINE url STRING
DEFINE method STRING
DEFINE doc xml.DomDocument
DEFINE type STRING
DEFINE ind INTEGER
DEFINE p com.HTTPPart
CALL com.WebServiceEngine.Start()
LET req = com.WebServiceEngine.getHTTPServiceRequest(-1)
LET url = req.getURL()
IF url IS NULL THEN
    DISPLAY "Failed: url should not be null"
    EXIT PROGRAM (-1)
END IF
LET method = req.getMethod()
IF method IS NULL OR method != "POST" THEN
    DISPLAY "Failed: method should be POST"
    EXIT PROGRAM (-1)
END IF
# Check multipart type
LET type = req.getRequestMultipartType()
IF type IS NULL THEN
    DISPLAY "Failed: expected multipart in request"
    EXIT PROGRAM (-1)
END IF
TRY
    LET doc = req.readXMLRequest()
    DISPLAY "Request is :", doc.saveToString()
CATCH
    DISPLAY "Failed: unexpected error :", STATUS
    EXIT PROGRAM (-1)
END TRY
# Process additional parts
FOR ind = 1 TO req.getRequestPartCount()
    LET p = req.getRequestPart(ind)
    IF p.getAttachment() IS NOT NULL THEN
        DISPLAY "Attached file at :", p.getAttachment()
    ELSE
        DISPLAY "Attached part is :", p.getContentAsString()
    END IF
END FOR
# Set multipart response type
CALL req.setResponseMultipartType("mixed",NULL,NULL)
# Add XML Part
LET p = com.HTTPPart.CreateAttachment("my_other_picture.jpg")
CALL p.setHeader("Content-Type","image/jpg")
CALL req.addResponsePart(p)
CALL req.sendXmlResponse(200,NULL,doc)
END MAIN

Related concepts
com.HTTPPart methods on page 2903
Methods for the com.HTTPPart class.
com.HTTPServiceRequest methods on page 2846
Methods of the com.HTTPServiceRequest class.

TCP classes
The TCP classes manage TCP client network operations.
• CLASS TCPRequest
• CLASS TCPResponse

The TCPRequest class
The com_TCPRequest class provides an interface to perform asynchronous XML and TEXT requests over TCP, with additional XML streaming possibilities.

Important: This Web Services class is not supported on GMI mobile devices.
**com.TCPRequest methods**
Methods of the `com.TCPRequest` class.

**Table 628: Class methods of com.TCPRequest**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>com.TCPRequest.Create (uri STRING )</code></td>
<td>Creates a new TCP request object.</td>
</tr>
<tr>
<td>RETURNS <code>com.TCPRequest</code></td>
<td></td>
</tr>
</tbody>
</table>
### Table 629: Object methods of com.TCPRequest

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>beginXmlRequest()</code></td>
<td>Starts a streaming XML request.</td>
</tr>
<tr>
<td>RETURNs <code>xml.StaxWriter</code></td>
<td></td>
</tr>
<tr>
<td><code>doDataRequest(data BYTE)</code></td>
<td>Performs the request by sending binary data.</td>
</tr>
<tr>
<td><code>doRequest()</code></td>
<td>Performs a TCP request.</td>
</tr>
<tr>
<td><code>doTextRequest(str STRING)</code></td>
<td>Performs a request with a string.</td>
</tr>
<tr>
<td><code>doXmlRequest(doc xml.DomDocument)</code></td>
<td>Performs a request with a DOM document.</td>
</tr>
<tr>
<td><code>endXmlRequest(stax xml.StaxWriter)</code></td>
<td>Terminates a streaming TCP request.</td>
</tr>
<tr>
<td><code>getAsyncResponse()</code></td>
<td>Returns the response after performing a TCP request, asynchronously.</td>
</tr>
<tr>
<td>RETURNs <code>com.TCPResponse</code></td>
<td></td>
</tr>
<tr>
<td><code>getResponse()</code></td>
<td>Returns the response after performing a TCP request.</td>
</tr>
<tr>
<td>RETURNs <code>com.TCPResponse</code></td>
<td></td>
</tr>
<tr>
<td><code>setConnectionTimeOut(timeout INTEGER)</code></td>
<td>Defines the connection time out.</td>
</tr>
<tr>
<td><code>setKeepConnection(keep INTEGER)</code></td>
<td>Defines if the TCP connection is kept open after sending a request.</td>
</tr>
<tr>
<td><code>setMaximumResponseLength(length INTEGER)</code></td>
<td>Defines the maximum size in Kbyte of the response.</td>
</tr>
<tr>
<td><code>setTimeOut(timeout INTEGER)</code></td>
<td>Defines the time out for read/write operations.</td>
</tr>
</tbody>
</table>

**Syntax**

```java
com.TCPRequest.beginXmlRequest
   Starts a streaming XML request.
```

```java
beginXmlRequest()
   RETURNS `xml.StaxWriter`
```
**Usage**

The `beginXmlRequest()` method begins a streaming HTTP request and returns an `xml.StaxWriter` object ready to send XML to the server.

After sending all the XML data to the server, you must call the `endXmlRequest()` method with the `xml.StaxWriter` object created by the `beginXmlRequest()` method.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

The `INT_FLAG` variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 3555.

**com.TCPRequest.Create**

Creates a new TCP request object.

**Syntax**

```java
com.TCPRequest.Create(
    uri STRING )
RETURNS com.TCPRequest
```

1. `uri` specifies the URL of the TCP request.

**Usage**

This class method creates a new `com.TCPRequest` object based on the URL passed as parameter.

The URL must use the TCP or TCPS protocol. Examples of valid URLs include:

- `tcp://localhost:4242/`
- `tcps://localhost:4343/`

The URL can be an identifier of an URL mapping with an optional `alias://` prefix. See FGLPROFILE configuration for more details about URL mapping with aliases, and for proxy and security configuration.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**com.TCPRequest.doDataRequest**

Performs the request by sending binary data.

**Syntax**

```java
doDataRequest(
    data BYTE )
```

1. `data` defines the binary data to be send for a TCP request. The `data` variable must be located IN MEMORY.

**Usage**

Performs the TCP request by sending binary data contained in the `BYTE` variable.

**Note:** The `data` variable must be located IN MEMORY.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**com.TCPRequest.doRequest**
Performs a TCP request.

**Syntax**

```java
doRequest ()
```

**Usage**

The `doRequest()` method performs the TCP request.

The connection is shutdown for writing, to confirm that no data will be sent.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

The `INT_FLAG` variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 3555.

**Example**

```java
IMPORT com
IMPORT XML

MAIN
  DEFINE url STRING
  LET url = "tcp://localhost:4242"
  CALL an_example(url)
END MAIN

FUNCTION an_example(url)
  DEFINE url STRING
  DEFINE req com.TCPRequest
  DEFINE resp com.TCPResponse
  DEFINE ret xml.DomDocument
  TRY
    LET req = com.TCPRequest.create(url)
    CALL req.doRequest()
    LET resp = req.getResponse()
    LET ret = resp.getXmlResponse()
  CATCH
    DISPLAY "ERROR : ", STATUS, SQLCA.SQLERRM
    EXIT PROGRAM(-1)
  END TRY
END FUNCTION
```

**com.TCPRequest.doXmlRequest**

Performs a request with a DOM document.

**Syntax**

```java
doXmlRequest (  
  doc xml.DomDocument  )
```

1. `doc` specifies the DOM document describing the request.
**Usage**

The `doXmlRequest()` method performs the TCP request by using the information defined in the `xml.DomDocument` object passed as parameter.

The connection is shutdown for writing, to confirm that no data will be sent.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

The `INT_FLAG` variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 3555.

```
com.TCPRequest.doTextRequest
Performs a request with a string.
```

**Syntax**

```
doTextRequest ( str STRING )
```

1. `str` defines the string describing the request.

**Usage**

The `doTextRequest()` method performs the TCP request by using the information defined in the string passed as parameter.

The connection is shutdown for writing, to confirm that no data will be sent.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

The `INT_FLAG` variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 3555.

```
com.TCPRequest.endXmlRequest
Terminates a streaming TCP request.
```

**Syntax**

```
endXmlRequest ( stax xml.StaxWriter )
```

1. `stax` specifies the `xml.StaxWriter` object used for streaming.

**Usage**

The `endXmlRequest()` method terminates a streaming TCP request performed with the `xml.StaxWriter` object that was created by the `beginXmlRequest()` method.

The connection is shutdown for writing, to confirm that no data will be sent.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

The `INT_FLAG` variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 3555.

```
com.TCPRequest.getResponse
```
Returns the response after performing a TCP request.

Syntax

```java
getResponse()
RETURNS com.TCPResponse
```

Usage

The `getResponse()` method returns a TCP response as a `com.TCPResponse` object, after a call to `doRequest()`, `doXmlRequest()`, `doTextRequest()`, or `beginXmlRequest() / endXmlRequest()` calls.

A call to this method will stop the program flow until the response is received.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

The `INT_FLAG` variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 3555.

com.TCPRequest.getAsyncResponse

Returns the response after performing a TCP request, asynchronously.

Syntax

```java
getAsyncResponse()
RETURNS com.TCPResponse
```

Usage

The `getAsyncResponse()` method returns a TCP response as a `com.TCPResponse` object, after a call to `doRequest()`, `doXmlRequest()`, `doTextRequest()`, or `beginXmlRequest() / endXmlRequest()` calls.

Unlike `getResponse()`, the `getAsyncResponse()` method does not stop the program flow: The method returns `NULL` if the response was not yet received.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

The `INT_FLAG` variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 3555.

com.TCPRequest.setTimeOut

Defines the time out for read/write operations.

Syntax

```java
setTimeout (timeout INTEGER )
```

1. `timeout` specifies a time out value in seconds.

Usage

This method defines a value in seconds to wait for a read or write operation to complete, before a break.
If the time out is set to -1, it waits indefinitely.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

com.TCPRequest.setConnectionTimeOut
Defines the connection time out.

**Syntax**

```
setConnectionTimeOut(
  timeout INTEGER )
```

1. `timeout` specifies a time out value in seconds.

**Usage**

This method defines the time in seconds to wait for a connection, before a break.

If the time out is set to -1, it waits indefinitely.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

com.TCPRequest.setKeepConnection
Defines if the TCP connection is kept open after sending a request.

**Syntax**

```
setKeepConnection(
  keep INTEGER )
```

1. `keep` indicates if the TCP connection must be kept open.

**Usage**

This method can be used to force the TCP socket to remain open after a send operation, in order to perform subsequent do*Request() calls, without closing the connection (in write mode).

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

com.TCPRequest.setMaximumResponseLength
Defines the maximum size in Kbyte of the response.

**Syntax**

```
setMaximumResponseLength(
  length INTEGER )
```

1. `length` specifies the max size of a response in Kbytes.

**Usage**

This method sets the maximum authorized size in Kbyte of the whole response, before a break.

A length of -1 defines no limit.
In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

The TCPResponse class
The com.TCPResponse class provides an interface to perform XML and TEXT responses over TCP, with additional XML streaming possibilities.

Important: This Web Services class is not supported on GMI mobile devices.

com.TCPResponse methods
Methods of the com.TCPResponse class.

Table 630: Object methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>beginXmlResponse()</td>
<td>Starts a streaming TCP response.</td>
</tr>
<tr>
<td></td>
<td>RETURNS xml.StaxReader</td>
</tr>
<tr>
<td>endXmlResponse()</td>
<td>Ends a streaming TCP response.</td>
</tr>
<tr>
<td></td>
<td>stax xml.StaxReader</td>
</tr>
<tr>
<td>getDataResponse()</td>
<td>Returns a TCP response in binary format.</td>
</tr>
<tr>
<td></td>
<td>data BYTE</td>
</tr>
<tr>
<td>getTextResponse()</td>
<td>Returns a TCP response in string format.</td>
</tr>
<tr>
<td></td>
<td>RETURNS STRING</td>
</tr>
<tr>
<td>getXmlResponse()</td>
<td>Returns an entire DOM document as TCP response.</td>
</tr>
<tr>
<td></td>
<td>RETURNS xml.DomDocument</td>
</tr>
</tbody>
</table>

com.TCPResponse.beginXmlResponse
Starts a streaming TCP response.

Syntax

beginXmlResponse()
RETURNS xml.StaxReader

Usage

 Begins the streaming TCP response and returns an xml.StaxReader object ready to read XML from the server.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 3555.

com.TCPResponse.endXmlResponse
Ends a streaming TCP response.

**Syntax**

```java
endXmlResponse(  
    stax xml.StaxReader  )
```

1. `stax` defines the `xml.StaxReader` object created with `beginXmlResponse()`.

**Usage**

Terminates the streaming TCP response identified by the `xml.StaxReader` object passed as parameter. This object must have been created with the `beginXmlResponse()` method.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

The `INT_FLAG` variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 3555.

**com.TCPResponse.getDataResponse**

Returns a TCP response in binary format.

**Syntax**

```java
dataGetResponse(  
    data BYTE  )
```

1. `data` defines a `BYTE` variable that will hold the response data in binary format. The `BYTE` variable must be located IN MEMORY.

**Usage**

This method retrieves the TCP response in binary format into the `data` variable passed as parameter. The method will read the TCP stream, until the peer closes the connection.

**Note:** The `data` variable must be located IN MEMORY.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

The `INT_FLAG` variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 3555.

**com.TCPResponse.getTextResponse**

Returns a TCP response in string format.

**Syntax**

```java
getTextResponse()  
    RETURNS STRING
```

**Usage**

This method returns a complete streaming TCP response from the server as a string.
In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 3555

com.TCPResponse.getTransaction
Returns an entire DOM document as TCP response.

Syntax

getXmlResponse()
RETURNS xml.DomDocument

Usage

This method returns a complete xml.DomDocument as streaming TCP response from the server.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 3555

Helper classes

The Helper classes provide utility classes.

- The Util class on page 2922

The Util class

The com.Util class provides static helper methods.

This class does not have to be instantiated.

Important: This Web Services class is not supported on GMI mobile devices.

com.Util methods

Methods of the com.Util class.

Table 631: Class methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.Util.UniqueApplicationInstance(file STRING)</td>
<td>Checks that the calling application is the only one to run.</td>
</tr>
<tr>
<td>_returns INTEGER</td>
<td></td>
</tr>
</tbody>
</table>

com.Util.UniqueApplicationInstance
Checks that the calling application is the only one to run.

Syntax

com.Util.UniqueApplicationInstance(file STRING)
>Returns INTEGER

1. file specifies the string for the lock file.
Usage
This method checks that the calling application is the only one to run, by trying to get an exclusive lock on the given
file.

If the lock could be set, the method returns TRUE. Otherwise, it returns FALSE and updates STATUS with an error
code.

Specific classes
Several classes support specific features.

• The APNS class on page 2923

The APNS class
The com.APNS class implements Apple® Push Notification Service APIs.

The com.APNS class implements a set of methods to build and handle push messages to be broadcast by the Apple
Push Notification service.

Related concepts
Apple Push Notification Service (APNs) on page 3625
Follow this procedure to implement push notification with APNs.

APNs SSL/TLS certificate
Get and configure an SSL/TLS certificate to establish secure connections to the Apple Push Notification service.

Basics
The Apple® Push Notification Certificate identifies the push notification service for a given mobile app. This
certificate is created from an App ID (also known as Bundle ID) and is used by the APNs system to dispatch the
notification message to the registered devices.

You can create two types of APN certificates for a given App ID:

• Sandbox (for development and test purpose)
• Production (for deployment)

An APNS push notification provider or an APNS feedback handler needs to establish a secure connection to Apple's
APNs server.

Get an APNs certificate for your app
In this section we will produce the myapp.cer file and myapp-key.p12 file.

To create an Apple® Push Notification Certificate:

1. Log in to Apple's Member Center with you iOS developer or enterprise account.
2. Select Certificates, Identifiers & Profiles.
3. Under App IDs, make sure that you have created an App ID with the Push Notification service enabled, for
development and/or distribution.
4. Under Certificates, select the + symbol.
5. Select Apple® Push Notification service SSL (Sandbox) for development, or Apple® Push Notification service
SSL (Sandbox & Production) for production.
6. Choose the App ID with push notifications service enabled.
7. Follow the instructions to create a Certificate Signing Request (CSR) file from your Mac, then click Continue.
8. Back in the web browser and IOS Certificate page, upload the CSR file you have generated.
9. Generate the certificate.
10. Once the certificate is generated, download it. The certificate will be downloaded into your Downloads folder, as a
    .cer file (for example, myapp.cer).
11. Double-click this file to import the certificate into the Mac® Keychain®.
12. The new certificate is now listed in the Certificates list.
13. Open your Keychain® app and locate the certificate you created, export the private key in p12 format (for example, myapp-key.p12). Note that you will be asked for a password to encode the .p12 file, and for your session password, to exported Keychain® files.

**Configure Genero to use the APNs certificate**

On the Genero push provider server, you will need the public certificate (myapp.crt file) and the private key (myapp-key.pem file) for your app. These files will be referenced in the security.global.certificate and security.global.privatekey entries of FGLPROFILE.

In order to authenticate the APNs server, you will also need the root certificate authority (apple_entrust_root_certification_authority.pem), that can be downloaded from Apple's web site. This file will be referenced by the security.global.ca entry in FGLPROFILE.

**Note:** When executing on a Mac, the root certificate (security.global.ca entry) is not required: The Web Services library reads the Keystore of the Mac® computer, to authenticate the APNs server.

Create the myapp.crt file (public certificate) from the myapp.cer file, with the openssl x509 command:

```
$ openssl x509 -in myapp.cer -inform der -out myapp.crt
```

Convert the myapp-key.p12 file (containing the private key) to a myapp-key.pem format, with the openssl pkcs12 command:

```
$ openssl pkcs12 -nocerts -in myapp-key.p12 -out myapp-key.pem
```

**Note:** You need to enter the passphrase for the .p12 file so that openssl can read it. Then you need to enter a new passphrase that will be used to encrypt the .pem file.

The FGLPROFILE entry security.global.privatekey expects unencrypted private key files. Therefore, you need to remove the passphrase from the myapp-key.pem file, with the openssl rsa command:

```
$ openssl rsa -in myapp-key.pem -out myapp-key-noenc.pem
```

Set up your FGLPROFILE with the appropriate **security.* entries:**

```
security.global.ca                   = "apple_entrust_root_certification_authority.pem"
security.global.certificate        = "myapp.crt"
security.global.privatekey          = "myapp-key-noenc.pem"
```

In the above example:

1. apple_entrust_root_certification_authority.pem is the HTTPS root certificate authenticating the APNs server (if the computer is not a Mac).
2. myapp.crt is the public certificate for your app.
3. myapp-key-noenc.pem is the unencrypted private key file for your app.

**Note:** If you want to keep the private key encrypted, you need to configure a password agent, as described in Using the password agent
**com.APNS methods**

Methods of the **com.APNS** class.

**Table 632: Class methods**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>com.APNS.DecodeError</strong> (data BYTE)</td>
<td>Decodes content of BYTE data returned from the APNS server in case of error.</td>
</tr>
<tr>
<td>Returns (uuid STRING, error INTEGER)</td>
<td></td>
</tr>
<tr>
<td><strong>com.APNS.DecodeFeedback</strong> (data BYTE, unregs RECORD)</td>
<td>Decodes content of BYTE data returned from the APNS feedback service.</td>
</tr>
<tr>
<td><strong>com.APNS.EncodeMessage</strong> (data BYTE, deviceIdBase64 STRING, json STRING, uuidBase64 STRING, expiration INTEGER, priority INTEGER)</td>
<td>Encodes an APNS specific push notification message into a BYTE.</td>
</tr>
<tr>
<td>Returns INTEGER</td>
<td></td>
</tr>
</tbody>
</table>

**com.APNS.DecodeError**

Decodes content of BYTE data returned from the APNS server in case of error.

**Syntax**

```java
com.APNS.DecodeError(data BYTE)
RETURNS (uuid STRING, error INTEGER)
```

1. *data* is the BYTE variable containing the error data. This BYTE variable must be located IN MEMORY.
2. *uuid* is a Base64 encoded string containing the push notification identifier.
3. *error* is the APNS error code returned by the server.

**Usage**

This method decodes the content of the BYTE variable passed as a parameter and received as response for a push notification message in the event of an error from the APNs server.

**Note:** This BYTE variable must be located IN MEMORY.

The *uuid* is a binary value that identifies the push notification message. It is returned as a Base64-encoded string.

The *error* returned value defines the APNs error code. For example, error will be set to 10 if the APNs server was shutdown. See the Apple® Push Notification Service error reference for more details.

In the case of a decoding error, the method will raise the exception -15566, with details in the SQLCA.SQLERRM register.

**Example**

```sql
DEFINE error_data BYTE,
    uuid STRING,
```
error INTEGER
LOCATE error_data IN MEMORY

-- Send push notification message TCP request
...
CALL req.doDataRequest(data)
LET resp = req.getResponse()
TRY
  CALL resp.getDataResponse(error_data)
  CALL com.APNS.DecodeError(error_data)
RETURNING uuid, ecode
...

For a complete example, see APNs push provider on page 2929.

com.APNS.DecodeFeedback
Decodes content of BYTE data returned from the APNS feedback service.

Syntax

com.APNS.DecodeFeedback(
  data BYTE,
  unregs RECORD
)

1.  data defines a BYTE variable containing the feedback data. This BYTE variable must be located IN MEMORY.
2.  unregs defines a structured dynamic array that contains the list of unregistered device tokens.
   a. timestamp is the number of seconds since UNIX® Epoch (in UTC)
   b.  deviceToken is an APNS device token that has been unregistered (encoded in Base-64)

Usage

Apple® recommends connecting frequently to the APNS feedback server in order to verify that your applications are still registered for push notifications.

To get APNS feedback, you must perform a TCP request (using SSL/TLS), to the following specific URI:

tcps://feedback.push.apple.com:2196

The DecodeFeedback() method decodes the content of the BYTE variable, which was passed as a parameter and received as response for the TCP request to the APNS feedback server.

Note: This BYTE variable must be located IN MEMORY.

For the second parameter, this method takes a structured dynamic array that will be filled with the list of unregistered APNS device tokens. It is up to the push program to stop sending push notification messages for these unregistered device tokens.

The timestamp member of an unregs dynamic array element can be used to verify that device tokens have not been re-registered since the feedback entry was generated. This timestamp is returned as a number of seconds since the UNIX® epoch, in UTC. Use the util.Datetime.fromSecondsSinceEpoch on page 2730 utility API to convert timestamp to a DATETIME value in the current local time.

The deviceToken member of an unregs dynamic array element identifies iOS devices that have been unregistered from the APNS server. Note that these identifier is encoded in Base64.

In the event of a decoding error, the method will raise the exception -15566, with details in the SQLCA.SQLERRM register.
Example

DEFINE feedback_data BYTE,
   unregs DYNAMIC ARRAY OF RECORD
      timestamp  INTEGER,
      deviceToken STRING
   END RECORD,
   i INTEGER

LOCATE feedback_data IN MEMORY

... TCP request to APNS feedback server ...

CALL com.APNS.DecodeFeedback(feedback_data, unregs)

FOR i=1 TO unregs.getLength()
   DISPLAY i, " ", unrefs[i].deviceToken
END FOR

For a complete example, see APNs feedback handler on page 2930.

com.APNS.EncodeMessage

Encodes an APNS specific push notification message into a BYTE.

Syntax

com.APNS.EncodeMessage (      
data BYTE, 
   deviceIdBase64 STRING, 
   json STRING, 
   uuidBase64 STRING, 
   expiration INTEGER, 
   priority INTEGER)       
RETURNS INTEGER

1. data defines a BYTE variable holding the APNs message. This BYTE variable must be located IN MEMORY.
2. deviceIdBase64 defines an APNs device token (encoded in Base-64).
3. json defines a JSON string containing the APNs push message data.
4. uuidBase64 defines the 4 bytes-long push message identifier (encoded in Base64).
5. expiration defines a number of seconds since UNIX® Epoch defining the expiration date of the message.
6. priority defines an integer as the priority of the message.

Usage

This method builds the APNS push notification message into a BYTE variable, for a given device token.

Note: This BYTE variable must be located IN MEMORY.

Note: The size of an APNs notification payload cannot exceed 2 Kilobytes. Make sure that the resulting BYTE variable does not exceed this size limitation. If more information needs to be passed, after receiving the push message, apps must contact the server part to query for more information. However, this is only possible when network is available.

The APNS push notification message protocol requires some binary data to be encoded in the message content before it is sent to the APNS server with a TCP (over SSL/TLS) request, to specific URIs, namely:

• "tcps://gateway.sandbox.apple.com:2195" (for development)
• "tcps://gateway.push.apple.com:2195" (for production)

You need to provide several parameters in order to build the push notification message:
The `deviceToken` parameter is an APNS device token encoded in Base-64. It's used to identify the target device that must receive the push message. The device token identifies a single iOS device: If you have N devices registered to your push notification provider, you will have N different device tokens. If you want to send one push notification message to all the devices, you must send N different messages, where the only difference between the messages is the device token.

**Note:** It's in your hands to handle the list of registered device tokens. A device token is assigned to a physical iOS device when the mobile app issues a `registerForRemoteNotifications` call. The app must then provide its device token to the push provider program using a method such as a web service mechanism.

Fill the `json` parameter with a JSON string containing the APNS push message data. For example:

```plaintext
LET json = '{"aps":{"alert":"Hello, world","sound":"default","badge":1,"content-available":1}}'
```

See APNS documentation for more details about the JSON content of a message.

The `uuid` parameter is the 4 bytes-long push message identifier, encoded in Base64. This parameter can be used later to identify the message in push notification errors (`com.APNS.DecodeError` on page 2925). This parameter can be `NULL`. To create the `uuid` parameter, use the `security.RandomGenerator.CreateRandomString` API, with a size of 4:

```plaintext
LET uuid = security.RandomGenerator.createRandomString(4)
```

The `expiration` parameter is a number of seconds since UNIX® Epoch. It defines the expiration date of the message if it cannot be sent by the APNS server to the target devices. This parameter can be `NULL`, to indicate that there is no expiration date:

```plaintext
LET dt = CURRENT + INTERVAL (10) MINUTE TO MINUTE
LET expiration = util.Datetime.toSecondsSinceEpoch(dt)
```

The `priority` parameter can be used to define a priority for the push notification message. Typically, use a value of 10 for immediate, 5 for delayed. This parameter can be `NULL`. See APNS documentation for more details.

If there's an encoding error, the method will raise the exception `-15566`, with details in the `SQLCA.SQLERRM` register.

**Example**

```plaintext
DEFINE push_data BYTE,
    deviceTokenHexa STRING,
    dt DATETIME YEAR TO FRACTION(3),
    expiration INTEGER,
    json_data STRING,
    uuid STRING

LOCATE push_data IN MEMORY

LET deviceTokenHexa = "84e3................."

LET dt = CURRENT + INTERVAL (10) MINUTE TO MINUTE
LET expiration = util.Datetime.toSecondsSinceEpoch(dt)

LET json_date = util.JSON.stringify(...)
LET uuid = security.RandomGenerator.createRandomString(4)

CALL com.APNS.EncodeMessage(
    push_data,
    security.HexBinary.ToBase64(deviceTokenHexa),
    json_data,
    uuid,
```
IF LENGTH(push_data) > 2000 THEN
   -- Must reduce the message content...
   RETURN FALSE
END IF

-- Do the TCP request with the push_data variable

For a complete example, see APNs push provider on page 2929.

**APNs examples**

**APNs push provider**

The `com.APNS` class can be used to implement an APNs push provider.

The following code example implements a push program using the `com.APNS` API to send a notification message to devices by using the TCP request API. See also `com.APNS methods` on page 2925 for more details about the APNs API.

**Note:** An SSL/TLS certificate needs to be defined in FGLPROFILE, as described in APNs SSL/TLS certificate on page 2923.

**Note:** The size of the resulting BYTE variable containing the APNS payload cannot exceed 2 Kilobytes.

**Important:** In order to check that the push message was properly handled by the APNs server, you need to define a TCP request timeout (2 seconds in this example). In case of error, the APNs server will return a response immediately. In case of success, there is no response from the APNs server. For more details about this protocol, see Apple's APNs documentation.

```
IMPORT com
IMPORT security
IMPORT util

MAIN
   DEFINE json STRING
   DEFINE deviceTokenHexa STRING
   DEFINE req com.TCPRequest
   DEFINE resp com.TCPResponse
   DEFINE uuid STRING
   DEFINE ecode INTEGER
   DEFINE dt DATETIME YEAR TO SECOND
   DEFINE exp INTEGER
   DEFINE data, err BYTE
   LOCATE data IN MEMORY
   LOCATE err IN MEMORY

   LET deviceTokenHexa = "84e3................."
   LET dt = CURRENT + INTERVAL(10) MINUTE TO MINUTE
   LET exp = util.Datetime.toSecondsSinceEpoch(dt)
   TRY
      LET req = com.TCPRequest.create( "tcps://gateway.push.apple.com:2195" )
      CALL req.setKeepConnection(true)
      CALL req.setTimeout(2) # Wait 2 seconds for APNs to return an error code
      LET uuid = security.RandomGenerator.createRandomString(4)
      LET json = '{"aps":{"alert":"Hello, world","sound":"default","badge":1,"content-available":1}}'
      CALL com.APNS.EncodeMessage( data, security.HexBinary.ToBase64(deviceTokenHexa),
```
IF LENGTH(data) > 2000 THEN
    DISPLAY "APNS payload cannot exceed 2 kilobytes"
    EXIT PROGRAM 1
END IF
DISPLAY "Sending notif with ID:" , uuid , " and expiring at ", dt
CALL req.doDataRequest(data)
LET resp = req.getResponse()
TRY
    CALL resp.getDataResponse(err)
    CALL com.APNS.DecodeError(err) RETURNING uuid, ecode
    DISPLAY "ERROR code ":",ecode
    DISPLAY "ERROR uuid ":",uuid
CATCH WHEN -15553 DISPLAY "Timeout Push sent without error"
     WHEN -15566 DISPLAY "Operation failed ":", SQLCA.SQLERRM
     WHEN -15564 DISPLAY "Server has shutdown"
     OTHERWISE DISPLAY "ERROR ":",STATUS
END CASE
END TRY
CATCH
    DISPLAY "ERROR ":",STATUS\ " (\ "SQLCA.SQLERRM\ |\ ")
END TRY
END MAIN

Related concepts

Apple Push Notification Service (APNs) on page 3625
Follow this procedure to implement push notification with APNs.

APNs feedback handler

The com.APNS class can be used to implement a server application to query the APNs feedback service.

Implement an APNs feedback handler to get a list of unregistered device tokens in order to stop sending push notification messages to these apps.

Note: An SSL/TLS certificate needs to be defined in FGLPROFILE, as described in APNs SSL/TLS certificate on page 2923.
LET req = com.TCPRequest.create( "tcps://feedback.push.apple.com:2196" )
CALL req.setKeepConnection(true)
CALL req setTimeout(2)
CALL req.doRequest()
LET resp = req.getResponse()
CALL resp.getDataResponse(data)
DISPLAY "Feedback service has responded"
CALL com.APNS.DecodeFeedback(data, feedback)
FOR i=1 TO feedback.getLength()
    LET timestamp =
        util.Datetime.fromSecondsSinceEpoch(feedback[i].timestamp)
        DISPLAY "Device Token :", feedback[i].deviceToken, " Timestamp :",
        timestamp
END FOR
CATCH
    CASE STATUS
        WHEN -15553 DISPLAY "Timeout: No feedback message"
        WHEN -15566 DISPLAY "Operation failed :", SQLCA.SQLERRM
        WHEN -15564 DISPLAY "Server has shutdown"
        OTHERWISE DISPLAY "ERROR :", STATUS
    END CASE
END TRY
END MAIN

Related concepts
Implementing a token maintainer on page 3631
The token maintainer is a BDL Web Services server program that handles push token registration from mobile apps.

The xml package
The Genero Web Services XML package provides classes and methods to handle any kind of XML documents, including documents with namespaces.

The library provides a W3C-compatible DOM API, integrating additional XML Schema and DTD validation methods. There is also an API compatible with StAX for writing or reading XML documents where performance and speed are important.

Use the IMPORT statement at the top of the module using this library:

IMPORT xml

Note: The DOM API of the om package is designed to handle specific FGL files and to manipulate the user interface tree (the AUI tree). For all other cases, we recommend that you use the DOM API of the Web Services xml package.

The Document Object Modeling (DOM) classes
The Document Object Modeling (DOM) classes manage XML documents entirely in memory with support for XML Schema and DTD validation.

The DomDocument class
The xml.DomDocument class provides methods to manipulate a data tree, following the DOM standards.

The STATUS variable is set to zero after a successful method call.
**xml.DomDocument methods**
Methods for the xml.DomDocument class.

**Table 633: Class methods: Creation**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>getDocumentElement()</td>
<td>Returns the root XML Element DomNode object for this DomDocument object.</td>
</tr>
<tr>
<td>RETURNS xml.DomNode</td>
<td></td>
</tr>
<tr>
<td>getDocumentNodesCount()</td>
<td>Returns the number of child DomNode objects for a DomDocument object.</td>
</tr>
<tr>
<td>RETURNS INTEGER</td>
<td></td>
</tr>
<tr>
<td>getDocumentNodeItem(index INTEGER)</td>
<td>Returns the child DomNode object at a given position for this DomDocument object.</td>
</tr>
<tr>
<td>RETURNS xml.DomNode</td>
<td></td>
</tr>
<tr>
<td>getElementById(id STRING)</td>
<td>Returns the element that has an attribute of type ID with the given value.</td>
</tr>
<tr>
<td>RETURNS xml.DomNode</td>
<td></td>
</tr>
<tr>
<td>getElementsByTagName(tag STRING)</td>
<td>Returns a DomNodeList object containing all XML Element DomNode objects with the same tag name in the entire document.</td>
</tr>
<tr>
<td>RETURNS xml.DomNodeList</td>
<td></td>
</tr>
<tr>
<td>getElementsByTagNameNS(tag STRING, ns STRING)</td>
<td>Returns a DomNodeList object containing all namespace qualified XML Element DomNode objects with the same tag name and namespace in the entire document</td>
</tr>
<tr>
<td>RETURNS xml.DomNodeList</td>
<td></td>
</tr>
<tr>
<td>getFirstDocumentNode()</td>
<td>Returns the first child DomNode object for a DomDocument object.</td>
</tr>
<tr>
<td>RETURNS xml.DomNode</td>
<td></td>
</tr>
<tr>
<td>getLastDocumentNode()</td>
<td>Returns the last child DomNode object in a DomDocument object.</td>
</tr>
<tr>
<td>RETURNS xml.DomNode</td>
<td></td>
</tr>
<tr>
<td>selectByXPath(expr STRING, args ...)</td>
<td>Returns a DomNodeList object containing all DomNode objects matching a XPath 1.0 expression.</td>
</tr>
<tr>
<td>RETURNS xml.DomNodeList</td>
<td></td>
</tr>
</tbody>
</table>
### Table 635: Object methods: Management

| Name                                      | Description                                           |
|--------------------------------==========|-------------------------------------------------------|
| `clone()`                                | Returns a copy of a DomDocument object.               |
| `declareNamespace(node xml.DomNode, alias STRING, ns STRING)` | Forces namespace declaration to a XML Element DomNode for a DomDocument object. |
| `importNode(n xml.DomNode, deep INTEGER)` | Imports a DomNode from a DomDocument object into its new context (attached to a DomDocument object). |
**Table 636: Object methods: Node Creation**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>createAttribute (elt STRING )</code></td>
<td>RETURNS <code>xml.DomNode</code></td>
</tr>
<tr>
<td><code>createAttributeNS (prefix STRING, elt STRING, ns STRING )</code></td>
<td>RETURNS <code>xml.DomNode</code></td>
</tr>
<tr>
<td><code>createCDATASection (data STRING )</code></td>
<td>RETURNS <code>xml.DomNode</code></td>
</tr>
<tr>
<td><code>createComment (cmt STRING )</code></td>
<td>RETURNS <code>xml.DomNode</code></td>
</tr>
<tr>
<td><code>createDocumentType (name STRING, publicID STRING, systemID STRING, internalSubset STRING )</code></td>
<td>RETURNS <code>xml.DomNode</code></td>
</tr>
<tr>
<td><code>createElement (elt STRING )</code></td>
<td>RETURNS <code>xml.DomNode</code></td>
</tr>
<tr>
<td><code>createElementNS (prefix STRING, elt STRING, ns STRING )</code></td>
<td>RETURNS <code>xml.DomNode</code></td>
</tr>
<tr>
<td><code>createEntityReference (entity STRING )</code></td>
<td>RETURNS <code>xml.DomNode</code></td>
</tr>
<tr>
<td><code>createNode (str STRING )</code></td>
<td>RETURNS <code>xml.DomNode</code></td>
</tr>
<tr>
<td><code>createProcessingInstruction (target STRING, data STRING )</code></td>
<td>RETURNS <code>xml.DomNode</code></td>
</tr>
<tr>
<td><code>createTextNode (txt STRING )</code></td>
<td>RETURNS <code>xml.DomNode</code></td>
</tr>
</tbody>
</table>
### Table 637: Object methods: Load and Save

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>load(filename STRING)</code></td>
<td>Loads a XML Document into a DomDocument object from a file or an URL.</td>
</tr>
<tr>
<td><code>loadFromPipe(name STRING)</code></td>
<td>Loads a XML Document into a DomDocument object from a PIPE.</td>
</tr>
<tr>
<td><code>loadFromString(xmlstr STRING)</code></td>
<td>Loads a XML Document into a DomDocument object from a string.</td>
</tr>
<tr>
<td><code>normalize()</code></td>
<td>Normalizes the entire Document.</td>
</tr>
<tr>
<td><code>save(filename STRING)</code></td>
<td>Saves a DomDocument object as a XML Document to a file or URL.</td>
</tr>
<tr>
<td><code>saveToPipe(name STRING)</code></td>
<td>Saves a DomDocument object as an XML Document to a PIPE.</td>
</tr>
<tr>
<td><code>saveToString()</code></td>
<td>Saves a DomDocument object as a XML Document to a string.</td>
</tr>
<tr>
<td><code>RETURNS STRING</code></td>
<td></td>
</tr>
</tbody>
</table>
### Table 638: Object methods: Configuration

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getFeature (property STRING)</code></td>
<td>Returns the document encoding as defined in the XML document declaration.</td>
</tr>
<tr>
<td><code>getXmlEncoding ()</code></td>
<td>Returns the document encoding as defined in the XML document declaration.</td>
</tr>
<tr>
<td><code>getXmlVersion ()</code></td>
<td>Returns the document version as defined in the XML document declaration.</td>
</tr>
<tr>
<td><code>isXmlStandalone ()</code></td>
<td>Checks whether the XML standalone attribute is set in the XML declaration.</td>
</tr>
<tr>
<td><code>setFeature (property STRING, value STRING)</code></td>
<td>Sets a feature for a DomDocument object.</td>
</tr>
<tr>
<td><code>setXmlEncoding (encoding STRING)</code></td>
<td>Sets the XML document encoding in the XML declaration.</td>
</tr>
<tr>
<td><code>setXmlStandalone (alone INTEGER)</code></td>
<td>Sets the XML standalone attribute in the XML declaration to &quot;yes&quot; or &quot;no&quot; in the XML declaration, or removes the standalone attribute.</td>
</tr>
</tbody>
</table>

### Table 639: Object methods: Validation

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>validate ()</code></td>
<td>Performs a DTD or XML Schema validation for a DomDocument object.</td>
</tr>
<tr>
<td><code>validateOneElement (elt xml.DonNode)</code></td>
<td>Performs a DTD or XML Schema validation of a XML Element DomNode object.</td>
</tr>
</tbody>
</table>
Table 640: Object methods: Error Management

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getErrorsCount ()</td>
<td>Returns the number of errors encountered during the loading, saving or validation of a XML document.</td>
</tr>
<tr>
<td>getErrorDescription (</td>
<td>Returns the error description at the given position.</td>
</tr>
<tr>
<td>index INTEGER )</td>
<td></td>
</tr>
<tr>
<td>Syntax</td>
<td></td>
</tr>
<tr>
<td>appendDocumentNode (</td>
<td></td>
</tr>
<tr>
<td>n xml.DomNode )</td>
<td></td>
</tr>
<tr>
<td>1. n defines the node to add.</td>
<td></td>
</tr>
<tr>
<td>Usage</td>
<td></td>
</tr>
<tr>
<td>Adds a child xml.DomNode object to the end of the DomNode children for this DomDocument object, where n is the node to add.</td>
<td></td>
</tr>
<tr>
<td>Only Text nodes, Processing Instruction nodes, Document Fragment nodes, one Element node, and one Document Type node allowed.</td>
<td></td>
</tr>
<tr>
<td>Note: A fragment is a structure created to receive XML nodes that are not always valid. Once a fragment is added to a valid node, the fragment becomes empty as all nodes are moved from the fragment as a child to a valid node. So developers can work on the fragment until it is added to another node. At that time developers should no longer work on the fragment but rather on the valid node.</td>
<td></td>
</tr>
<tr>
<td>In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.</td>
<td></td>
</tr>
<tr>
<td>Syntax</td>
<td></td>
</tr>
<tr>
<td>clone ()</td>
<td></td>
</tr>
<tr>
<td>returns xml.DomDocument</td>
<td></td>
</tr>
<tr>
<td>Usage</td>
<td></td>
</tr>
<tr>
<td>Returns a copy of this DomDocument object, or NULL.</td>
<td></td>
</tr>
<tr>
<td>In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.</td>
<td></td>
</tr>
<tr>
<td>xml.DomDocument.create</td>
<td></td>
</tr>
</tbody>
</table>
Constructor of an empty DomDocument object.

Syntax

```java
xml.DomDocument.create()
RETURNS xml.DomDocument
```

Usage

Constructor of an empty DomDocument object.

Returns a DomDocument object.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

Example

Create a DomDocument without a root node:

```java
xml.domDocument.create()
```

Create a DomDocument with an initial root node named ARoot:

```java
xml.domDocument.create("ARoot")
```

xml.DomDocument.createAttribute

Creates a XML Attribute DomNode object for a DomDocument object.

Syntax

```java
createAttribute( 
   elt STRING ) 
RETURNS xml.DomNode
```

1. `elt` defines the name of the XML attribute.

Usage

Creates a XML Attribute DomNode object for a DomDocument object, where `elt` is the name of the XML attribute. It cannot be NULL.

Returns the XML element DomNode object, or NULL.

To create a default namespace declaration attribute use xmlns as the name. (Using declareNamespace instead is recommended)

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomDocument.createAttributeNS

Creates a XML namespace-qualified Attribute DomNode object for a DomDocument object.

Syntax

```java
createAttributeNS( 
   prefix STRING, 
   elt STRING, 
   ns STRING )
```
RETURNS xml.DomNode

1. *prefix* defines the prefix of the XML attribute.
2. *elt* defines the name of the XML attribute.
3. *ns* defines the namespace URI of the XML attribute.

**Usage**

Creates a XML namespace-qualified Attribute DomNode object for this DomDocument object where *prefix* is the prefix of the XML attribute, it cannot be NULL; *elt* is the name of the XML attribute; it cannot be NULL; *ns* is the namespace URI of the XML attribute, it cannot be NULL.

Returns the XML element DomNode object, or NULL.

To create a namespace declaration attribute use *xmlns* as the prefix and http://www.w3.org/XML/1998/namespace as the namespace. Using declareNamespace instead is recommended.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomDocument.createCDATASection

Creates an XML CDATA DomNode object for a DomDocument object.

**Syntax**

```
cREATECDATASECTION(  
data STRING  )  
RETURNS xml.DomNode
```

1. *data* defines the data of the XML CDATA node.

**Usage**

Creates a XML CDATA DomNode object for this DomDocument object, where *data* is the data of the XML CDATA node, or NULL.

Returns the XML element DomNode object, or NULL.

Only the characters #x9, #xA, #xD, [#x20-#xD7FF], [#xE000-#xFFFD] and [#x10000-#x10FFFF] are allowed in the content of an XML CDATASection node.

The character sequence (Double-Hyphen) ‘--’ is not allowed in the content of a XML CDATASection node. The save and normalize methods will fail if the sequence of characters other than those allowed exist in a CDATASection node.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomDocument.createComment

Creates a XML Comment DomNode object for a DomDocument object.

**Syntax**

```
cREATECOMMENT(  
cmt STRING  )  
RETURNS xml.DomNode
```

1. *cmt* defines the data of the XML Comment node.
Usage

Creates a XML Comment DomNode object for this DomDocument object, where \textit{cmt} is the data of the XML Comment node, or NULL.

Returns the XML element DomNode object, or NULL.

Only the characters 
$\#x9$, \$\#xA$, \$\#xD$, \$[\#x20-\#xD7FF]$, \$[\#xE000-\#xFFFD]$ and \$[\#x10000-\#x10FFFF]$ are allowed in the content of an XML Comment node.

The character sequence (Double-Hyphen) ‘--’ is not allowed in the content of a XML CDATASection node.

The \texttt{save} and \texttt{normalize} methods will fail if the sequence of characters other than those allowed exist in a CDATASection node.

In case of error, the method throws an exception and sets the \texttt{STATUS} variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

\texttt{xml.DomDocument.createDocument}

Constructor of a DomDocument with a XML root element.

Syntax

\begin{verbatim}
xml.DomDocument.createDocument(
   name STRING
) RETURNS xml.DomDocument
\end{verbatim}

1. \texttt{name} defines the XML element.

Usage

Constructor of a \texttt{xml.DomDocument} with a XML root element; where \texttt{name} is the name of the XML Element.

Returns a \texttt{xml.DomDocument} object or NULL.

In case of error, the method throws an exception and sets the \texttt{STATUS} variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

\texttt{xml.DomDocument.createDocumentFragment}

Creates a XML Document Fragment DomNode object for a DomDocument object.

Syntax

\begin{verbatim}
createDocumentFragment ()
RETURNS xmlDOMNode
\end{verbatim}

Usage

Creates a XML Document Fragment DomNode object for this DomDocument object.

Returns the XML element DomNode object, or NULL.

In case of error, the method throws an exception and sets the \texttt{STATUS} variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

\texttt{xml.DomDocument.createDocumentNS}
Constructor of a DomDocument with a root namespace-qualified XML root element

Syntax

```javascript
xml.DomDocument.createDocumentNS(
    prefix STRING,
    name STRING,
    ns STRING )
RETURNS xml.DomDocument
```

1. `prefix` defines the prefix of the XML Element or NULL.
2. `name` defines the XML element.
3. `ns` is the namespace of the XML Element.

Usage

Constructor of a `xml.DomDocument` with a root namespace-qualified XML root element where `prefix` is the prefix of the XML element or NULL, `name` is the name of the XML element, and `ns` is the namespace of the XML element. Returns a `xml.DomDocument` object.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (`STATUS`) on page 3554.

Example

Create a DomDocument with an initial root node named "List" with `abc` as the prefix and `http://www.mysite.com/xmlapi` as the namespace:

```javascript
xml.domdocument.createDocumentNS("abc","List","http://www.mysite.com/xmlapi")
```

Produces:

```xml
<abc:List xmlns:abc="http://www.mysite.com/xmlapi">
[...]
</abc:List>
```

`xml.DomDocument.createDocumentType`

Creates a XML Document Type (DTD) DomNode object for a DomDocument object.

Syntax

```javascript
createDocumentType( 
    name STRING,
    publicID STRING,
    systemID STRING,
    internalSubset STRING )
RETURNS xml.DomNode
```

1. `name` defines the name of the document type.
2. `publicID` defines the URI of the public identifier.
3. `systemID` defines the URL of the system identifier (Specifies the file location of the external DTD subset).
4. `internalSubset` defines the internal DTD subset.
Usage

Creates a XML Document Type (DTD) DomNode object for this DomDocument object; name is the name of the document type; publicID is the URI of the public identifier or NULL; systemID is the URL of the system identifier or NULL (Specifies the file location of the external DTD subset); internalSubset is the internal DTD subset or NULL.

Returns a xml.DomNode object, or NULL if internalSubset is malformed.

Important: This method is not part of W3C standard API.

Only internal DTDs are supported.

The public identifier cannot be set without the system identifier.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomDocument.createElement

Creates a XML Element DomNode object for a DomDocument object

Syntax

```javascript
createElement ( 
  elt STRING ) 
RETURNS xml.DomNode
```

1. elt defines the name of the XML element.

Usage

Creates a XML Element DomNode object for this DomDocument object, where elt is the name of the XML element, which cannot be NULL.

Returns the XML element DomNode object, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomDocument.createElementNS

Creates a XML namespace-qualified Element DomNode object for a DomDocument object.

Syntax

```javascript
createElementNS ( 
  prefix STRING, 
  elt STRING, 
  ns STRING ) 
RETURNS xml.DomNode
```

1. prefix defines the prefix of the XML element, or NULL to use the default namespace.
2. elt defines the name of the XML element.
3. ns defines the namespace URI of the XML element.

Usage

Creates a XML namespace-qualified Element DomNode object for this DomDocument object, where prefix is the prefix of the XML element, or NULL to use the default namespace. elt is the name of the XML element, this cannot be NULL. ns is the namespace URI of the XML element, this cannot be NULL.

Returns the XML element DomNode object, or NULL.
In case of error, the method throws an exception and sets the \texttt{STATUS} variable. Depending on the error, a human-readable description of the problem is available in the \texttt{SQLCA.SQLERRM} register. See \textit{Error handling in GWS calls (STATUS)} on page 3554.

\texttt{xml.DomDocument.createEntityReference}

Creates a XML EntityReference \texttt{DomNode} object for a \texttt{DomDocument} object

\textbf{Syntax}

\begin{verbatim}
createEntityReference(
    entity STRING )
RETURNS xml.DomNode
\end{verbatim}

1. \textit{entity} defines the name of the entity reference.

\textbf{Usage}

Creates a XML EntityReference \texttt{DomNode} object for this \texttt{DomDocument} object, where \textit{entity} is the name of the entity reference.

Returns the XML element \texttt{DomNode} object, or NULL.

An Entity Reference node is read-only and cannot be modified.

In case of error, the method throws an exception and sets the \texttt{STATUS} variable. Depending on the error, a human-readable description of the problem is available in the \texttt{SQLCA.SQLERRM} register. See \textit{Error handling in GWS calls (STATUS)} on page 3554.

\texttt{xml.DomDocument.createNode}

Creates a XML \texttt{DomNode} object from a string for a \texttt{DomDocument} object.

\textbf{Syntax}

\begin{verbatim}
createNode(
    str STRING )
RETURNS xml.DomNode
\end{verbatim}

1. \textit{str} defines the string representation of the \texttt{DomNode} to be created.

\textbf{Usage}

Creates a XML \texttt{DomNode} object from a string for this \texttt{DomDocument} object; \textit{str} is the string representation of the \texttt{DomNode} to be created.

Returns the XML element \texttt{DomNode} object, or NULL.

\textbf{Important}: This method is not part of W3C standard API.

In case of error, the method throws an exception and sets the \texttt{STATUS} variable. Depending on the error, a human-readable description of the problem is available in the \texttt{SQLCA.SQLERRM} register. See \textit{Error handling in GWS calls (STATUS)} on page 3554.

\texttt{xml.DomDocument.createProcessingInstruction}

Creates a XML Processing Instruction \texttt{DomNode} object for this \texttt{DomDocument} object.

\textbf{Syntax}

\begin{verbatim}
createProcessingInstruction(
    target STRING, 
    data STRING )
RETURNS xml.DomNode
\end{verbatim}
1. **target** defines the target part of the XML Processing Instruction.
2. **data** defines the data part of the XML Processing Instruction.

**Usage**

Creates a XML Processing Instruction DomNode object for this DomDocument object, where **target** is the target part of the XML Processing Instruction, cannot be NULL; **data** is the data part of the XML Processing Instruction, or NULL.

Returns the XML element DomNode object, or NULL.

Only the characters #x9, #xA, #xD, [#x20-#xD7FF], [#xE000-#xFFFFD] and [#x10000-#x10FFFF] are allowed in the content of an XML Processing Instruction node.

The character sequence (Double-Hyphen) ‘--’ is not allowed in the content of an XML Processing Instruction. The **save()** and **normalize()** methods will fail if this sequence or characters other than those allowed exist in a Processing Instruction node.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**xml.DomDocument.createTextNode**

Creates a XML Text DomNode object for a DomDocument object.

**Syntax**

```sql
createTextNode( 
    txt STRING 
) RETURNS xml.DomNode
```

1. **txt** defines the data of the XML Text node.

**Usage**

Creates a XML Text DomNode object for this DomDocument object, where **txt** is the data of the XML Text node, or NULL.

Returns the XML element DomNode object, or NULL.

Only the characters #x9, #xA, #xD, [#x20-#xD7FF], [#xE000-#xFFFFD] and [#x10000-#x10FFFF] are allowed in the content of an XML Text node. The **save()** and **normalize()** methods will fail if characters other than those allowed exist in a Text node.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**xml.DomDocument.declareNamespace**

Forces namespace declaration to a XML Element DomNode for a DomDocument object.

**Syntax**

```sql
declareNamespace( 
    node xml.DomNode, 
    alias STRING, 
    ns STRING )
```

1. **node** defines the XML Element DomNode that carries the namespace definition.
2. **alias** defines the alias of the namespace to declare.
3. **ns** defines the URI of the namespace to declare.
Usage
Forces namespace declaration to a XML Element **DomNode** for this DomDocument object; *node* is the XML Element DomNode that carries the namespace definition; *alias* is the alias of the namespace to declare, or NULL to declare the default namespace; *ns* is the URI of the namespace to declare (can only be NULL if alias is NULL).

**Important:** This method is not part of W3C standard API.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomDocument.getDocumentElement
Returns the root XML Element DomNode object for this DomDocument object.

**Syntax**

```java
getElement() 
RETURNS xml.DomNode
```

Usage
Returns the root XML Element DomNode object for this DomDocument object.

Returns the XML element DomNode object, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomDocument.getDocumentNodesCount
Returns the number of child DomNode objects for a DomDocument object.

**Syntax**

```java
getDocumentNodesCount() 
RETURNS INTEGER
```

Usage
Returns the number of child DomNode objects in this DomDocument object.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomDocument.getDocumentNodeItem
Returns the child DomNode object at a given position for this DomDocument object.

**Syntax**

```java
getNodeItem(index INTEGER) 
RETURNS xml.DomNode
```

1. *index* defines the position of the node to return (index starts at 1).
Usage

Returns the child xmlDOMNode object at a given position for this DomDocument object where index is the position of the node to return (index starts at 1), or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomDocument.getElementById

Returns the element that has an attribute of type ID with the given value.

Syntax

getElementById (id STRING)
RETURNS xmlDOMNode

1. id defines the Id value.

Usage

Use this method to return the xmlDOMNode element that has an attribute of type ID with the given value, or NULL if there is none.

Attributes with the name "ID" or "id" are not of type ID unless so defined with setIdAttribute or setIdAttributeNS. However, there is a specific attribute called xml:id belonging to the namespace http://www.w3.org/XML/1998/namespace that is always of type ID even if not set with setIdAttributeNS.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomDocument.getElementsByTagName

Returns a DomNodeList object containing all XML Element DomNode objects with the same tag name in the entire document.

Syntax

getElementsByTagName (tag STRING)
RETURNS xmlDOMNodeList

1. tag defines the name of the XML Element tag to match or "*" to match all tags.

Usage

Use this method to return a DomNodeList object containing all XML Element DomNode objects with the same tag name in the entire document. The tag string contains the name of the XML Element tag to match, or use "*" to match all tags.

Returns a DomNodeList object, or NULL.

The returned list is ordered using a Depth-First pass algorithm.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomDocument.getElementsByTagNameNS
Returns a DomNodeList object containing all namespace qualified XML Element DomNode objects with the same tag name and namespace in the entire document.

**Syntax**

```vbnet
getElementsByTagNameNS (tag STRING, ns STRING )
RETURNS xml.DomNodeList
```

1. *tag* defines the name of the XML Element tag to match or "*" to match all tags.
2. *ns* defines the namespace URI of the XML Element tag to match, or "*" to match all namespaces.

**Usage**

Use this method to return a `xml.DomNodeList` object containing all namespace qualified XML Element DomNode objects with the same tag name and namespace in the entire document. The *tag* string contains the name of the XML Element tag to match, or use "*" to match all tags. *ns* is the namespace URI of the XML Element tag to match, or use "*" to match all namespaces.

Returns a `DomNodeList` object, or `NULL`. The returned list is ordered using a Depth-First pass algorithm.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**Example error management**

```vbnet
FOR i=1 TO doc.getErrorsCount()  
  DISPLAY "[", i, "] ", doc.getErrorDescription(i)  
END FOR
```

Displays all the errors encountered in the save, load, or validate of the `doc` `DomDocument`.

To display other errors, use the global variable `STATUS` to get the error code and `err_get(status)` or `SQLCA.SQLERRM` to get the description of the error. See error code for more details.

**Syntax**

```vbnet
getErrorDescription (index INTEGER )
RETURNS STRING
```

1. *index* defines the position of the error description (index starts at 1).

**Usage**

This method returns the error description at the given position. *index* is the position of the error description (index starts at 1). It returns a string with an error description.

**Important:** This method is not part of W3C standard API.

**Example error management**

```vbnet
xml.DomDocument.getErrorDescription
```

Returns the error description at the given position.

**Syntax**

```vbnet
xml.DomDocument.getErrorDescription
```

Returns the error description at the given position.
Returns the number of errors encountered during the loading, saving or validation of a XML document.

**Syntax**

```java
getErrorsCount ()
RETURNS INTEGER
```

**Usage**

This method returns the number of errors encountered during the loading, saving, or the validation of a XML document.

Returns the number of errors, or zero if there are none.

**Important:** This method is not part of W3C standard API.

**Example error management**

```java
FOR i=1 TO doc.getErrorsCount()
    DISPLAY "[", i, "] ", doc.getErrorDescription(i)
END FOR
```

Displays all the errors encountered in the save, load, or validate of the `doc` DomDocument.

To display other errors, use the global variable STATUS to get the error code and `err_get (status)` or `SQLCA.SQLERRM` to get the description of the error. See error code for more details.

**xml.DomDocument.getFirstDocumentNode**

Returns the first child DomNode object for a DomDocument object.

**Syntax**

```java
getFirstDocumentNode ()
RETURNS xml.DomNode
```

**Usage**

Use this method to return the first child `xml.DomNode` object for this `DomDocument` object, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**xml.DomDocument.getFeature**

Gets a feature for a DomDocument object.

**Syntax**

```java
getFeature (property STRING)
RETURNS STRING
```

1. `property` is the name of the input parameter defining the name of a DomDocument feature.

**Usage**

This method returns the name of a feature for the `DomDocument` object, where `property` is the name of the `DomDocument` feature.

Returns the value of the feature.
In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomDocument.getLastDocumentNode

    Returns the last child DomNode object in a DomDocument object.

**Syntax**

```
getLastDocumentNode ()
RETURNS xml.DomNode
```

**Usage**

Use this method to return the last child xml.DomNode object in this DomDocument object, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomDocument.getXmlEncoding

    Returns the document encoding as defined in the XML document declaration.

**Syntax**

```
getXmlEncoding ()
RETURNS STRING
```

**Usage**

This method returns the document encoding as defined in the XML document declaration, or NULL if there is none.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomDocument.getXmlVersion

    Returns the document version as defined in the XML document declaration.

**Syntax**

```
getXmlVersion ()
RETURNS STRING
```

**Usage**

This method returns the document version as defined in the XML document declaration, which is 1.0. No other versions are supported.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomDocument.importNode

    Imports a DomNode from a DomDocument object into its new context (attached to a DomDocument object).

**Syntax**

```
importNode (
```

```
n xml.DomNode
  deep INTEGER )
RETURNS xml.DomNode

1. n defines the node to import.
2. deep defines a boolean identifying whether to import the node only or the node and all its child nodes.

Usage
Imports a xml.DomNode object from a DomDocument object into its new context (attached to this DomDocument object), where n is the node to import. When deep is FALSE only the node is imported; when TRUE the node and all its child nodes are imported.

Returns the xml.DomNode object that has been imported to this DomDocument, or NULL.

Document and Document Type nodes cannot be imported.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

Inserts a child DomNode object before another child DomNode for this DomDocument object.

Syntax
insertBeforeDocumentNode (  
  newNode xml.DomNode,  
  ref xml.DomNode )

1. newNode defines the node to insert.
2. ref defines the reference node (the node before which the new node must be inserted).

Usage
Inserts a child xml.DomNode object before another child DomNode for this DomDocument object; newNode is the node to insert, ref is the reference node (the node before which the new node must be inserted).

Only Text nodes, Processing Instruction nodes, Document Fragment nodes, one Element node, and one Document Type node allowed.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomDocument.insertAfterDocumentNode
Inserts a child DomNode object after another child DomNode in a DomDocument object.

Syntax
insertAfterDocumentNode (  
  newNode xml.DomNode,  
  ref xml.DomNode )

1. newNode defines the node to insert.
2. ref defines the reference node (the node after which the new node must be inserted).
**Usage**

Inserts a child `xml.DomNode` object after another child `DomNode` in this `DomDocument` object; `newNode` is the node to insert; `ref` is the reference node (the node after which the new node must be inserted).

**Important:** This method is not part of W3C standard API.

Only Text nodes, Processing Instruction nodes, Document Fragment nodes, one Element node, and one Document Type node allowed.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**xml.DomDocument.isXmlStandalone**

Checks whether the XML standalone attribute is set in the XML declaration.

**Syntax**

```
isXmlStandalone()  RETURNS INTEGER
```

**Usage**

Use this method to check if the XML standalone attribute is set in the XML declaration or not. It returns `TRUE` if set to yes, `FALSE` if not.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**xml.DomDocument.load**

Loads a XML Document into a `DomDocument` object from a file or an URL.

**Syntax**

```
load(filename STRING )
```

1. `filename` defines a valid URL or the name of the file.

**Usage**

This method loads a XML Document into a `DomDocument` object from a file or an URL, where `filename` is a valid URL or the name of the file.


Use `setFeature()` to specify how the document can be loaded. HTML parsing is possible when `enable-html-compliancy` is enabled.

See `getErrorsCount()` and `getErrorDescription()` to retrieve error messages related to XML document.

**Note:** When loading a document, if `xml.DomNode` objects are still referenced in other variables of the program, the entire document is kept in memory. Otherwise, the DOM nodes of the document are deleted before loading the new document. For more details about object references and garbage collection in BDL, see Working with objects on page 571.
In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**Important:** On Mac® OS X versions prior to 10.9, the libxml library (used by Genero Web Services classes) has a bug when parsing HTML documents. If you set the HTML compliancy option with CALL doc.setFeature("enable-html-compliancy",TRUE), loading an HTML document with xml.DomDocument.load() may produce additional blank TEXT nodes, because the libxml library loads some ignorable white-space nodes from the HTML document. Starting with Mac® OS X 10.9, the libxml library of the system has fixed this bug.

xml.DomDocument.loadFromPipe
Loads a XML Document into a DomDocument object from a PIPE.

**Syntax**

```
loadFromPipe(
    name STRING )
```

1. `name` defines the command to read from the PIPE.

**Usage**

Use this method to load a XML Document into a DomDocument object from a PIPE, where `name` is the command to read from the PIPE.

**Important:** This method is not part of W3C standard API.

Use `setFeature()` to specify how the document can be loaded. HTML parsing is possible when `enable-html-compliancy` is enabled.

See `getErrorsCount()` and `getErrorDescription()` to retrieve error messages related to XML document.

**Note:** When loading a document, if xml.DomNode objects are still referenced in other variables of the program, the entire document is kept in memory. Otherwise, the DOM nodes of the document are deleted before loading the new document. For more details about object references and garbage collection in BDL, see Working with objects on page 571.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomDocument.loadFromstring
Loads a XML Document into a DomDocument object from a string.

**Syntax**

```
loadFromString(
    xmlstr STRING )
```

1. `xmlstr` defines the string to load.

**Usage**

Use this method to load a XML Document into a DomDocument object from a string, where `xmlstr` is the string to load.

**Important:** This method is not part of W3C standard API.

Use `setFeature()` to specify how the document can be loaded. HTML parsing is possible when `enable-html-compliancy` is enabled.
See `getErrorsCount()` and `getErrorDescription()` to retrieve error messages related to XML document.

**Note:** When loading a document, if `xml.DomNode` objects are still referenced in other variables of the program, the entire document is kept in memory. Otherwise, the DOM nodes of the document are deleted before loading the new document. For more details about object references and garbage collection in BDL, see Working with objects on page 571.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

### `xml.DomDocument.normalize`  
Normalizes the entire Document.

**Syntax**

```java
normalize()
```

**Usage**

Normalizes the entire Document. This method merges adjacent text nodes, removes empty text nodes, and sets namespace declarations as if the document had been saved.

See `getErrorsCount()` and `getErrorDescription()` to retrieve error messages related to XML document.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

### `xml.DomDocument.prependDocumentNode`  
Adds a child `xml.DomNode` object to the beginning of the `DomNode` children of a `DomDocument` object

**Syntax**

```java
prependDocumentNode (  
  n xml.DomNode  
)
```

1. `n` defines the node to add.

**Usage**

Adds a child `xml.DomNode` object to the beginning of the `DomNode` children in this `DomDocument` object; `n` is the node to add.

**Important:** This method is not part of W3C standard API.

See `getErrorsCount()` and `getErrorDescription()` to retrieve error messages related to XML document.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

### `xml.DomDocument.removeDocumentNode`  
Removes a child `DomNode` object from the `DomNode` children in a `DomDocument` object

**Syntax**

```java
removeDocumentNode (  
  n xml.DomNode  
)
```

1. `n` defines the node to remove.
**Usage**

Removes a child DomNode object from the DomNode children in this DomDocument object, where \( n \) is the node to remove.

Only Text nodes, Processing Instruction nodes, Document Fragment nodes, one Element node, and one Document Type node allowed.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

```java
xml.DomDocument.save()
```

Saves a DomDocument object as a XML Document to a file or URL.

**Syntax**

```
save(
    filename STRING )
```

1. `filename` defines a valid URL or the name of a file.

**Usage**

Saves a DomDocument object as a XML Document to a file or URL, where `filename` is a valid URL or the name of the file.

Only the following kinds of URLs are supported:

- `http://`
- `https://`
- `tcp://`
- `tcps://`
- `file:///
- `alias://`

See fglprofile Configuration for more details about URL mapping with aliases, and for proxy and security configuration.

See `setFeature()` to specify how the document can be saved.

See `getErrorsCount()` and `getErrorDescription()` to retrieve error messages related to XML document.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

```java
xml.DomDocument.saveToPipe()
```

Saves a DomDocument object as an XML Document to a PIPE.

**Syntax**

```
saveToPipe(
    name STRING )
```

1. `name` defines the command to start the pipe.

**Usage**

Saves a DomDocument object as a XML Document to a PIPE, where `name` is the command to start the pipe.

See `setFeature()` to specify how the document can be saved.
See `getErrorsCount()` and `getErrorDescription()` to retrieve error messages related to XML document.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (`STATUS`) on page 3554.

```java
xml.DomDocument.saveToString
Saves a DomDocument object as a XML Document to a string.

Syntax
```saveToString()
RETURNS STRING```

Usage
Saves a DomDocument object as a XML Document to a string. Returns the string that will contain the resulting document.

**Important:** This method is not part of W3C standard API.

See `setFeature()` to specify how the document can be saved.

See `getErrorsCount()` and `getErrorDescription()` to retrieve error messages related to XML document.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (`STATUS`) on page 3554.

```java
xml.DomDocument.selectByXPath
Returns a DomNodeList object containing all DomNode objects matching a XPath 1.0 expression.

Syntax
```selectByXPath(
   expr STRING,
   args ...)
RETURNS xml.DomNodeList```

1. `expr` defines the XPath1.0 expression
2. `args` defines a list of prefixes bound to namespaces in order to resolve qualified names in the XPath expression.

**Usage**
Returns a `xml.DomNodeList` object containing all DomNode objects matching a XPath 1.0 expression. `expr` is the XPath1.0 expression, `args` is a list of prefixes bound to namespaces in order to resolve qualified names in the XPath expression. This list must be filled with an even number of arguments, representing the prefix and its corresponding namespace.

**Important:** This method is not part of W3C standard API.

Valid example:
```java
selectByXPath("//d:Record",
   "d",
   "http://defaultnamespace")
selectByXPath(""/ns1:Record",
   NULL)
selectByXPath(""/ns1:Records/ns2:Record",
   ...)"
"ns1",
"http://namespace1",
"ns2",
"http://namespace2")

Invalid example:

```java
selectByXPath("//ns1:Record", "ns1")
```

This example is invalid because the namespace definition is missing.

If the namespaces list is NULL, the prefixes and namespaces defined in the document itself are used if available.

A namespace must be an absolute URI (for example, 'http://', 'file://').

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomDocument.setFeature
Sets a feature for a DomDocument object.

**Syntax**

```java
setFeature(
    property STRING,
    value STRING)
```

1. `property` is the name of the input parameter defining the name of a DomDocument feature.
2. `value` is the value of a feature.

**Usage**

Use this method to set a feature for the DomDocument object, where `property` is the name of a DomDocument feature, and `value` is the value of a feature.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomDocument.setXmlEncoding
Sets the XML document encoding in the XML declaration.

**Syntax**

```java
setXmlEncoding(
    encoding STRING)
```

1. `encoding` defines the XML document encoding.

**Usage**

Sets the XML document encoding in the XML declaration, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomDocument.setXmlStandalone
Sets the XML standalone attribute in the XML declaration to "yes" or "no" in the XML declaration, or removes the standalone attribute.

Syntax

```plaintext
setXmlStandalone(
    alone INTEGER )
```

1. `alone` defines a boolean flag.
   - `1` sets the standalone attribute to "yes".
   - `0` sets the standalone attribute to "no".
   - `-1` removes the standalone attribute.

Usage

This sample sets the standalone attribute to "no":

```plaintext
IMPORT XML
MAIN
    DEFINE doc xml.DomDocument
    LET doc = xml.DomDocument.Create()  
        CALL doc.setXmlStandalone(0)
    DISPLAY doc.saveToString()
END MAIN
```

The output displayed by the above example is

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
```

This sample sets the standalone attribute to "yes":

```plaintext
IMPORT XML
MAIN
    DEFINE doc xml.DomDocument
    LET doc = xml.DomDocument.Create()  
        CALL doc.setXmlStandalone(1)
    DISPLAY doc.saveToString()
END MAIN
```

The output displayed by the above example is

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
```

This sample removes the standalone attribute:

```plaintext
IMPORT XML
MAIN
    DEFINE doc xml.DomDocument
    LET doc = xml.DomDocument.Create()  
        CALL doc.setXmlStandalone(-1)
    DISPLAY doc.saveToString()
END MAIN
```
The output displayed by the above example is 

```xml
<?xml version="1.0" encoding="UTF-8"?>.
```

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**xml.DomDocument.validate**

Performs a DTD or XML Schema validation for a DomDocument object.

**Syntax**

```java
validate() 
RETURNS INTEGER
```

**Usage**

Performs a DTD or XML Schema validation for this DomDocument object. Returns the number of validation errors, or zero if there are none.

**Important:** This method is not part of W3C standard API.

See `setFeature()` to specify what kind of validation to do.

See `getErrorsCount()` and `getErrorDescription()` to retrieve error messages related to XML document.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**xml.DomDocument.validateOneElement**

Performs a DTD or XML Schema validation of a XML Element DomNode object.

**Syntax**

```java
validateOneElement( 
   elt xml.DomNode 
) 
RETURNS INTEGER
```

1. *elt* defines the XML element DomNode to validate.

**Usage**

Performs a DTD or XML Schema validation of a XML Element DomNode object; *elt* is the XML Element DomNode to validate.

Returns the number of validation errors, or zero if there are none.

**Important:** This method is not part of W3C standard API.

See `setFeature()` to specify what kind of validation to do.

See `getErrorsCount()` and `getErrorDescription()` to retrieve error messages related to XML document.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.
**Navigation methods usage examples**

Examples using the navigation methods of the xml.DomDocument class.

DomDocument navigation functions deal with nodes immediately under the DomDocument object, except for search features. To navigate through all the nodes, you can refer to the navigation functions of the class xml.DomNode.

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="card.xsl"?>
<!-- demo card -->
<CardList xml:id="1" >[...]
</CardList>
```

In the example the first node of the document is xml-stylesheet. Use `getFirstDocumentNode` to get the node. The element at position 2 is the comment `<!-- demo card -->`. Use `getDocumentNodeItem` function to get the node.

The last node of the document is CardList. Use `getLastDocumentNode` to get the node.

The number of nodes in the document is 3. This is the result of the function `getDocumentNodesCount`. This function only counts the number of children immediately under the DomDocument.

**Note:** The first line of the example, `<?xml version="1.0" encoding="ISO-8859-1"?>`, is not considered as a node. To access the information of the first line, use `getXmlVersion()` and `getXmlEncoding` functions.

Caution, if the example is in pretty printed format, the results are not the same. There are additional text nodes representing the carriage returns.

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="card.xsl"?>
<!-- demo card -->
<CardList xml:id="1" > [...
</CardList>
```

See **Cautions** section for more details.

You can select nodes using their tag names, by XPath, or by their attributes value (if of type ID, xml:id for example). The `getElementsbyTagName` and `getElementsbyTagNameNS` methods return a DomNodeList object, unlike the other methods that return a DomNode object. The DomNodeList is restricted to containing objects with the same tag name and/or namespace. The `selectByXPath` method also returns a DomNodeList object, but each node can have a different name.

```javascript
getElementsByTagNameNS("message","http://schemas.xmlsoap.org/wsd1/")
```

Get the message nodes that have `http://schemas.xmlsoap.org/wsd1/` as the namespace.

```javascript
getElementsByTagNameNS("message","*")
```

Get all the message nodes, regardless of the namespace they have.

```javascript
getElementsByTagName("message")
```

Get all the message nodes that do not have any namespace.

```javascript
selectByXPath("//xs:element",NULL)
```

Get all the xs:element nodes that have a namespace corresponding to the prefix `xs`.
Get all the Card nodes that do not have any namespace.

```javascript
getElementById("1")
```

Get the unique node whose attribute of type ID has a value of "1".

**Node creation methods usage examples**

Node creation methods usage examples for the `xml.DomDocument` class.

Creating a node for the `DomDocument` is done in two steps:

- Create the node.
- Add the node to the `DomDocument`.

Each time you create a node, you need to append it at the right place in the `DomDocument`. To add a node to the document use the `DomDocument` management methods or the `DomNode` manipulation methods.

```javascript
createNode("<LastName>PATTERSON</LastName><FirstName>Andrew</FirstName>")
```

Creates a structure of nodes.

```javascript
createElement("CardList")
```

Produces

```xml
<CardList/>
```

```javascript
createElementNS("cny", "Company", "http://www.mysite.com/")
```


```javascript
createAttribute("Country")
```

Creates a `Country` attribute node.

- To set a value to the attribute, use the method `setNodeValue` of the `xml.DomNode` class.
- To add the attribute to an element node, use the method `setAttributeNode` of the `xml.DomNode` class.

```javascript
createAttributeNS("tw","Town","http://www.mysite.com/cities")
```

Produces `xmlns:tw="http://www.mysite.com/cities" tw:Town=""`

- To set a value to the attribute use the method `setNodeValue` of the `xml.DomNode` class.
- To add the attribute to an element node use the method `setAttributeNodeNS` of the `xml.DomNode` class.
- For optimization reasons, the namespace is not written beside the attribute until the saving of the `DomDocument`.
- When accessing the element node, the namespace is not listed in the list of children. In the example above, `tw:Town=""` is in the list of children, not `xmlns:tw="http://www.mysite.com/cities"`.
- To access the namespace during the `DomDocument` building, use the method `normalize` first. Normalize writes the namespace declaration at the appropriate place. If there is no previous declaration, it will be accessible as an attribute of this element, otherwise it will be an attribute of one of the ancestors of the element.

```javascript
createTextNode("My Company")
```

Creates a text node.

```javascript
createComment("End of the card")
```
Produces <!--End of the card-->

createCDATASection("<website><a href="www.mysite.com">My Company</a></website>")

Produces <![CDATA[<website><a href="www.mysite.com">My Company</a></website>]]>

createEntityReference("title")

Creates the entity reference &title.

createProcessingInstruction("xml-stylesheet", "type="text/xsl" href="card.xsl""")

Produces <?xml-stylesheet type="text/xsl" href="card.xsl"?>

createDocumentType("Card", NULL, NULL,"<!ELEMENT Card (lastname, firstname, company, location)>")

Produces <!DOCTYPE Card [ <!ELEMENT Card (lastname, firstname, company, location)>]

• Only inline DTD are supported. The DTD has to been inserted in the DomDocument at an appropriate place.

createDocumentFragment

Is a method that creates a lightweight DomDocument. It represents a subtree of nodes that do not need to conform to well-formed XML rules. This makes DocumentFragment easier to manipulate than a DomDocument.

for i=1 to 5
  let node = doc.createElement("Card")
  call root.appendChild(node) end for

This produces a subtree with 5 Card nodes that do not have any root node. Once the subtree is completed, it can be added to the DomDocument object like any other node.

**HTML document usage example**
The HTML language provides tags that allow the user to provide an embedded style sheet (the "style" tag) and to write embedded client side script (the "script" tag). According to the HTML 4.0 specification, the content of these tags must be managed as CDATA section.

**Note:** For more information, see the HTML 4.0 specification.

Because HTML document management via the xml.DomDocument object provides HTML compliancy only (and not strict HTML management), there is a specific way to add these nodes inside a loaded HTML document:

1. Create an element node with the name of the tag to be created.
2. Append the element node to its parent.
3. Create a CDATASection node with the required embedded piece of style sheet or piece of script content.
4. Append the CDATASection node to the previously-created element node.

By following this procedure, the "script" and "style" tags content are recognized as CDATA section content and not TEXT section content and will be preserved. Other methods for adding nodes to the document manage text and therefore will not treat these types of content properly, resulting in invalid HTML code.

**Example**

```ruby
IMPORT xml
MAIN
  DEFINE myDoc xml.DomDocument
  DEFINE myEltNode, myAttrNode, bodyNode, myCdataNode xml.DomNode
```
DEFINE nodeLst xml.DomNodeList
TRY
LET myDoc = xml.DomDocument.create()
CALL myDoc.setFeature("enable-html-compliancy", 1)
CALL myDoc.load("testHtml.html")
LET myEltNode = myDoc.createElement("script")
LET myCdataNode = myDoc.createCDATASection("document.write("CDATA");")
LET myAttrNode = myDoc.createAttribute("type")
CALL myAttrNode.setNodeValue("text/javascript")
LET nodeLst = myDoc.getElementsByTagName("body")
LET bodyNode = nodeLst.getItem(1)
CALL bodyNode.appendChild(myEltNode)
CALL myEltNode.setAttributeNode(myAttrNode)
CALL myEltNode.appendChild(myCdataNode)
CATCH
DISPLAY "ERROR : ", STATUS, " - ", SQLCA.SQLERRM
EXIT PROGRAM(-1)
END TRY
END MAIN

Load and save methods usage examples
Load and save method usage examples for the xml.DomDocument class.
You can load an existing XML document. Before loading an XML document you need to create the DomDocument object.
A DomDocument can load files using different URI: http://, https://, tcp://, tcps://, file:// and alias://.
See getErrorsCount() and getErrorDescription() to retrieve error messages related to XML document.

load("data.xml")
load("http://www.w3schools.com/xml/cd_catalog.xml")
load("https://localhost:6394/ws/r/calculator?WSDL")
load("file:///data/cd_catalog.xml")
load("tcp://localhost:4242/")
load("tcps://localhost:4243/")
load("alias://demo")

In the example the demo alias is defined in fglprofile as ws.demo.url = "http://www.w3schools.com/xml/cd_catalog.xml"

loadfromstring("<List> <elt>First element</elt> <elt>Second element</elt> <elt>Third element</elt> </List>")
The example, produces a subtree with a root node List and three nodes elt and three textnode.
A DomDocument can be saved at different URI beginning with: http://, https://, tcp://, tcps://, file:// and alias://.

save("myfile.xml")
save("http://myserver:8080/data/save1.xml")
save("file:///data/save.xml")
save("tcp://localhost:4242/")
save("alias://test")

In the example the test alias is defined in fglprofile as ws.test.url = "http://localhost:8080/data/save3.xml"
The saveToString method saves the DomDocument in a string.
See getErrorsCount() and getErrorDescription() to retrieve error messages related to XML document.
The normalize function emulates a DomDocument save and load. It can be called at any stage of the DomDocument building. This removes empty Text nodes and sets namespace declarations as if the document had been saved.

**Cautions**

Some things you need to be aware of when working with the xml.DomDocument class.

Whitespaces, line feeds and carriage returns between elements are represented as text nodes in memory. A XML document written in a single line and a human readable (pretty printed format) do not have the same representation in the DomDocument. Take this into account when navigating in the document.

If a xml.DomNode is not attached to a DomDocument and not referenced by any variable it can be destroyed. If one child of this node is still referenced, this child is not destroyed but its parent and the other nodes of the subtree are destroyed. To check if a node is attached to a DomDocument use isAttached method.

The DomDocument remains in memory if any of its nodes are still referenced in a variable.

**DomDocument Features**

A list of features for the xml.DomDocument class.

**DomDocument features**

**Table 641: DomDocument Features**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>format-pretty-print</td>
<td>Formats the output by adding whitespace to produce a pretty-printed, indented,</td>
</tr>
<tr>
<td></td>
<td>human-readable form. Possible values are TRUE or FALSE. Default value is</td>
</tr>
<tr>
<td></td>
<td>FALSE.</td>
</tr>
<tr>
<td>comments</td>
<td>Defines whether the XML comments are kept during the loading of a document</td>
</tr>
<tr>
<td></td>
<td>into a DomDocument object. Possible values are TRUE or FALSE. Default</td>
</tr>
<tr>
<td></td>
<td>value is TRUE.</td>
</tr>
<tr>
<td>whitespace-in-element-content</td>
<td>Defines whether XML Text nodes that can be considered &quot;Ignorable&quot; are kept</td>
</tr>
<tr>
<td></td>
<td>during the loading of an XML document into a DomDocument object. Possible</td>
</tr>
<tr>
<td></td>
<td>values are TRUE or FALSE. Default value is TRUE.</td>
</tr>
<tr>
<td>cdata-sections</td>
<td>Defines whether XML CDATA nodes are kept or replaced by XML Text nodes</td>
</tr>
<tr>
<td></td>
<td>during the loading of an XML document into a DomDocument object. Possible</td>
</tr>
<tr>
<td></td>
<td>values are TRUE or FALSE. Default value is TRUE.</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>expand-entity-references</td>
<td>Defines whether XML EntityReference nodes are kept or replaced during the loading of an XML document into a DomDocument object. Possible values are TRUE or FALSE. Default value is FALSE.</td>
</tr>
<tr>
<td>validation-type</td>
<td>Defines what kind of validation is performed. Possible values are: DTD, Schema. Default is Schema.</td>
</tr>
<tr>
<td>external-schemaLocation</td>
<td>Defines a list of namespace-qualified XML schemas to use for validation on a DomDocument object. Value is a space-separated string of one or several pairs of strings representing the namespace URI of the schema, followed by its location.</td>
</tr>
<tr>
<td></td>
<td>Example:</td>
</tr>
<tr>
<td>external-noNamespaceSchemaLocation</td>
<td>Defines a list of XML schemas to use for validation on a DomDocument object. Value is a space-separated string of one or several strings representing the location of a schema.</td>
</tr>
<tr>
<td></td>
<td>Example:</td>
</tr>
<tr>
<td></td>
<td>&quot;mySchema1.xsd mySchema2.xsd&quot;</td>
</tr>
<tr>
<td>schema-uriRecovery</td>
<td>Changes the schema location of an XML schema referenced by import tags in other schemas. Value is a space-separated string of one or several pairs of strings representing the original schema location followed by the new schema location</td>
</tr>
<tr>
<td></td>
<td>Example:</td>
</tr>
<tr>
<td></td>
<td>&quot;<a href="http://www.w3.org/2001/xml.xsd">http://www.w3.org/2001/xml.xsd</a> myXML.xsd <a href="http://www.mycompany.com/">http://www.mycompany.com/</a> GWS.xsd myGWS.xsd&quot;</td>
</tr>
<tr>
<td>load-save-base64-string</td>
<td>Changes methods loadFromString() and saveToString() to handle Base64 strings. Parsing an XML document is done from a BASE64 encoded string, and saving an XML document results in a BASE64 encoded string. Possible values are TRUE or FALSE. Default is FALSE.</td>
</tr>
</tbody>
</table>
### Name | Description
--- | ---
**auto-id-attribute** | Changes the parsing of an XML document in order to set all unqualified attributes named ID, Id, iD or id to be of type ID.

They can then be retrieved with method `getElementById()` or with an XPath expression without calling `setIdAttribute()`.

Possible values are TRUE or FALSE.

Default is FALSE.

**auto-id-qualified-attribute** | Changes the parsing of an XML document in order to set all qualified attributes named ID, Id, iD or id to be of type ID.

They can then be retrieved with method `getElementById()` or with an XPath expression without calling `setIdAttributeNS()`.

Possible values are TRUE or FALSE.

Default is FALSE.

**enable-html-compliancy** | Changes methods to parse, normalize and save HTML document via the `DomDocument` object.

Possible values are TRUE or FALSE.

Default value is FALSE.

The HTML parsing isn't namespace qualified, and document is considered as an XML document after loading.

**Note:** This feature works only for HTML 4, it is not supported for HTML 5.

### Security issues with expand-entity-references


For example, in its DTD, the following XML file defines the `myref` ENTITY element referencing the `/etc/passwd` file:

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [ 
  <!ELEMENT foo ANY >
  <!ENTITY myref SYSTEM "file:///etc/passwd" >
]>
<foo>&myref;</foo>
```

When loading this XML file with `expand-entity-references` set to TRUE, the resulting DOM document will have a `<foo>` node containing a text node with the content of `/etc/passwd`. 
Examples
xml.DomDocument usage examples.
Example 1 : Create a namespace qualified document with processing instructions

To create the following XML document on disk:

```xml
<?Target1 This is my first PI ?>
<MyPre:RootNode xmlns:MyPre="http://www.tempuri.org" >
<MyPre:Element />
</MyPre:RootNode>
<?Target2 This is my last PI ?>
```

Write the following code:

```xml
IMPORT xml

MAIN
  DEFINE doc xml.DomDocument
  DEFINE pi xml.DomNode
  DEFINE node xml.DomNode
  DEFINE elt xml.DomNode

  # Create a document with an initial namespace qualified root node
  # Create a Processing instruction
  LET pi = doc.createProcessingInstruction("Target1", "This is my first PI")
  # And add it at the begining of the document
  CALL doc.prependDocumentNode(pi)
  # Create another Processing instruction
  LET pi = doc.createProcessingInstruction("Target2", "This is my last PI")
  # And add it at the end of the document
  CALL doc.appendDocumentNode(pi)
  # Retrieve initial root node of the document
  LET elt = doc.getDocumentElement()
  # Create a new Element node
  LET node = doc.createElementNS("MyPre", "Element", "http://www.tempuri.org")
  # And add it as child of the rootNode
  CALL elt.appendChild(node)
  # Then save the document on disk
  CALL doc.save("MyFile.xml")
END MAIN
```

Example 2 : Validating a document against XML schemas or a DTD

This code example loads one or more XML schemas or uses an embedded DTD to validate against a XML document:

```xml
IMPORT xml

MAIN
  DEFINE location STRING
  DEFINE xmlfile STRING
  DEFINE doc xml.DomDocument
  DEFINE ind INTEGER

  IF num_args()<2 THEN
    # Checks the number of arguments
    CALL ExitHelp()
  ELSE
    LET doc = xml.DomDocument.Create()
    LET xmlfile = arg_val(num_args())
    IF num_args() == 2 AND arg_val(1) == "-dtd" THEN
```
# User chose DTD validation
CALL doc.setFeature("validation-type", "DTD")
ELSE
# User chose XML Schema validation
IF arg_val(1) == "-ns" THEN
    # Handle namespace qualified XML schemas
    IF num_args() MOD 2 != 0 THEN
        CALL ExitHelp()
    END IF
    FOR ind = 2 TO num_args()-1 STEP 2
        IF location IS NULL THEN
            LET location = arg_val(ind) || " " || arg_val(ind+1)
        ELSE
            LET location = location || " " || arg_val(ind) || " " || arg_val(ind+1)
        END IF
    END FOR
    TRY
        CALL doc.setFeature("external-schemaLocation", location)
    CATCH
        FOR ind = 1 TO doc.getErrorsCount()
            DISPLAY "Schema error (" || ind || ") :", doc.getErrorDescription(ind)
        END FOR
        EXIT PROGRAM (-1)
    END TRY
ELSE
    # Handle unqualified XML schemas
    FOR ind = 1 TO num_args()-1
        IF location IS NULL THEN
            LET location = arg_val(ind)
        ELSE
            LET location = location || " " || arg_val(ind)
        END IF
    END FOR
    TRY
        CALL doc.setFeature("external-noNamespaceSchemaLocation", location)
    CATCH
        FOR ind = 1 TO doc.getErrorsCount()
            DISPLAY "Schema error (" || ind || ") :", doc.getErrorDescription(ind)
        END FOR
        EXIT PROGRAM (-1)
    END TRY
END IF
END IF
TRY
    # Load XML document from disk
    CALL doc.load(xmlfile)
CATCH
    # Display errors if loading fails
    IF doc.getErrorsCount() > 0 THEN
        FOR ind = 1 TO doc.getErrorsCount()
            DISPLAY "LOADING ERROR #" || ind || ":", doc.getErrorDescription(ind)
        END FOR
        EXIT PROGRAM(-1)
    ELSE
        DISPLAY "Unable to load file :", xmlfile
    END IF
END TRY
TRY
# Validate loaded document
LET ind = doc.validate()
IF ind == 0 THEN
    # Successful validation
    DISPLAY "OK"
ELSE
    # Display validation errors
    FOR ind = 1 TO doc.getErrorsCount()
        DISPLAY "VALIDATING ERROR #"||ind||" :",doc.getErrorDescription(ind)
    END FOR
    EXIT PROGRAM(-1)
END IF
CATCH
    DISPLAY "Unable to validate file :",xmlfile
    EXIT PROGRAM(-1)
END TRY
END MAIN

# Display help
FUNCTION ExitHelp()
    DISPLAY "Validator < -dtd | -ns [namespace schema]+ | [schema]+ > xmlfile"
    EXIT PROGRAM
END FUNCTION

Example

$ fglrun Validator -dtd MyFile.xml
Validates XML file using DTD embedded in the XML file.

$ fglrun Validator Schema1.xsd Schema2.xsd MyFile.xml
Validates unqualified XML file using two unqualified XML schemas.

Validates namespace qualified XML file using two namespace qualified XML schemas.

The **DOMNode** class
The **xmlDOMNode** class provides methods to manipulate a node of a DomDocument object.

You can create a DomNode object using creation methods in the **DomDocument** class.

The **STATUS** variable is set to zero after a successful method call.
### xmlDOMNode methods
Methods for the xmlDOMNode class.

#### Table 642: Object methods: Navigation

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getChildrenCount()</code></td>
<td>RETURNS INTEGER Returns the number of child DomNode objects for a DomNode object.</td>
</tr>
<tr>
<td><code>getChildNodeItem(index)</code></td>
<td>RETURNS xmlDOMNode Returns the child DomNode object at a given position for a DomNode object.</td>
</tr>
<tr>
<td><code>getFirstChild()</code></td>
<td>RETURNS xmlDOMNode Returns the first child DomNode object for this XML Element DomNode object.</td>
</tr>
<tr>
<td><code>getFirstChildElement()</code></td>
<td>RETURNS xmlDOMNode Returns the first XML Element child DomNode object for this DomNode object.</td>
</tr>
<tr>
<td><code>getLastChild()</code></td>
<td>RETURNS xmlDOMNode Returns the last child DomNode object for a XML Element DomNode object.</td>
</tr>
<tr>
<td><code>getLastChildElement()</code></td>
<td>RETURNS xmlDOMNode Returns the last child XML element DomNode object for this DomNode object.</td>
</tr>
<tr>
<td><code>getNextSibling()</code></td>
<td>RETURNS xmlDOMNode Returns the DomNode object immediately following a DomNode object.</td>
</tr>
<tr>
<td><code>getNextSiblingElement()</code></td>
<td>RETURNS xmlDOMNode Returns the XML Element DomNode object immediately following a DomNode object.</td>
</tr>
<tr>
<td><code>getParentNode()</code></td>
<td>RETURNS xmlDOMNode Returns the parent DomNode object for this DomNode object.</td>
</tr>
<tr>
<td><code>getPreviousSibling()</code></td>
<td>RETURNS xmlDOMNode Returns the DomNode object immediately preceding a DomNode object.</td>
</tr>
<tr>
<td><code>getPreviousSiblingElement()</code></td>
<td>RETURNS xmlDOMNode Returns the XML Element DomNode object immediately preceding a DomNode object.</td>
</tr>
<tr>
<td><code>hasChildNodes()</code></td>
<td>RETURNS INTEGER Returns TRUE if a node has child nodes.</td>
</tr>
</tbody>
</table>
### Table 643: Object methods: Manipulation

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| clone(  
  deep INTEGER )  
  RETURNS xml.DomNode | Returns a duplicate DomNode object of a node. |
| appendChild(  
  newNode xml.DomNode ) | Adds a child DomNode object to the end of the child list for a DomNode object. |
| appendChildElement(  
  name STRING )  
  RETURNS xml.DomNode | Creates and adds a child XML Element node to the end of the list of child nodes for an XML Element DomNode object. |
| appendChildElementNS(  
  prefix STRING,  
  name STRING,  
  ns STRING )  
  RETURNS xml.DomNode | Creates and adds a child namespace qualified XML Element node to the end of the list of child nodes for an XML Element DomNode object. |
| addNextSibling(  
  newNode xml.DomNode ) | Adds a DomNode object as the next sibling of a DomNode object. |
| addPreviousSibling(  
  newNode xml.DomNode ) | Adds a DomNode object as the previous sibling of a DomNode object. |
| insertBeforeChild(  
  newNode xml.DomNode,  
  refChild xml.DomNode ) | Inserts a DomNode object before an existing child DomNode object. |
| insertAfterChild(  
  newNode xml.DomNode,  
  refChild xml.DomNode ) | Inserts a DomNode object after an existing child DomNode object. |
| prependChild(  
  newNode xml.DomNode ) | Adds a child DomNode object to the beginning of the child list for a DomNode object. |
| prependChildElement(  
  name STRING )  
  RETURNS xml.DomNode | Creates and adds a child XML Element node to the beginning of the list of child nodes for this XML Element DomNode object. |
| prependChildElementNS(  
  prefix STRING,  
  name STRING,  
  ns STRING )  
  RETURNS xml.DomNode | Creates and adds a child namespace-qualified XML Element node to the beginning of the list of child nodes for an XML Element DomNode object. |
| removeAllChildren() | Removes all child DomNode objects from a DomNode object. |
| removeChild(  
  oldChild xml.DomNode ) | Removes a child DomNode object from the list of child DomNode objects. |
### Table 644: Object methods: Access

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getLocalName()</code></td>
<td>GETS the local name for a DomNode object.</td>
</tr>
<tr>
<td><code>getNamespaceURI()</code></td>
<td>Returns the namespace URI for a DomNode object.</td>
</tr>
<tr>
<td><code>getNodeName()</code></td>
<td>GETS the name for a DomNode object.</td>
</tr>
<tr>
<td><code>getNodeType()</code></td>
<td>GETS the XML type for this DomNode object.</td>
</tr>
<tr>
<td><code>getNodeValue()</code></td>
<td>Returns the value for a DomNode object.</td>
</tr>
<tr>
<td><code>getPrefix()</code></td>
<td>Returns the prefix for a DomNode object.</td>
</tr>
<tr>
<td><code>isAttached()</code></td>
<td>Returns whether the node is attached to the XML document.</td>
</tr>
</tbody>
</table>

### Table 645: Object methods: Modifier

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setNodeValue()</code></td>
<td>Sets the node value for a DomNode object.</td>
</tr>
<tr>
<td><code>setPrefix()</code></td>
<td>Sets the prefix for a DomNode object.</td>
</tr>
<tr>
<td><code>toString()</code></td>
<td>Returns a string representation of a DomNode object.</td>
</tr>
</tbody>
</table>
Table 646: Object methods: Attribute

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hasAttribute</code> (name STRING )</td>
<td>Checks whether an XML Element DomNode object has the XML Attribute specified by a specified name.</td>
</tr>
<tr>
<td>RETURNS INTEGER</td>
<td></td>
</tr>
<tr>
<td><code>hasAttributeNS</code> (name STRING, ns STRING )</td>
<td>Checks whether a namespace qualified XML Attribute of a given name is carried by an XML Element DomNode object.</td>
</tr>
<tr>
<td>RETURNS INTEGER</td>
<td></td>
</tr>
<tr>
<td><code>getAttributesCount</code> ()</td>
<td>Returns the number of XML Attribute DomNode objects on this XML Element DomNode object.</td>
</tr>
<tr>
<td>RETURNS INTEGER</td>
<td></td>
</tr>
<tr>
<td><code>getAttributeNode</code> (name STRING )</td>
<td>Returns an XML Attribute DomNode object for an XML Element DomNode object.</td>
</tr>
<tr>
<td>RETURNS xml.DomNode</td>
<td></td>
</tr>
<tr>
<td><code>getAttributeNodeItem</code> (index INTEGER )</td>
<td>Returns the XML Attribute DomNode object at a given position on this XML Element DomNode object.</td>
</tr>
<tr>
<td>RETURNS xml.DomNode</td>
<td></td>
</tr>
<tr>
<td><code>getAttributeNodeNS</code> (name STRING, ns STRING )</td>
<td>Returns the value of a namespace-qualified XML Attribute DomNode object for an XML Element DomNode object.</td>
</tr>
<tr>
<td>RETURNS xml.DomNode</td>
<td></td>
</tr>
<tr>
<td><code>getAttribute</code> (name STRING )</td>
<td>Returns the value of a XML Attribute for an XML Element DomNode object.</td>
</tr>
<tr>
<td>RETURNS STRING</td>
<td></td>
</tr>
<tr>
<td><code>getAttributeNS</code> (name STRING, ns STRING )</td>
<td>Returns the value of a namespace qualified XML Attribute for an XML Element DomNode object.</td>
</tr>
<tr>
<td>RETURNS STRING</td>
<td></td>
</tr>
<tr>
<td><code>hasAttributes</code> ()</td>
<td>Identifies whether a node has XML Attribute nodes.</td>
</tr>
<tr>
<td>RETURNS INTEGER</td>
<td></td>
</tr>
<tr>
<td><code>setAttribute</code> (name STRING, value STRING )</td>
<td>Sets (or resets) an XML Attribute for an XML Element DomNode object.</td>
</tr>
<tr>
<td><code>setAttributeNode</code> (attr xml.DomNode )</td>
<td>Sets (or resets) an XML Attribute DomNode object to an XML Element DomNode object.</td>
</tr>
<tr>
<td><code>setAttributeNodeNS</code> (attr xml.DomNode )</td>
<td>Sets (or resets) a namespace-qualified XML Attribute DomNode object to an XML Element DomNode object.</td>
</tr>
<tr>
<td><code>setAttributeNS</code> (prefix STRING, name STRING, ns STRING, value STRING )</td>
<td>Sets (or resets) a namespace-qualified XML Attribute for an XML Element DomNode object.</td>
</tr>
</tbody>
</table>
Table 647: Object methods: Search

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getElementsByTagName (tag STRING )</code></td>
<td>RETURNS xml.DomNodeList</td>
</tr>
<tr>
<td><code>getElementsByTagNameNS (tag STRING, ns STRING )</code></td>
<td>RETURNS xml.DomNodeList</td>
</tr>
<tr>
<td><code>isDefaultNamespace (ns STRING )</code></td>
<td>RETURNS INTEGER</td>
</tr>
<tr>
<td><code>lookupNamespaceURI (prefix STRING )</code></td>
<td>RETURNS STRING</td>
</tr>
<tr>
<td><code>lookupPrefix (ns STRING )</code></td>
<td>RETURNS STRING</td>
</tr>
<tr>
<td><code>selectByXPath (expr STRING, args ...)</code></td>
<td>RETURNS xml.DomNodeList</td>
</tr>
</tbody>
</table>

xml.DomNode.addPreviousSibling

Adds a DomNode object as the previous sibling of a DomNode object.

**Syntax**

```
addPreviousSibling(
    newNode xml.DomNode  )
```

1. `newNode` defines the node to add.

**Usage**

Adds a DomNode object as the previous sibling of this DomNode object; `newNode` is the node to add.

**Important:** This method is not part of W3C standard API.

The DomNode object node must be the child of an element or document node; otherwise the operation fails.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomNode.addNextSibling
Adds a DomNode object as the next sibling of a DomNode object.

**Syntax**

```plaintext
addNextSibling(
    newNode xml.DomNode
)
```

1. `newNode` defines the node to add.

**Usage**

Adds a DomNode object as the next sibling of this DomNode object; `newNode` is the node to add.

**Important:** This method is not part of W3C standard API.

The DomNode object node must be the child of an element or document node; otherwise the operation fails.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**xml.DomNode.appendChild**

Adds a child DomNode object to the end of the child list for a DomNode object.

**Syntax**

```plaintext
appendChild(
    newChild xml.DomNode
)
```

1. `newChild` defines the node to add.

**Usage**

Adds a child DomNode object to the end of the child list for this DomNode object.

The DomNode object node must be the child of an element or document node; otherwise the operation fails.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**xml.DomNode.appendChildElement**

Creates and adds a child XML Element node to the end of the list of child nodes for an XML Element DomNode object.

**Syntax**

```plaintext
appendChildElement(
    name STRING
)
RETURNS xml.DomNode
```

1. `name` defines the XML Element name.

**Usage**

Creates and adds a child XML Element node to the end of the list of child nodes for this XML Element DomNode object.

**Important:** This method is not part of W3C standard API.

Returns the XML element DomNode object, or NULL.
In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**xml.DomNode.appendChildElementNS**

Creates and adds a child namespace qualified XML Element node to the end of the list of child nodes for an XML Element DomNode object.

**Syntax**

```java
appendChildElementNS (prefix STRING, name STRING, ns STRING) RETURNS xml.DomNode
```

1. prefix defines the prefix of the XML Element to add.
2. name defines the name of the XML Element to add.
3. ns defines the namespace URI of the XML Element to add.

**Usage**

Creates and adds a child namespace qualified XML Element node to the end of the list of child nodes for this XML Element DomNode object.

**Important:** This method is not part of W3C standard API.

Returns the XML element DomNode object, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**xml.DomNode.clone**

Returns a duplicate DomNode object of a node.

**Syntax**

```java
clone (deep INTEGER) RETURNS xml.DomNode
```

1. deep defines a boolean. If deep is TRUE, child DomNode objects are cloned too; otherwise only the DomNode itself is cloned.

**Usage**

Returns a duplicate DomNode object of this node. If deep is TRUE, child DomNode objects are cloned too; otherwise only the DomNode itself is cloned.

Returns a copy of this xml.DomNode object, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**xml.DomNode.getAttribute**
Returns the value of a XML Attribute for an XML Element DomNode object

Syntax

```java
getAttribute(
    name STRING )
RETURNS STRING
```

1. `name` defines the name of the XML attribute to retrieve.

Usage

Returns the value of a XML Attribute for this XML Element DomNode object, where `name` is the name of the XML attribute to retrieve.

Returns the XML attribute value, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomNode.getAttributeNode

Returns an XML Attribute DomNode object for an XML Element DomNode object

Syntax

```java
getAttributeNode(
    name STRING )
RETURNS xml.DomNode
```

1. `name` defines the name of the attribute to retrieve.

Usage

Returns an XML attribute DomNode object for this XML Element DomNode object, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomNode.getAttributeNodeItem

Returns the XML Attribute DomNode object at a given position on this XML Element DomNode object.

Syntax

```java
getAttributeNodeItem(
    index INTEGER )
RETURNS xml.DomNode
```

1. `index` defines the position of the node to return.

Usage

Returns the XML Attribute DomNode object at a given position on this XML Element DomNode object, where `index` is the position of the node to return (Index starts at 1).

Returns the XML Attribute DomNode object at the given position, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.
xml.DomNode.getAttributeNodeNS
Returns a namespace-qualified XML Attribute DomNode object for an XML Element DomNode object

Syntax
```java
getAttributeNodeNS (        
    name STRING,        
    ns STRING )        
RETURNS xml.DomNode
```

1. `name` defines the name of the XML Attribute to retrieve.
2. `ns` defines the namespace URI of the XML Attribute to retrieve.

Usage
Returns a namespace-qualified XML Attribute DomNode object for this XML Element DomNode object, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomNode.getAttributeNS
Returns the value of a namespace qualified XML Attribute for an XML Element DomNode object

Syntax
```java
getAttributeNS (        
    name STRING,        
    ns STRING )        
RETURNS STRING
```

1. `name` defines the name.
2. `ns` defines the namespace URI of the XML Attribute to retrieve

Usage
Returns the value of a namespace qualified XML Attribute for this XML Element xml.DomNode object, where `name` is the name and `ns` is the namespace URI of the XML Attribute to retrieve.

Returns the XML attribute value, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomNode.getAttributesCount
Returns the number of XML Attribute DomNode objects on this XML Element DomNode object.

Syntax
```java
getAttributesCount ()        
RETURNS INTEGER
```

Usage
Returns the number of XML Attribute xml.DomNode objects on this XML Element DomNode object.
In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomNode.getChildNodeItem
Returns the child DomNode object at a given position for a DomNode object.

Syntax

```plaintext
getChildNodeItem(
   index INTEGER )
RETURNS xml.DomNode
```

1. *index* defines the position of the child node in the collection.

Usage

Returns the child DomNode object at a given position for this DomNode object.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomNode.getChildrenCount
Returns the number of child DomNode objects for a DomNode object.

Syntax

```plaintext
getChildrenCount ()
RETURNS INTEGER
```

Usage

Returns the number of child xml.DomNode objects for this DomNode object.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomNode.getElementsByTagName
Returns a DomNodeList object containing all XML Element DomNode objects with the same tag name.

Syntax

```plaintext
getElementsByTagName (tag STRING )
RETURNS xml.DomNodeList
```

1. *tag* defines the name of the XML Element tag to match or "*" to match all tags.

Usage

Returns a DomNodeList object containing all XML Element DomNode objects with the same tag name, or NULL; *tag* is the name of the XML Element tag to match, or "*" to match all tags.

The getElementsByTagName and getElementsByTagNameNS methods return a xml.DomNodeList object, unlike the other methods that return a DomNode object. The DomNodeList is restricted to objects with the same tag name and/or namespace.
In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomNode.getElementsByTagNameNS

Returns a DomNodeList object containing all namespace-qualified XML Element DomNode objects with the same tag name and namespace.

**Syntax**

```java
getElementsByTagNameNS (  
tag STRING,  
ns STRING  )  
RETURNS xml.DomNodeList
```

1. *tag* defines the name of the XML Element tag to match or "*" to match all tags.
2. *ns* defines the namespace URI of the XML Element tag to match or "*" to match any namespace.

**Usage**

Returns a DomNodeList object containing all namespace-qualified XML Element DomNode objects with the same tag name and namespace, or NULL. *tag* is the name of the XML Element tag to match, or "*" to match all tags; *ns* is the namespace URI of the XML Element tag to match, or "*" to match any namespace.

The `getElementsByTagNameNS` and `getElementsByTagName` methods return a DomNodeList object, unlike the other methods that return a DomNode object. The DomNodeList is restricted to contain objects with the same tag name and/or namespace.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomNode.getFirstChild

Returns the first child DomNode object for this XML Element DomNode object.

**Syntax**

```java
getFirstChild ()  
RETURNS xml.DomNode
```

**Usage**

Returns the first child DomNode object for this XML Element DomNode object, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomNode.getFirstChildElement

Returns the first XML Element child DomNode object for this DomNode object.

**Syntax**

```java
getFirstChildElement ()  
RETURNS xml.DomNode
```

**Usage**

Returns the first XML Element child DomNode object for this DomNode object, or NULL.
**Important:** This method is not part of W3C standard API.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**xml.DomNode.getLastChild**

Returns the last child `DomNode` object for a XML Element `DomNode` object.

**Syntax**

```java
getLastChild()
RETURNS xml.DomNode
```

**Usage**

Returns the last child `DomNode` object for this XML Element `DomNode` object, or NULL.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**xml.DomNode.getLastChildElement**

Returns the last child XML element `DomNode` object for this `DomNode` object.

**Syntax**

```java
getLastChildElement()
RETURNS xml.DomNode
```

**Usage**

Returns the last child XML element `DomNode` object for this `DomNode` object, or NULL.

**Important:** This method is not part of W3C standard API.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**xml.DomNode.getLocalName**

Gets the local name for a `DomNode` object.

**Syntax**

```java
getLocalName()
RETURNS STRING
```

**Usage**

Gets the local name for this `DomNode` object. If `xml.DomNode` has a qualified name, only the local part is returned.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**xml.DomNode.getNamespaceURI**
Returns the namespace URI for a DomNode object.

**Syntax**

```plaintext
getNamespaceURI ()
RETURNS STRING
```

**Usage**

This method returns the namespace URI for this xml.DomNode object, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**xml.DomNode.getNextSibling**

Returns the DomNode object immediately following a DomNode object.

**Syntax**

```plaintext
getNextSibling ()
RETURNS xml.DomNode
```

**Usage**

Returns the xml.DomNode object immediately following this DomNode object, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**xml.DomNode.getNextSiblingElement**

Returns the XML Element DomNode object immediately following a DomNode object.

**Syntax**

```plaintext
getNextSiblingElement ()
RETURNS xml.DomNode
```

**Usage**

Returns the XML Element xml.DomNode object immediately following this DomNode object, or NULL.

**Important:** This method is not part of W3C standard API.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**xml.DomNode.getNodeName**

Gets the name for a DomNode object.

**Syntax**

```plaintext
getNodeName ()
RETURNS STRING
```
Usage

Gets the name for this `xml.DomNode` object. This method returns the qualified name of the `DomNode` object, or NULL. If `DomNode` does not have a qualified name, the local part is returned.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

`xml.DomNode.getNodeType`

Gets the XML type for this `DomNode` object.

**Syntax**

```
getNodeType()
RETURNS STRING
```

Usage

This method returns the XML type for this `DomNode` object; it returns one of the XML `DomNode` types, or NULL.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

`xml.DomNode.getNodeValue`

Returns the value for a `DomNode` object.

**Syntax**

```
getNodeValue()
RETURNS STRING
```

Usage

Use this method to return the value for this `xml.DomNode` object, or NULL.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

`xml.DomNode.getOwnerDocument`

Returns the `DomDocument` object containing this `DomNode` object.

**Syntax**

```
getOwnerDocument()
RETURNS `xml.DomDocument`
```

Usage

This method returns the `xml.DomDocument` object containing this `DomNode` object, or NULL.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

`xml.DomNode.getParentNode`
Returns the parent DomNode object for this DomNode object.

**Syntax**

```java
getParentNode()
RETURNS xml.DomNode
```

**Usage**

This method returns the parent xml.DomNode object for this DomNode object, or NULL. In the case of a xml.DomDocument node, this method will return NULL (parent is not a DomNode object) but `isAttached()` will return TRUE.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomNode.getPrefix

Returns the prefix for a DomNode object.

**Syntax**

```java
getPrefix()
RETURNS STRING
```

**Usage**

This method returns the prefix for this xml.DomNode object, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomNode.getPreviousSibling

Returns the DomNode object immediately preceding a DomNode object.

**Syntax**

```java
getPreviousSibling()
RETURNS xml.DomNode
```

**Usage**

Use this method to return the xml.DomNode object immediately preceding this DomNode object, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomNode.getPreviousSiblingElement

Returns the XML Element DomNode object immediately preceding a DomNode object.

**Syntax**

```java
getPreviousSiblingElement()
RETURNS xml.DomNode
```
Usage

Use this method to return the XML Element `DomNode` object immediately preceding this `DomNode` object, or NULL.

**Important:** This method is not part of W3C standard API.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (`STATUS`) on page 3554.

xml.DomNode.hasAttribute

Checks whether an XML Element `DomNode` object has the XML Attribute specified by a specified name.

Syntax

```plaintext
hasAttribute(
    name STRING
) RETURNS INTEGER
```

1. `name` defines the object name to check.

Usage

Checks whether this XML Element `DomNode` object has the XML Attribute specified by `name`. Returns TRUE if an XML Attribute with the given name is carried by this XML Element `DomNode` object, otherwise returns FALSE.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (`STATUS`) on page 3554.

xml.DomNode.hasAttributeNS

Checks whether a namespace qualified XML Attribute of a given name is carried by an XML Element `DomNode` object.

Syntax

```plaintext
hasAttributeNS(
    name STRING,
    ns STRING
) RETURNS INTEGER
```

1. `name` defines the name of the XMLAttribute to check
2. `ns` defines the namespace URI of the XML Attribute to check.

Usage

Use this method to check whether a namespace qualified XML Attribute of a given name is carried by this XML Element `DomNode` object, where `name` is the name of the XMLAttribute to check; `ns` the namespace URI of the XML Attribute to check. Returns TRUE if an XML Attribute with the given name and namespace URI is carried by this XML Element `DomNode` object, otherwise returns FALSE.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (`STATUS`) on page 3554.

xml.DomNode.hasAttributes
Identifies whether a node has XML Attribute nodes.

Syntax

```java
hasAttributes ()
RETURNS INTEGER
```

Usage

This method returns `TRUE` if this node has XML Attribute nodes; otherwise returns `FALSE`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

```
xml.DomNode.hasChildNodes
```

Returns `TRUE` if a node has child nodes.

Syntax

```java
hasChildNodes ()
RETURNS INTEGER
```

Usage

Use this method to check if a node has child nodes. It returns `TRUE` if this node has child nodes; otherwise, returns `FALSE`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

```
xml.DomNode.insertAfterChild
```

Inserts a `DomNode` object after an existing child `DomNode` object.

Syntax

```java
insertAfterChild (  
    newChild xml.DomNode,  
    refChild xml.DomNode  
)
```

1. `newChild` defines the node to insert.
2. `refChild` defines the reference node (the node before which the new node must be inserted).

Usage

Use this method to inserts a `DomNode` object after an existing child `DomNode` object; `newChild` is the node to insert, `refChild` is the reference node (the node before which the new node must be inserted).

**Important:** This method is not part of W3C standard API.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

```
xml.DomNode.insertBeforeChild
```

Inserts a `DomNode` object before an existing child `DomNode` object in the order of the children list.

Syntax

```java
insertBeforeChild (  
    newChild xml.DomNode,  
    refChild xml.DomNode  
)
```

1. `newChild` defines the node to insert.
2. `refChild` defines the reference node (the node before which the new node must be inserted).

Usage

Use this method to inserts a `DomNode` object before an existing child `DomNode` object; `newChild` is the node to insert, `refChild` is the reference node (the node before which the new node must be inserted).

**Important:** This method is not part of W3C standard API.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.
Inserts a DomNode object before an existing child DomNode object.

**Syntax**

```csharp
insertBeforeChild(
    newNode xml.DomNode,
    refChild xml.DomNode )
```

1. `newChild` defines the node to insert.
2. `refChild` defines the reference node (the node before which the new node must be inserted).

**Usage**

Use this method to insert a DomNode object before an existing child DomNode object; `newChild` is the node to insert, `refChild` is the reference node (the node before which the new node must be inserted).

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**xml.DomNode.isAttached**

Returns whether the node is attached to the XML document.

**Syntax**

```csharp
isAttached ()
RETURNS INTEGER
```

**Usage**

The returned integer indicates whether the node is attached to the XML document or not.

**Important:** This method is not part of W3C standard API.

This method returns TRUE if the DomNode object is attached to a xml.DomDocument object as a child; otherwise it returns FALSE.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**xml.DomNode.isDefaultNamespace**

Checks whether the specified namespace URI is the default namespace.

**Syntax**

```csharp
isDefaultNamespace (  
    ns STRING )
RETURNS INTEGER
```

1. `ns` defines the namespace URI to look for.

**Usage**

Use this method to check whether the specified namespace URI is the default namespace, where `ns` is the namespace URI to look for. Returns TRUE if the given namespace is the default namespace, FALSE otherwise.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.
xml.DomNode.lookupNamespaceURI

Looks up the namespace URI associated to a prefix, starting from a specified node.

**Syntax**

```sql
lookupNamespaceURI(
    prefix STRING
)
RETURNS STRING
```

1. `prefix` defines the prefix to look for.

**Usage**

Use this method to look up the namespace URI associated with a prefix, starting from this node, where `prefix` is the prefix to look for. If NULL, the default namespace URI will be returned. A namespace URI, or NULL, is returned.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomNode.lookupPrefix

Looks up the prefix associated to a namespace URI, starting from the specified node.

**Syntax**

```sql
lookupPrefix(
    ns STRING
)
RETURNS STRING
```

1. `ns` defines the namespace URI to look for.

**Usage**

Use this method to look up the prefix associated with a namespace URI, starting from this node, where `ns` is the namespace URI to look for. Returns the prefix associated with this namespace URI, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomNode.prependChild

Adds a child DomNode object to the beginning of the child list for a DomNode object.

**Syntax**

```sql
prependChild(
    newChild xml.DomNode
)
```

1. `newChild` defines the node to add.

**Usage**

Use this method to add a child DomNode object to the beginning of the child list for this DomNode object; `newChild` is the node to add.

**Important:** This method is not part of W3C standard API.

The DomNode object node must be the child of an element or document node, otherwise the operation fails.
In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**xml.DomNode.prependChildElement**

Creates and adds a child XML Element node to the beginning of the list of child nodes for this XML Element DomNode object.

**Syntax**

```java
prependChildElement (
    name STRING )
RETURNS xml.DomNode
```

1. `name` defines the name of the XML element to add.

**Usage**

Use this method to create and add a child XML Element node to the beginning of the list of child nodes for this XML Element DomNode object; `name` is the name of the XML element to add.

It returns the XML Element DomNode object, or NULL.

**Important:** This method is not part of W3C standard API.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**xml.DomNode.prependChildElementNS**

Creates and adds a child namespace-qualified XML Element node to the beginning of the list of child nodes for an XML Element DomNode object.

**Syntax**

```java
prependChildElementNS ( 
    prefix STRING,  
    name STRING,  
    ns STRING )
RETURNS xml.DomNode
```

1. `prefix` defines the prefix of the XML Element to add.
2. `name` defines the name of the XML Element to add.
3. `ns` defines the namespace URI of the XML Element to add.

**Usage**

Use this method to create and add a child namespace-qualified XML Element node to the beginning of the list of child nodes for this XML Element DomNode object.

It returns the XML Element DomNode object, or NULL.

**Important:** This method is not part of W3C standard API.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**xml.DomNode.removeAllChildren**
Removes all child DomNode objects from a DomNode object.

**Syntax**

```
removeAllChildren()
```

**Usage**

Use this method to remove all child xml.DomNode objects from this DomNode object. In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomNode.removeAttribute

Removes an XML Attribute for an XML Element DomNode object.

**Syntax**

```
removeAttribute( 
    name STRING 
)
```

1. `name` defines the name of the XML attribute to remove.

**Usage**

Use this method to remove an XML Attribute for this XML Element DomNode object, where `name` is the name of the XML attribute to remove. Status is updated with an error code. In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomNode.removeAttributeNS

Removes a namespace qualified XML Attribute for an XML Element DomNode object.

**Syntax**

```
removeAttributeNS( 
    name STRING, 
    ns STRING 
)
```

1. `name` defines the name of the XML Attribute to remove.
2. `ns` defines the namespace URI of the XML Attribute to remove.

**Usage**

Use this method to remove a namespace qualified XML Attribute for this XML Element DomNode object, where `name` is the name and `ns` is the namespace URI of the XML Attribute to remove. In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomNode.removeChild
Removes a child DomNode object from the list of child DomNode objects.

Syntax

```java
removeChild(
    oldChild xml.DomNode
)
```

1. `oldChild` defines the node to remove.

Usage

Use this method to remove a child DomNode object from a list of child DomNode objects, where `oldchild` is the node to remove.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

`xml.DomNode.replaceChild`

Replaces an existing child DomNode with another child DomNode object.

Syntax

```java
replaceChild(
    newChild xml.DomNode,
    oldChild xml.DomNode
)
```

1. `newChild` defines the replacement child.
2. `oldChild` defines the child to be replaced.

Usage

Use this method to replace an existing child DomNode with another child DomNode object, where `oldChild` is the child to be replaced and `newChild` is the replacement child.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

`xml.DomNode.selectByXPath`

Returns a DomNodeList object containing all DomNode objects matching an XPath 1.0 expression.

Syntax

```java
selectByXPath(
    expr STRING,
    args ...)
RETURNS xml.DomNodeList
```

1. `expr` defines the XPath 1.0 expression.
2. `args` defines a list of prefixes bound to namespaces in order to resolve qualified names in the XPath expression.

Usage

This method returns a DomNodeList object containing all DomNode objects matching an XPath 1.0 expression; `expr` is the XPath 1.0 expression, `args` is a list of prefixes bound to namespaces in order to resolve qualified names in the XPath expression. This list must be filled with an even number of arguments, representing the prefix and it corresponding namespace.
**Important:** This method is not part of W3C standard API.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**Example**

```xml
selectByXPath("../../d:Record/*[last()]", "d",
"http://defaultnamespace")
```

```xml
selectByXPath("ns:Record", NULL)
```

```xml
selectByXPath("ns1:Records/ns2:Record",
"ns1",
"http://namespace1",
"ns2",
"http://namespace2")
```

```xml
selectByXPath("ns1:Record", "ns1") is invalid because the namespace definition is missing.
```

If the namespaces list is NULL, the prefixes and namespaces defined in the document itself are used if available.

A namespace must be an absolute URI (ex 'http://', 'file://').

xml.DomNode.setAttribute

Sets (or resets) an XML Attribute for an XML Element DomNode object.

**Syntax**

```xml
setAttribute (  
  name STRING,  
  value STRING )
```

1. `name` defines the name of the XML Attribute.
2. `value` defines the value of the XML Attribute.

**Usage**

Use this method to set (or reset) an XML Attribute for this XML Element DomNode object, where `name` is the name of the XML Attribute and `value` is the value of the XML Attribute.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomNode.setAttributeNode

Sets (or resets) an XML Attribute DomNode object to an XML Element DomNode object.

**Syntax**

```xml
setAttributeNode (  
  attr xml.DomNode  )
```

1. `attr` defines the XML Attribute DomNode object to set.
**Usage**

Use this method to set (or reset) an XML Attribute `DOMNode` object to an XML Element `DOMNode` object, where `attr` is the XML Attribute `DOMNode` object to set.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

`xml.DomNode.setAttributeNodeNS`

Sets (or resets) a namespace-qualified XML Attribute `DOMNode` object to an XML Element `DOMNode` object.

**Syntax**

```java
setAttributeNodeNS (
    attr xml.DomNode
)
```

1. `attr` defines the XML Attribute `DOMNode` object to set.

**Usage**

Use this method to set (or reset) a namespace-qualified XML Attribute `DOMNode` object to an XML Element `DOMNode` object, where `attr` is the XML Attribute `DOMNode` object to set.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

`xml.DomNode.setAttributeNS`

Sets (or resets) a namespace-qualified XML Attribute for an XML Element `DOMNode` object.

**Syntax**

```java
setAttributeNS (
    prefix STRING,
    name STRING,
    ns STRING,
    value STRING
)
```

1. `prefix` defines the prefix of the XML Attribute.
2. `name` defines the name of the XML Attribute.
3. `ns` defines the namespace URI of the XML Attribute.
4. `value` defines the value of the XML Attribute.

**Usage**

Use this method to set (or reset) a namespace-qualified XML Attribute for this XML Element `DOMNode` object, where `prefix` is the prefix of the XML Attribute, `name` is the name of the XML Attribute, `ns` is the namespace URI of the XML Attribute, and `val` is the value of the XML Attribute.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

`xml.DomNode.setIdAttribute`

Set the XML Attribute of given name to be of type ID. Declare (or undeclare) the ID as user-determined.

**Syntax**

```java
setIdAttribute (
)
```
name STRING,
   isId INTEGER )

1. *name* defines the name of the XML Attribute to set.
2. *isId* declares whether the attribute is a user-determined ID attribute.

**Usage**

Use this method to set (or reset) the XML Attribute of given name to be of type ID. Use the value TRUE for the parameter *isId*, to declare the attribute as a user-determined ID attribute, otherwise use FALSE.

This affects the behavior of `getElementById`.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**xml DomNode.setIdAttributeNS**

Set the namespace-qualified XML Attribute of given name and namespace to be of type ID. Declare (or undeclare) the ID as user-determined.

**Syntax**

```
setIdAttributeNS (  
    name STRING,  
    ns STRING,  
    isId INTEGER )
```

1. *name* defines the name of the XML Attribute to set.
2. *ns* defines the namespace URI of the XML Attribute to set.
3. *isId* declares whether the attribute is a user-determined ID attribute.

**Usage**

This method sets the namespace-qualified XML Attribute of given name and namespace to be of type ID. Use the value TRUE for the parameter *isId*, to declare that attribute as a user-determined ID attribute, otherwise use FALSE.

This affects the behavior of `getElementById`.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**xml DomNode.getNodeValue**

Sets the node value for a DomNode object.

**Syntax**

```
getNodeValue (  
    value STRING  
)
```

1. *value* defines the node value.

**Usage**

This method sets the node value for this DomNode object, where *value* is the node value.

Use of this method is only recommended for nodes that are not parents of other nodes, which means it can be used for a node of type:

- ATTRIBUTE_NODE
• TEXT_NODE
• CDATA_SECTION_NODE
• PROCESSING_INSTRUCTION_NODE
• COMMENT_NODE

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DonNode.setPrefix
Sets the prefix for a DomNode object.

**Syntax**

```
setPrefix(
    prefix STRING )
```

1. `prefix` defines the prefix for this DomNode object.

**Usage**

Use this method to set the prefix for this xml.DonNode object.

This method is only valid on namespace qualified Element or Attribute nodes.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DonNode.toString
Returns a string representation of a DomNode object.

**Syntax**

```
toString()
RETURNS STRING
```

**Usage**

This method returns a string representation of this DomNode object, or NULL.

**Important:** This method is not part of W3C standard API.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**DomNode types**
List of types for the xml.DonNode class.

**Table 648: DomNode types**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELEMENT_NODE</td>
<td>The DomNode is an XML Element node.</td>
</tr>
<tr>
<td>ATTRIBUTE_NODE</td>
<td>The DomNode is an XML Attribute node.</td>
</tr>
<tr>
<td>TEXT_NODE</td>
<td>The DomNode is an XML Text node.</td>
</tr>
<tr>
<td>CDATA_SECTION_NODE</td>
<td>The DomNode is an XML CData Section node.</td>
</tr>
</tbody>
</table>
### Type vs. Description

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENTITY_REFERENCE_NODE</td>
<td>The DomNode is an XML Entity Reference node.</td>
</tr>
<tr>
<td>PROCESSING_INSTRUCTION_NODE</td>
<td>The DomNode is an XML Processing Instruction node.</td>
</tr>
<tr>
<td>COMMENT_NODE</td>
<td>The DomNode is an XML Comment node.</td>
</tr>
<tr>
<td>DOCUMENT_TYPE_NODE</td>
<td>The DomNode is an XML DTD node.</td>
</tr>
<tr>
<td>DOCUMENT_FRAGMENT_NODE</td>
<td>The DomNode is an XML Document Fragment node.</td>
</tr>
</tbody>
</table>

#### Examples

**xml.DomNode usage examples.**

**Example Counting the number of nodes in an XML document**

This code example counts the number of nodes of each type.

```xml
IMPORT XML

DEFINE nbElt INTEGER
DEFINE nbAttr INTEGER
DEFINE nbComment INTEGER
DEFINE nbPI INTEGER
DEFINE nbTxt INTEGER
DEFINE nbCData INTEGER

MAIN
  DEFINE document xml.DomDocument
  DEFINE ind INTEGER
  # Handle arguments
  IF num_args() !=1 THEN
    CALL ExitHelp()
  END IF
  # Create document, load it, and count the nodes
  LET document = xml.DomDocument.Create()
  CALL document.load(arg_val(1))
  CALL CountDoc(document)
  # Display result
  DISPLAY "Results: ",nbElt
  DISPLAY " Elements: ",nbAttr
  DISPLAY " Attributes:"nbAttr
  DISPLAY " Comments: ",nbComment
  DISPLAY " PI: ",nbPI
  DISPLAY " Texts: ",nbElt
  DISPLAY " CDat: ",nbCData
END MAIN

FUNCTION CountDoc(d)
  DEFINE d xml.DomDocument
  DEFINE n xml.DomNode
  LET n = d.getFirstDocumentNode()
  WHILE (n IS NOT NULL) |
    CALL Count(n)
    LET n = n.getNextSibling()
  END WHILE
END FUNCTION

FUNCTION Count(n)
  DEFINE n xml.DomNode
  DEFINE child xml.DomNode
  DEFINE next xml.DomNode
  DEFINE node xml.DomNode
  DEFINE ind INTEGER
```
DEFINE name STRING
IF n IS NOT NULL THEN
  IF n.getNodeType() == "COMMENT_NODE" THEN
    LET nbComment = nbComment + 1
  END IF
  IF n.getNodeType() == "ATTRIBUTE_NODE" THEN
    LET nbAttr = nbAttr + 1
  END IF
  IF n.getNodeType() == "PROCESSING_INSTRUCTION_NODE " THEN
    LET nbPI = nbPI + 1
  END IF
  IF n.getNodeType() == "ELEMENT_NODE" THEN
    LET nbElt = nbElt + 1
  END IF
  IF n.getNodeType() == "TEXT_NODE" THEN
    LET nbTxt = nbTxt +1
  END IF
  IF n.getNodeType() == "CDATA_SECTION_NODE" THEN
    LET nbCData = nbCData + 1
  END IF
  IF n.hasChildNodes() THEN
    LET name = n.getLocalName()
    LET child = n.getFirstChild()
    WHILE (child IS NOT NULL )
      CALL Count(child)
      LET child = child.getNextSibling()
    END WHILE
  END IF
  IF n.hasAttributes() THEN
    FOR ind = 1 TO n.getAttributesCount()
      LET node = n.getAttributeNodeItem(ind)
      CALL Count(node)
    END FOR
  END IF
END IF
END FUNCTION

FUNCTION ExitHelp()
  DISPLAY "DomCount <xml>"
  EXIT PROGRAM
END FUNCTION

The DomNodeList class
The xml.DomNodeList class provides methods to manipulate a list of DomNode objects.

You can create a DomNodeList object using selection methods in the DomDocument and DomNode classes. The relationship between the DomNode objects in the list depends on the method used to create the DomNodeList object.

The STATUS variable is set to zero after a successful method call.
xml.DomNodeList methods
Methods for the xml.DomNodeList class.

Table 649: Object methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getCount()</td>
<td>Returns the number of DomNode objects in a DomNodeList object.</td>
</tr>
<tr>
<td>getItem(index)</td>
<td>Returns the DomNode object at a given position in a DomNodeList object.</td>
</tr>
</tbody>
</table>

xml.DomNodeList.getCount
Returns the number of DomNode objects in a DomNodeList object.

Syntax

getCount()  
RETURNS INTEGER

Usage
Use this method to return the number of xml.DomNode objects in this DomNodeList object.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.DomNodeList.getItem
Returns the DomNode object at a given position in a DomNodeList object.

Syntax

getItem(index INTEGER)  
RETURNS xml.DomNode

1. index defines the position of the DomNode object to return (index starts at 1).

Usage
Use this method to return a DomNode object at the given position in this DomNodeList object, where index is the position of the DomNode object to return (Index starts at 1).

Returns NULL when no DomNode object is at the given position.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.
The streaming API for XML (StAX) classes

The streaming API for XML (StAX) classes use streaming while managing XML documents.

The StaxWriter class

The `xml.StaxWriter` class provides methods compatible with Streaming API for XML (StAX) for writing XML documents.

The `STATUS` variable is set to zero after a successful method call.

xml.StaxWriter methods

Methods for the `xml.StaxWriter` class.

Table 650: Class methods: Creation

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xml.StaxWriter.create()</code></td>
<td>Constructor of a StaxWriter object.</td>
</tr>
<tr>
<td>RETURNS <code>xml.StaxWriter</code></td>
<td></td>
</tr>
</tbody>
</table>

Table 651: Object methods: Configuration

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getFeature()</code></td>
<td>Gets a feature of a StaxWriter object.</td>
</tr>
<tr>
<td><code>property STRING</code></td>
<td>RETURNS <code>STRING</code></td>
</tr>
<tr>
<td><code>setFeature()</code></td>
<td>Sets a feature of a StaxWriter object.</td>
</tr>
<tr>
<td><code>property STRING, value STRING</code></td>
<td></td>
</tr>
</tbody>
</table>

Table 652: Object methods: Output

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>close()</code></td>
<td>Closes the StaxWriter streaming, and releases all associated resources.</td>
</tr>
<tr>
<td><code>writeTo(name STRING)</code></td>
<td>Sets the output stream of the StaxWriter object to a file or an URL, and starts the streaming.</td>
</tr>
<tr>
<td><code>writeToPipe(name STRING)</code></td>
<td>Sets the output stream of the StaxWriter object to a PIPE, and starts the streaming.</td>
</tr>
<tr>
<td><code>writeToText(txt TEXT)</code></td>
<td>Sets the output stream of the StaxWriter object to a TEXT large object, and starts the streaming.</td>
</tr>
</tbody>
</table>
Table 653: Object methods: Document

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dtd</code> (</td>
<td>Writes a DTD to the StaxWriter stream.</td>
</tr>
<tr>
<td><code>dtd</code> STRING )</td>
<td></td>
</tr>
<tr>
<td><code>endDocument()</code></td>
<td>Closes any open tags and writes corresponding end tags.</td>
</tr>
<tr>
<td><code>startDocument</code> (</td>
<td>Writes a XML declaration to the StaxWriter stream.</td>
</tr>
<tr>
<td><code>encoding</code> STRING,</td>
<td></td>
</tr>
<tr>
<td><code>version</code> STRING,</td>
<td></td>
</tr>
<tr>
<td><code>standalone</code> INTEGER )</td>
<td></td>
</tr>
</tbody>
</table>

Table 654: Object methods: Namespace

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>declareDefaultNamespace</code> (</td>
<td>Binds a namespace URI to the default namespace, and forces the output to the StaxWriter stream.</td>
</tr>
<tr>
<td><code>ns</code> STRING )</td>
<td></td>
</tr>
<tr>
<td><code>declareNamespace</code> (</td>
<td>Binds a namespace URI to a prefix, and forces the output of the XML namespace definition to the StaxWriter stream.</td>
</tr>
<tr>
<td><code>prefix</code> STRING,</td>
<td></td>
</tr>
<tr>
<td><code>ns</code> STRING )</td>
<td></td>
</tr>
<tr>
<td><code>setDefaultNamespace</code> (</td>
<td>Binds a namespace URI to the default namespace.</td>
</tr>
<tr>
<td><code>ns</code> STRING )</td>
<td></td>
</tr>
<tr>
<td><code>setPrefix</code> (</td>
<td>Binds a namespace URI to a prefix.</td>
</tr>
<tr>
<td><code>prefix</code> STRING,</td>
<td></td>
</tr>
<tr>
<td><code>ns</code> STRING )</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| attribute (  
   name STRING,  
   value STRING ) | Writes a XML attribute to the StaxWriter stream. |
| attributeNS (  
   name STRING,  
   ns STRING,  
   value STRING ) | Writes a XML namespace qualified attribute to the StaxWriter stream. |
| cdata (  
   cdata STRING ) | Writes a XML CData to the StaxWriter stream. |
| characters (  
   characters STRING ) | Writes a XML text to the StaxWriter stream. |
| comment (  
   comment STRING ) | Writes a XML comment to the StaxWriter stream. |
| emptyElement (  
   name STRING ) | Writes an empty XML element to the StaxWriter stream. |
| emptyElementNS (  
   name STRING,  
   ns STRING ) | Writes an empty namespace qualified XML element to the StaxWriter stream. |
| endElement () | Writes an end tag to the StaxWriter stream. |
| entityRef (  
   name STRING ) | Writes a XML EntityReference to the StaxWriter stream. |
| processingInstruction (  
   target STRING,  
   data STRING ) | Writes a XML ProcessingInstruction to the StaxWriter stream. |
| startElement (  
   name STRING ) | Writes a XML start element to the StaxWriter stream. |
| startElementNS (  
   name STRING,  
   ns STRING ) | Writes a namespace-qualified XML start element to the StaxWriter stream. |
Writes a XML attribute to the StaxWriter stream.

Syntax

```java
attribute(
    name STRING,
    value STRING )
```

1. `name` defines the local name of the XML attribute. It cannot be NULL.
2. `value` defines the value of the XML attribute. It cannot be NULL.

Usage

Attributes can only be written on the StaxWriter stream if it points to a START_ELEMENT or an EMPTY_ELEMENT, otherwise the operation fails with an exception. This method can only be called after a startElement, startElementNS, emptyElement, emptyElementNS, or attribute and attributeNS.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxWriter.attributeNS

Writes a XML namespace qualified attribute to the StaxWriter stream.

Syntax

```java
attributeNS(
    name STRING,
    ns STRING,
    value STRING )
```

1. `name` defines the local name of the XML attribute, cannot be NULL.
2. `ns` defines the namespace URI of the XML attribute, cannot be NULL.
3. `value` defines the value of the XML attribute, cannot be NULL.

Usage

Attributes can only be written on the StaxWriter stream if it points to a START_ELEMENT or an EMPTY_ELEMENT, otherwise the operation fails with an exception. This method can only be called after a startElement, startElementNS, emptyElement, emptyElementNS, or attribute and attributeNS.

If namespace URI has not been bound to a prefix with one of the methods setPrefix, declareNamespace, setDefaultNamespace or declareDefaultNamespace, the operation fails with an exception.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxWriter.cdata

Writes a XML CData to the StaxWriter stream.

Syntax

```java
cdata(
    cdata STRING )
```

1. `cdata` defines the data contained in the CData section, or NULL.
Usage
This method writes XML character data passed as parameter as a CData.
In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxWriter.characters
Writes a XML text to the StaxWriter stream.

Syntax

```java
characters (characters STRING )
```

1. `characters` defines the value to write.

Usage
This method writes the character string passed as parameter as a text element.
In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxWriter.close
Closes the StaxWriter streaming, and releases all associated resources.

Syntax

```java
close ()
```

Usage
This method closes the stream.
In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxWriter.comment
Writes a XML comment to the StaxWriter stream.

Syntax

```java
comment (comment STRING )
```

1. `comment` defines the data in the XML comment, or NULL.

Usage
This method writes a XML comment to the stream.
In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxWriter.create
Constructor of a StaxWriter object.

Syntax

```java
xml.StaxWriter.create()
RETURNS xml.StaxWriter
```

Usage

This method returns a StaxWriter object.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxWriter.declareDefaultNamespace

Binds a namespace URI to the default namespace, and forces the output to the StaxWriter stream.

Syntax

```java
declareDefaultNamespace(
   ns STRING
)
```

1. `ns` defines the URI to bind to the default namespace. It cannot be NULL.

Usage

This method binds a namespace URI to the default namespace, and forces the output of the default XML namespace definition to the StaxWriter stream.

The stream must point to a START_ELEMENT, and the prefix scope is the current START_ELEMENT / END_ELEMENT pair.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxWriter.declareNamespace

Binds a namespace URI to a prefix, and forces the output of the XML namespace definition to the StaxWriter stream.

Syntax

```java
declareNamespace(
   prefix STRING,
   ns STRING
)
```

1. `prefix` defines the prefix to bind to the URI, cannot be NULL.
2. `ns` defines the URI to bind to the default namespace, cannot be NULL.

Usage

This method binds a namespace URI to a prefix, and forces the output of the XML namespace definition to the StaxWriter stream.

The stream must point to a START_ELEMENT, and the prefix scope is the current START_ELEMENT / END_ELEMENT pair.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.
xml.StaxWriter.dtd
Writes a DTD to the StaxWriter stream.

Syntax

```
dtd ( 
    dtd STRING )
```

1. `dtd` defines a string representing a valid DTD, cannot be NULL.

Usage

This method writes a document type definition (DTD) for the StaxWriter stream, where `dtd` represents a valid DTD.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxWriter.emptyElement
Writes an empty XML element to the StaxWriter stream.

Syntax

```
emptyElement ( 
    name STRING )
```

1. `name` defines the local name of the XML empty element, cannot be NULL.

Usage

This method writes an empty XML element to the StaxWriter stream, where `name` defines the local name of the empty element. It cannot be NULL.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxWriter.emptyElementNS
Writes an empty namespace qualified XML element to the StaxWriter stream.

Syntax

```
emptyElementNS ( 
    name STRING, 
    ns STRING )
```

1. `name` defines the local name of the XML empty element, cannot be NULL.
2. `ns` defines the namespace URI of the XML empty element, cannot be NULL.

Usage

This method writes an empty namespace qualified XML element to the StaxWriter stream.

If namespace URI has not been bound to a prefix with one of the functions `setPrefix`, `setDefaultNamespace`, or `declareDefaultNamespace`, the operation fails with an exception.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.
xml.StaxWriter.endDocument
Closes any open tags and writes corresponding end tags.

Syntax

```java
endDocument()
```

Usage

This method closes any open tags and writes corresponding ending tags.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxWriter.endElement
Writes an end tag to the StaxWriter stream.

Syntax

```java
endElement()
```

Usage

This method writes an end tag to the StaxWriter stream relying on the internal state to determine the prefix and local name of the last START_ELEMENT.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxWriter.entityRef
Writes a XML EntityReference to the StaxWriter stream.

Syntax

```java
entityRef(
    name  STRING
)
```

1. `name` defines the name of the entity, cannot be NULL.

Usage

This method writes a XML EntityReference, specified by `name`, to the StaxWriter stream.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxWriter.getFeature
Gets a feature of a StaxWriter object.

Syntax

```java
getFeature(
    property  STRING
)
```

1. `property` defines the name of a feature.
Usage

This method returns the name of a feature, specified in property, of the StaxWriter stream.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxWriter.processingInstruction

Writes a XML ProcessingInstruction to the StaxWriter stream

Syntax

```java
processingInstruction(
    target STRING,
    data STRING )
```

1. target defines the target of the Processing Instruction, cannot be NULL.
2. data defines the data of the Processing Instruction, or NULL.

Usage

This method writes a XML ProcessingInstruction to the StaxWriter stream, where target is the target of the Processing Instruction, which cannot be NULL. data is the data of the Processing Instruction, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxWriter.setDefaultNamespace

Binds a namespace URI to the default namespace.

Syntax

```java
setDefaultNamespace(
    ns STRING )
```

1. ns defines the URI to bind to the default namespace, cannot be NULL.

Usage

This method binds a namespace URI to the default namespace. The default namespace scope is the current START_ELEMENT / END_ELEMENT pair; ns is the URI to bind to the default namespace. It cannot be NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxWriter.setFeature

Sets a feature of a StaxWriter object.

Syntax

```java
setFeature(
    property STRING,
    value STRING )
```

1. property defines the name of a feature.
2. value is the value of the feature.
Usage

Use this method to set a feature of a StaxWriter object, where `property` is the name of a feature, and `value` is the value of the feature. The features can be changed at any time, but will only be taken into account at the beginning of a new stream (see `writeTo` or `writeToDocument`).

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

```java
xml.StaxWriter.setPrefix
Binds a namespace URI to a prefix.
```

Syntax

```java
setPrefix(
    prefix STRING,
    ns STRING
)
```

1. `prefix` defines the prefix to be bind to the URI, cannot be NULL.
2. `ns` defines the namespace URI to be bind to the prefix, cannot be NULL.

Usage

Use this method to bind a namespace URI, specified in `ns`, to a prefix defined in `prefix`.

The prefix scope is the current `START_ELEMENT` / `END_ELEMENT` pair.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

```java
xml.StaxWriter.startDocument
Writes a XML declaration to the StaxWriter stream.
```

Syntax

```java
startDocument(
    encoding STRING,
    version STRING,
    standalone INTEGER
)
```

1. `encoding` defines the encoding declaration in the XML declaration. Passing a NULL value will use the default UTF-8 encoding.
2. `version` defines the XML version in the XML declaration. Passing a NULL value will use the default 1.0 version.
3. `standalone` defines the XML standalone declaration. Possible values are:
   - 1: Set `standalone="yes"`.
   - 0: Set `standalone="no"`.
   - -1: Do not set the `standalone` attribute.

Usage

This method writes a XML declaration to the StaxWriter stream to specify the encoding, the version, and whether the document is standalone.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.
**Example**

This call:

```java
startDocument("utf-8","1.0",1)
```

Produces:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
dtd("note [<!ENTITY writer "Donald Duck.">]")
```

**xml.StaxWriter.startElement**

Writes a XML start element to the StaxWriter stream.

**Syntax**

```java
startElement (name STRING )
```

1. *name* defines the local name of the XML start element, cannot be NULL.

**Usage**

This method writes a XML start element to the StaxWriter stream. All `startElement` methods open a new scope and set the stream to a `START_ELEMENT`. Writing the corresponding `endElement` ( ) causes the scope to be closed.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**xml.StaxWriter.startElementNS**

Writes a namespace-qualified XML start element to the StaxWriter stream.

**Syntax**

```java
startElementNS (name STRING, ns STRING )
```

1. *name* defines the local name of the XML start element, cannot be NULL.
2. *ns* defines the namespace URI of the XML start element, cannot be NULL.

**Usage**

All `startElementNS` methods open a new scope and set the stream to a `START_ELEMENT`. Writing the corresponding `endElement` causes the scope to be closed.

If namespace URI has not been bound to a prefix with one of the functions `setPrefix`, `setDefaultNamespace`, or `declareDefaultNamespace`, the operation fails with an exception.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**xml.StaxWriter.writeTo**
Sets the output stream of the StaxWriter object to a file or an URL, and starts the streaming.

Syntax

```java
writeTo(
    name STRING
)
```

1. `name` defines a valid URL or the name of the file that will contain the resulting XML document.

Usage

This method sets the output stream of the StaxWriter object to a file or an URL specified in `name`, and starts the streaming.

Only the following kinds of URLs are supported:

- `http://`
- `https://`
- `tcp://`
- `tcps://`
- `file:///
- `alias://`

See `fglprofile Configuration` for more details about URL mapping with aliases, and for proxy and security configuration.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See `Error handling in GWS calls (STATUS)` on page 3554.

Examples

```java
writeTo("printerList.xml")
writeTo("http://myserver:1100/documents/printerList.xml")
writeTo("https://myserver:1100/documents/printerList.xml")
writeTo("alias://printerlist")
```

In the example `printerlist` alias is defined in `fglprofile` as `ws.printerlist.url = "http://myserver:1100/documents/ptinterList.xml"`.

xml.StaxWriter.writeToDocument

Sets the output stream of the StaxWriter object to an xml.dom.Document object, and starts the streaming.

Syntax

```java
writeToDocument(
    doc xml.dom.Document
)
```

1. `doc` defines the empty `DomDocument` object that will contain the resulting XML document.

Usage

This method sets the output stream of the StaxWriter object to a `DomDocument` specified in `doc`, and starts the streaming.
In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxWriter.writeToPipe
Sets the output stream of the StaxWriter object to a PIPE, and starts the streaming.

**Syntax**

```java
writeToPipe(  
   name STRING )
```

1. `name` defines the command to start the PIPE that will get the resulting XML document.

**Usage**

This method sets the output stream of the StaxWriter object to a PIPE command specified in `name`, and starts the streaming.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxWriter.writeToText
Sets the output stream of the StaxWriter object to a TEXT large object, and starts the streaming.

**Syntax**

```java
writeToText(  
   txt TEXT )
```

1. `txt` defines a TEXT lob located in memory that will contain the resulting XML document.

**Usage**

This method sets the output stream of the StaxWriter object to a text large object in memory, specified in `txt`, and starts the streaming.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**StaxWriter Features**

Features of the `xml.StaxWriter` class.

**Table 656: StaxWriter features**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>format-pretty-print</td>
<td>Formats the output by adding whitespace to produce a pretty-printed, indented, human-readable form. Default value is FALSE.</td>
</tr>
<tr>
<td>smart-ending-elements</td>
<td>Outputs each tag closed with an <code>endElement()</code> call as empty element if it has no children. Default value is FALSE.</td>
</tr>
</tbody>
</table>
Example
This example uses methods from the xml.StaxWriter class.

```plaintext
IMPORT xml

FUNCTION save(file)
    DEFINE file STRING
    DEFINE writer xml.StaxWriter
    TRY
        LET writer = xml.StaxWriter.Create()
        CALL writer.setFeature("format-pretty-print", TRUE)
        CALL writer.writeTo(file)
        CALL writer.startDocument("utf-8","1.0",true)
        CALL writer.comment("This is my first comment using a stax writer")
        CALL writer.setPrefix("c","http://www.mycompany.com/c")
        CALL writer.setPrefix("d","http://www.mycompany.com/d")
        CALL writer.startElementNS("root","http://www.mycompany.com/d")
        CALL writer.attribute("attr1","value1")
        CALL writer.attribute("attr2","value2")
        CALL writer.attributeNS("attr3","http://www.mycompany.com/d","value3")
        CALL writer.comment("This is a comment using a stax writer")
        CALL writer.startElementNS("eltA","http://www.mycompany.com/d")
        CALL writer.CData("<this is a CData section>")
        CALL writer.setDefaultNamespace("http://www.mycompany.com/d")
        CALL writer.startElementNS("eltB","http://www.mycompany.com/c")
        CALL writer.characters("Hello world, I'm from the development team")
        CALL writer.entityRef("one")
        CALL writer.endElement()
        CALL writer.startElementNS("eltA","http://www.mycompany.com/d")
        CALL writer.startElementNS("eltB","http://www.mycompany.com/c")
        CALL writer.characters("Hello world, I'm from the development team")
        CALL writer.startElementNS("eltA","http://www.mycompany.com/d")
        CALL writer.CData("<this is a CData section>")
        CALL writer.setDefaultNamespace("http://www.mycompany.com/d")
        CALL writer.startElementNS("eltB","http://www.mycompany.com/c")
        CALL writer.characters("Hello world, I'm from the development team")
        CALL writer.entityRef("one")
        CALL writer.endElement()
        CALL writer.startElementNS("eltA","http://www.mycompany.com/d")
        CALL writer.CData("<this is a CData section>")
        CALL writer.setDefaultNamespace("http://www.mycompany.com/d")
        CALL writer.startElementNS("eltB","http://www.mycompany.com/c")
        CALL writer.characters("Hello world, I'm from the development team")
        CALL writer.entityRef("one")
        CALL writer.endElement()
        RETURN TRUE
    CATCH
        DISPLAY "StaxWriter ERROR ":,STATUS, SQLCA.SQLERRM
        RETURN FALSE
    END TRY
END FUNCTION
```

The StaxReader class
The StaxReader class provides methods compatible with Streaming API for XML(StAX) for reading XML documents.

The STATUS variable is set to zero after a successful method call.

Syntax

```
xml.StaxReader
```

xml.StaxReader methods
Methods for the xml.StaxReader class.

Table 657: Class methods: Creation

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xml.StaxReader.Create()</td>
<td>Constructor of a StaxReader object.</td>
</tr>
</tbody>
</table>
### Table 658: Object methods: Configuration

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setFeature(property STRING, value STRING)</code></td>
<td>Sets a feature of a StaxReader object.</td>
</tr>
<tr>
<td><code>getFeature(property STRING)</code></td>
<td>Gets a feature of a StaxReader object.</td>
</tr>
</tbody>
</table>

### Table 659: Object methods: Input

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>readFrom(name STRING)</code></td>
<td>Sets the input stream of the StaxReader object to a file or an URL and starts the streaming.</td>
</tr>
<tr>
<td><code>readFromDocument(_doc xml.DomDocument)</code></td>
<td>Sets the input stream of the StaxReader object to a DomDocument object and starts the streaming.</td>
</tr>
<tr>
<td><code>readFromText(_txt TEXT)</code></td>
<td>Sets the input stream of the StaxReader object to a TEXT large object and starts the streaming.</td>
</tr>
<tr>
<td><code>readFromPipe(name STRING)</code></td>
<td>Sets the input stream of the StaxReader object to a PIPE and starts the streaming.</td>
</tr>
<tr>
<td><code>close()</code></td>
<td>Closes the StaxReader streaming and releases all associated resources.</td>
</tr>
</tbody>
</table>
### Table 660: Object methods: Access

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getEventType()</code></td>
<td>Returns a string that indicates the type of event the cursor of the StaxReader object is pointing to.</td>
</tr>
<tr>
<td><code>hasName()</code></td>
<td>Checks whether the StaxReader cursor points to a node with a name.</td>
</tr>
<tr>
<td><code>hasText()</code></td>
<td>Checks whether the StaxReader cursor points to a node with a text value.</td>
</tr>
<tr>
<td><code>isEmptyElement()</code></td>
<td>Checks whether the StaxReader cursor points to an empty element node.</td>
</tr>
<tr>
<td><code>isStartElement()</code></td>
<td>Checks whether the StaxReader cursor points to a start element node.</td>
</tr>
<tr>
<td><code>isEndElement()</code></td>
<td>Checks whether the StaxReader cursor points to an end element node.</td>
</tr>
<tr>
<td><code>isCharacters()</code></td>
<td>Checks whether the StaxReader cursor points to a text node.</td>
</tr>
<tr>
<td><code>isIgnorableWhitespace()</code></td>
<td>Checks whether the StaxReader cursor points to ignorable whitespace.</td>
</tr>
</tbody>
</table>

### Table 661: Object methods: Document

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getEncoding()</code></td>
<td>Returns the document encoding defined in the XML Document declaration, or NULL.</td>
</tr>
<tr>
<td><code>getVersion()</code></td>
<td>Returns the document version defined in the XML Document declaration, or NULL.</td>
</tr>
<tr>
<td><code>isStandalone()</code></td>
<td>Checks whether the document standalone attribute defined in the XML Document declaration is set to yes.</td>
</tr>
<tr>
<td><code>standaloneSet()</code></td>
<td>Checks whether the document standalone attribute is defined in the XML Document declaration.</td>
</tr>
</tbody>
</table>
Table 662: Object methods: Nodes

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getPrefix()</code></td>
<td>Returns the prefix of the current XML node, or NULL.</td>
</tr>
<tr>
<td><code>getLocalName()</code></td>
<td>Returns the local name of the current XML node, or NULL.</td>
</tr>
<tr>
<td><code>getName()</code></td>
<td>Returns the qualified name of the current XML node, or NULL.</td>
</tr>
<tr>
<td><code>getNamespace()</code></td>
<td>Returns the namespace URI of the current XML node, or NULL.</td>
</tr>
<tr>
<td><code>getText()</code></td>
<td>Returns as a string the value of the current XML node, or NULL.</td>
</tr>
</tbody>
</table>

Table 663: Object methods: Processing Instructions

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getPITarget()</code></td>
<td>Returns the target part of a XML Processing Instruction node, or NULL.</td>
</tr>
<tr>
<td><code>getPIData()</code></td>
<td>Returns the data part of a XML Processing Instruction node, or NULL.</td>
</tr>
</tbody>
</table>
### Table 664: Object methods: Attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getAttributeCount()</code></td>
<td>Returns the number of XML attributes defined on the current XML node, or zero.</td>
</tr>
<tr>
<td><code>getAttributeLocalName()</code></td>
<td>Returns the local name of a XML attribute defined at a given position on the current XML node, or NULL.</td>
</tr>
<tr>
<td><code>getAttributeNamespace()</code></td>
<td>Returns the namespace URI of a XML attribute defined at a given position on the current XML node, or NULL.</td>
</tr>
<tr>
<td><code>getAttributePrefix()</code></td>
<td>Returns the prefix of a XML attribute defined at a given position on the current XML node, or NULL.</td>
</tr>
<tr>
<td><code>getAttributeValue()</code></td>
<td>Returns the value of a XML attribute defined at a given position on the current XML node, or NULL.</td>
</tr>
<tr>
<td><code>findAttributeValue()</code></td>
<td>Returns the value of an XML attribute of a given name and/or namespace.</td>
</tr>
</tbody>
</table>
### Table 665: Object methods: Namespace

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lookupNamespace (prefix STRING )</code></td>
<td>LOOKS up the namespace URI associated with a given prefix starting from the current XML node the StaxReader cursor is pointing to.</td>
</tr>
<tr>
<td><code>lookupPrefix (ns STRING )</code></td>
<td>LOOKS up the prefix associated with a given namespace URI, starting from the current XML node the StaxReader cursor is pointing to.</td>
</tr>
<tr>
<td><code>getNamespaceCount ()</code></td>
<td>RETURNS the number of namespace declarations defined on the current XML node, or zero.</td>
</tr>
<tr>
<td><code>getNamespacePrefix (index INTEGER )</code></td>
<td>RETURNS the prefix of a namespace declaration defined at a given position on the current XML node, or NULL.</td>
</tr>
<tr>
<td><code>getNamespaceURI (index INTEGER )</code></td>
<td>RETURNS the URI of a namespace declaration defined at a given position on the current XML node, or NULL.</td>
</tr>
</tbody>
</table>

### Table 666: Object methods: Navigation

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hasNext ()</code></td>
<td>CHECKS whether the StaxReader cursor can be moved to a XML node next to it.</td>
</tr>
<tr>
<td><code>next ()</code></td>
<td>MOVES the StaxReader cursor to the next XML node.</td>
</tr>
<tr>
<td><code>nextTag ()</code></td>
<td>MOVES the StaxReader cursor to the next XML open or end tag</td>
</tr>
<tr>
<td><code>nextSibling ()</code></td>
<td>MOVES the StaxReader cursor to the immediate next sibling XML Element of the current node, skipping all its child nodes.</td>
</tr>
</tbody>
</table>

xml.StaxReader.close
Closes the StaxReader streaming and releases all associated resources.

**Syntax**

```
close()
```

**Usage**

This method closes the stream.
In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxReader.Create

Constructor of a StaxReader object.

**Syntax**

```java
xml.StaxReader.Create()
RETURNS xml.StaxReader
```

**Usage**

Use this method to create and return a StaxReader object.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxReader.findAttributeValue

Returns the value of an XML attribute of a given name and/or namespace.

**Syntax**

```java
findAttributeValue(
    name STRING,
    ns STRING )
RETURNS STRING
```

1. **name** defines the name of the attribute to retrieve. It cannot be NULL.
2. **ns** defines the namespace URI of the attribute to retrieve, or NULL if the attribute is not namespace-qualified.

**Usage**

This method returns the value of an XML attribute of a given name and/or namespace on the current XML node, where **name** is the name of the attribute and **ns** is the namespace or is NULL if not namespace-qualified.

This method is only valid on a START_ELEMENT node.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxReader.getAttributeCount

Returns the number of XML attributes defined on the current XML node, or zero.

**Syntax**

```java
getAttributeCount()
RETURNS INTEGER
```

**Usage**

Use this method to return the number of XML attributes defined on the current XML node.

This method is only valid on a START_ELEMENT node.
In case of error, the method throws an exception and sets the **STATUS** variable. Depending on the error, a human-readable description of the problem is available in the **SQLCA.SQLERRM** register. See **Error handling in GWS calls (STATUS)** on page 3554.

**xml.StaxReader.getAttributeLocalName**

Returns the local name of a XML attribute defined at a given position on the current XML node, or NULL.

**Syntax**

```java
getAttributeLocalName(
    index INTEGER )
RETURNS STRING
```

1. *index* defines the position of the attribute to return (index starts at 1).

**Usage**

Gets the local name of a XML attribute defined at *index* position on the current XML node. It returns the local name or NULL.

This method is only valid on a **START_ELEMENT** node.

In case of error, the method throws an exception and sets the **STATUS** variable. Depending on the error, a human-readable description of the problem is available in the **SQLCA.SQLERRM** register. See **Error handling in GWS calls (STATUS)** on page 3554.

**xml.StaxReader.getAttributeNamespace**

Returns the namespace URI of a XML attribute defined at a given position on the current XML node, or NULL.

**Syntax**

```java
getAttributeNamespace(
    index INTEGER )
RETURNS STRING
```

1. *index* defines the position of the attribute to return (index starts at 1).

**Usage**

This method returns the value of a namespace URI attribute at *index* position on the current XML node, or NULL.

This method is only valid on a **START_ELEMENT** node.

In case of error, the method throws an exception and sets the **STATUS** variable. Depending on the error, a human-readable description of the problem is available in the **SQLCA.SQLERRM** register. See **Error handling in GWS calls (STATUS)** on page 3554.

**xml.StaxReader.getAttributePrefix**

Returns the prefix of a XML attribute defined at a given position on the current XML node, or NULL.

**Syntax**

```java
getAttributePrefix(
    index INTEGER )
RETURNS STRING
```

1. *index* defines the position of the attribute to return (index starts at 1).

**Usage**

This method returns the prefix attribute at *index* position on the current XML node, or NULL.
This method is only valid on a \texttt{START\_ELEMENT} node.

In case of error, the method throws an exception and sets the \texttt{STATUS} variable. Depending on the error, a human-readable description of the problem is available in the \texttt{SQLCA.SQLERRM} register. See Error handling in GWS calls (\texttt{STATUS}) on page 3554.

\texttt{xml.StaxReader.getAttributeValue}

Returns the value of a XML attribute defined at a given position on the current XML node, or NULL.

\textbf{Syntax}

\begin{verbatim}
getAttributeValue(index INTEGER )
RETURNS STRING
\end{verbatim}

1. \textit{index} defines the position of the attribute to return (index starts at 1).

\textbf{Usage}

Use this method to return the value of a XML attribute at \textit{index} position, or NULL.

This method is only valid on a \texttt{START\_ELEMENT} node.

In case of error, the method throws an exception and sets the \texttt{STATUS} variable. Depending on the error, a human-readable description of the problem is available in the \texttt{SQLCA.SQLERRM} register. See Error handling in GWS calls (\texttt{STATUS}) on page 3554.

\texttt{xml.StaxReader.getEncoding}

Returns the document encoding defined in the XML Document declaration, or NULL.

\textbf{Syntax}

\begin{verbatim}
getEncoding()
RETURNS STRING
\end{verbatim}

\textbf{Usage}

This method returns the document encoding as defined in the XML document declaration, or NULL if there is none.

In case of error, the method throws an exception and sets the \texttt{STATUS} variable. Depending on the error, a human-readable description of the problem is available in the \texttt{SQLCA.SQLERRM} register. See Error handling in GWS calls (\texttt{STATUS}) on page 3554.

\texttt{xml.StaxReader.getEventType}

Returns a string that indicates the type of event the cursor of the StaxReader object is pointing to.

\textbf{Syntax}

\begin{verbatim}
getEventType()
RETURNS STRING
\end{verbatim}

\textbf{Usage}

This method returns the name of the event type the StaxReader object cursor is pointing to. See StaxReader Event Types on page 3031 for the full list of StaxReader event types.

In case of error, the method throws an exception and sets the \texttt{STATUS} variable. Depending on the error, a human-readable description of the problem is available in the \texttt{SQLCA.SQLERRM} register. See Error handling in GWS calls (\texttt{STATUS}) on page 3554.
xml.StaxReader.getFeature

Gets a feature of a StaxReader object.

**Syntax**

```java
getFeature (
    property STRING
) 
RETURNS STRING
```

1. `property` defines the name of a feature.

**Usage**

This method returns the name of a feature for the StaxReader object, where `property` is the name of the StaxReader feature.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxReader.getLocalName

Returns the local name of the current XML node, or NULL.

**Syntax**

```java
getLocalName ()
RETURNS STRING
```

**Usage**

Use this method to return the local name of the current XML node, or NULL if there is none.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxReader.getName

Returns the qualified name of the current XML node, or NULL.

**Syntax**

```java
getName ()
RETURNS STRING
```

**Usage**

This method returns the qualified name of the current XML node, or NULL.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxReader.getNamespace

Returns the namespace URI of the current XML node, or NULL.

**Syntax**

```java
getNamespace ()
```
Usage

This method returns the namespace URI of the current XML node, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxReader.getNamespaceCount

Returns the number of namespace declarations defined on the current XML node, or zero.

Syntax

```java
getNamespaceCount ()
RETURNS INTEGER
```

Usage

This method returns the number of namespace declarations in the current XML node, or zero if none.

This method is only valid on a START_ELEMENT node.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxReader.getNamespacePrefix

Returns the prefix of a namespace declaration defined at a given position on the current XML node, or NULL.

Syntax

```java
getNamespacePrefix ( 
 index INTEGER )
RETURNS STRING
```

1. `index` defines the position of the namespace declaration (index starts at 1).

Usage

This method returns the prefix of a namespace declaration at `index` position on the current XML node, or NULL.

This method is only valid on a START_ELEMENT node.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxReader.getNamespaceURI

Returns the URI of a namespace declaration defined at a given position on the current XML node, or NULL.

Syntax

```java
getNamespaceURI ( 
 index INTEGER )
RETURNS STRING
```

1. `index` defines the position of the namespace declaration (index starts at 1).
Usage

This method returns the URI of a namespace declaration defined at index position on the current XML node, or NULL.

This method is only valid on a START_ELEMENT node.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxReader.getPIData
Returns the data part of a XML Processing Instruction node, or NULL.

Syntax

getPIData ()
RETURNS STRING

Usage

Use this method to return the data part of a XML PROCESSING_INSTRUCTION node, or NULL.

This method is only valid on a PROCESSING_INSTRUCTION node.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxReader.getPITarget
Returns the target part of a XML Processing Instruction node, or NULL.

Syntax

getPITarget ()
RETURNS STRING

Usage

This method returns the target part of a XML Processing Instruction node, or NULL.

This method is only valid on a PROCESSING_INSTRUCTION node.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxReader.getPrefix
Returns the prefix of the current XML node, or NULL.

Syntax

getPrefix ()
RETURNS STRING

Usage

This method returns the prefix of the current XML node, or NULL.
In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxReader.getText

Returns as a string the value of the current XML node, or NULL.

Syntax

```
getText()
RETURNS STRING
```

Usage

Use this method to return a string containing text in the current XML node, or NULL if there is none.

This method is only valid on CHARACTERS, CDATA, SPACE, COMMENT, DTD, and ENTITY_REFERENCE nodes. For an ENTITY_REFERENCE, this method returns the replacement value, or NULL if none. See StaxReader Event Types on page 3031.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxReader.getVersion

Returns the document version defined in the XML Document declaration, or NULL.

Syntax

```
getVersion()
RETURNS STRING
```

Usage

This method returns the document version as defined in the XML document declaration, or NULL if not defined.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxReader.hasName

Checks whether the StaxReader cursor points to a node with a name.

Syntax

```
hasName()
RETURNS INTEGER
```

Usage

This method returns TRUE if the current XML node has a name, FALSE otherwise. It returns TRUE for START_ELEMENT and END_ELEMENT, FALSE for all other nodes.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxReader.hasNext
Checks whether the StaxReader cursor can be moved to a XML node next to it.

**Syntax**

```java
hasNext ()
RETURNS INTEGER
```

**Usage**

Use this method to check if there is still a XML node in the stream. It returns TRUE if there is, FALSE otherwise.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxReader.hasText
Checks whether the StaxReader cursor points to a node with a text value.

**Syntax**

```java
hasText ()
RETURNS INTEGER
```

**Usage**

Use this method to check if the StaxReader cursor is pointing to a node with a text value. It returns TRUE if the current XML node has a text value, FALSE otherwise. This method returns TRUE for CHARACTERS, SPACE, CDATA, COMMENT, ENTITY_REFERENCE, and DTD nodes, FALSE for all other nodes.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxReader.isCharacters
Checks whether the StaxReader cursor points to a text node.

**Syntax**

```java
isCharacters ()
RETURNS INTEGER
```

**Usage**

Use this method to check if the StaxReader cursor is pointing to a text node. It returns TRUE if the current XML node is a text node, FALSE otherwise.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxReader.isEmptyElement
Checks whether the StaxReader cursor points to an empty element node.

**Syntax**

```java
isEmptyElement ()
RETURNS INTEGER
```
**Usage**

Use this method to check if the StaxReader cursor is pointing to an XML element node that has no children. It returns TRUE if the node has no children, FALSE otherwise.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxReader.isEndElement

Checks whether the StaxReader cursor points to an end element node.

**Syntax**

```java
isEndElement()
RETURNS INTEGER
```

**Usage**

Use this method to check if the StaxReader cursor is pointing to an end element node. It returns TRUE if the current XML node is an end element node, FALSE otherwise.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxReader.isIgnorableWhitespace

Checks whether the StaxReader cursor points to ignorable whitespace.

**Syntax**

```java
isIgnorableWhitespace()
RETURNS INTEGER
```

**Usage**

Use this method to check if the StaxReader cursor is pointing to a text node with ignorable whitespace. It returns TRUE if there is ignorable whitespace, FALSE otherwise.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxReader.isStandalone

Checks whether the document standalone attribute defined in the XML Document declaration is set to yes.

**Syntax**

```java
isStandalone()
RETURNS INTEGER
```

**Usage**

Use this method to check if the document standalone attribute defined in the XML Document declaration is set to yes. It returns TRUE if set to yes, FALSE otherwise.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.
xml.StaxReader.isStartElement
Checks whether the StaxReader cursor points to a start element node.

**Syntax**

```plaintext
isStartElement()
RETURNS INTEGER
```

**Usage**

Use this method to check if the StaxReader cursor is pointing to a start element node. It returns TRUE if the current XML node is a start element node, FALSE otherwise.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxReader.lookupNamespace
Looks up the namespace URI associated with a given prefix starting from the current XML node the StaxReader cursor is pointing to.

**Syntax**

```plaintext
lookupNamespace(  
  prefix STRING  
)  
RETURNS STRING
```

1. prefix defines the prefix to look for; if NULL the default namespace URI will be returned.

**Usage**

Use this method to return the namespace URI associated with the prefix specified by prefix at the current XML node. It returns a string with the namespace URI associated with the prefix, or NULL if there is none.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxReader.lookupPrefix
Looks up the prefix associated with a given namespace URI, starting from the current XML node the StaxReader cursor is pointing to.

**Syntax**

```plaintext
lookupPrefix(  
  ns STRING  
)  
RETURNS STRING
```

1. ns defines the namespace URI to look for. It cannot be NULL.

**Usage**

Use this method to return the prefix associated with the namespace specified by ns at the current XML node. It returns the prefix, or NULL if there is none.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.
xml.StaxReader.next

Moves the StaxReader cursor to the next XML node.

**Syntax**

```java
next ()
```

**Usage**

Use this method to move the StaxReader cursor to the next XML node in the stream.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxReader.nextSibling

Moves the StaxReader cursor to the immediate next sibling XML Element of the current node, skipping all its child nodes.

**Syntax**

```java
nextSibling ()
```

**Usage**

Use this method to move the StaxReader cursor to the next sibling of the current XML Element node in the stream, skipping all its child nodes. The cursor points to the parent end tag if there are no siblings.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxReader.nextTag

Moves the StaxReader cursor to the next XML open or end tag

**Syntax**

```java
nextTag ()
```

**Usage**

Use this method to move the StaxReader cursor to the next XML open or end tag. The cursor points to the end of the document if there is no next XML open or end tag.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxReader.readFrom

Sets the input stream of the StaxReader object to a file or an URL and starts the streaming

**Syntax**

```java
readFrom( 
    name STRING )
```

1. **name** defines a valid URL or the name of the file to read.
Usage
This method sets the input stream of the StaxReader object to a file or an URL, where name is a valid URL or the name of the file.

Only the following kinds of URLs are supported:

- http://
- https://
- tcp://
- tcps://
- file:////
- alias://

See fglprofile Configuration for more details about URL mapping with aliases, and for proxy and security configuration.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxReader.readFromDocument
Sets the input stream of the StaxReader object to a DomDocument object and starts the streaming.

Syntax

```java
readFromDocument(
    _doc xml.DomDocument
)
```


Usage
Use this method to set the input stream of the StaxReader object to a xml.DomDocument object, where _doc is a valid object containing an XML document.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxReader.readFromPipe
Sets the input stream of the StaxReader object to a PIPE and starts the streaming.

Syntax

```java
readFromPipe(
    name STRING
)
```

1. name defines the command to start the PIPE and where the reader will get the XML from.

Usage
Use this method to set the input stream of the StaxReader object to a PIPE, where name is the command to read from the PIPE.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.StaxReader.readFromText
Sets the input stream of the StaxReader object to a TEXT large object and starts the streaming.

**Syntax**

```java
readFromText (
    _txt TEXT )
```

1. `_txt` defines a TEXT lob located in memory and containing the XML to read.

**Usage**

Use this method to set the input stream of the StaxReader object to a TEXT large object, where `_txt` is the text to read.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (`STATUS`) on page 3554.

**xml.StaxReader.setFeature**

Sets a feature of a StaxReader object.

**Syntax**

```java
setFeature (   
    property STRING,   
    value STRING )
```

1. `property` defines the name of a feature.
2. `value` defines the value of the feature.

**Usage**

Use this method to set a feature for the StaxReader object, where `property` is the name of a `XmlStaxReader` feature, and `value` is the value of a feature.

The features can be changed at any time, but they will only be taken into account at the beginning of a new stream (see `readFrom` or `readFromDocument`).

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (`STATUS`) on page 3554.

**xml.StaxReader.standaloneSet**

Checks whether the document standalone attribute is defined in the XML Document declaration.

**Syntax**

```java
standaloneSet ()
RETURNS INTEGER
```

**Usage**

Use this method to check if the document standalone attribute is defined in the XML Document declaration. It returns `TRUE` if the standalone attribute is set, `FALSE` otherwise.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (`STATUS`) on page 3554.
StaxReader Features
Features of the xml.StaxReader class.

Table 667: StaxReader Features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expand-entity-references</td>
<td>Defines whether XML EntityReference nodes are kept or replaced during the parsing of a XML document. Default value is TRUE.</td>
</tr>
</tbody>
</table>

StaxReader Event Types
Event types of the xml.StaxReader class.

Table 668: StaxReader event types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>XML sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>START_DOCUMENT</td>
<td>StaxReader cursor points to the beginning of the XML document.</td>
<td>&lt;?xml version=&quot;1.0&quot; standalone=&quot;no&quot;?&gt;</td>
</tr>
<tr>
<td>END_DOCUMENT</td>
<td>StaxReader cursor has reached the end of the XML document.</td>
<td></td>
</tr>
<tr>
<td>START_ELEMENT</td>
<td>StaxReader cursor points to a XML start element or empty element node.</td>
<td>&lt;p:elt attr=&quot;val&quot;/&gt; or &lt;p:elt attr=&quot;val&quot;/&gt;</td>
</tr>
<tr>
<td>END_ELEMENT</td>
<td>StaxReader cursor points to a XML end element node.</td>
<td>&lt;/p:elt&gt;</td>
</tr>
<tr>
<td>CHARACTERS</td>
<td>StaxReader cursor points to a XML text node.</td>
<td>... eltA/&gt;This is text&lt;eltB ...</td>
</tr>
<tr>
<td>CDATA</td>
<td>StaxReader cursor points to a XML CDATA node.</td>
<td>&lt;![CDATA[&lt;Hello, world!&gt;]]&gt;</td>
</tr>
<tr>
<td>SPACE</td>
<td>StaxReader cursor points to a XML text node containing only whitespaces.</td>
<td>... eltA/&gt; &lt;eltB ...</td>
</tr>
<tr>
<td>COMMENT</td>
<td>StaxReader cursor points to a XML comment node.</td>
<td>&lt;!-- a comment --&gt;</td>
</tr>
<tr>
<td>DTD</td>
<td>StaxReader cursor points to a DTD string.</td>
<td>&lt;!DOCTYPE A [ &lt;!ELEMENT B (C +)&gt; ]&gt;</td>
</tr>
<tr>
<td>ENTITYREFERENCE</td>
<td>StaxReader cursor points to a XML entity reference node.</td>
<td>&amp;ref;</td>
</tr>
<tr>
<td>PROCESSING_INSTRUCTION</td>
<td>StaxReader cursor points to a XML processing instruction node.</td>
<td>&lt;?target data?&gt;</td>
</tr>
</tbody>
</table>
**Type** | **Description** | **XML sample**
---|---|---
ERROR | StaxReader cursor points to an unexpected XML node. | |

**Example**
Example using methods of the `xml.StaxReader` class.

```
IMPORT xml

FUNCTION parse(file)
DEFINE file STRING
DEFINE event STRING
DEFINE ind INTEGER
DEFINE reader xml.StaxReader
TRY
  LET reader=xml.StaxReader.Create()
  CALL reader.readFrom(file)
  WHILE (true)
    LET event=reader.getEventType()
    CASE event
      WHEN "START_DOCUMENT"
        DISPLAY "Document reading started"
        DISPLAY "XML Version : ",reader.getVersion()
        DISPLAY "XML Encoding : ",reader.getEncoding()
        IF reader.standaloneSet() THEN
          IF reader.isStandalone() THEN
            DISPLAY "Standalone : yes"
          ELSE
            DISPLAY "Standalone : no"
          END IF
        END IF
      WHEN "END_DOCUMENT"
        DISPLAY "Document reading finished"
      WHEN "START_ELEMENT"
        IF reader.isEmptyElement() THEN
          DISPLAY "<"||reader.getName()||"/>
        ELSE
          DISPLAY "<"||reader.getName()||">
          FOR ind=1 TO reader.getNamespaceCount()
            DISPLAY "xmlns:"||reader.getNamespacePrefix(ind)||"=
            ||reader.getNamespaceURI(ind)
          END FOR
          FOR ind=1 TO reader.getAttributeCount()
            IF reader.getAttributePrefix(ind) THEN
              DISPLAY reader.getAttributePrefix(ind)||":"
            ELSE
              DISPLAY reader.getAttributeLocalName(ind)||"=
            END IF
            ||reader.getAttributeValue(ind)
          END FOR
          DISPLAY "</"||reader.getName()||">
      WHEN "CHARACTERS"
        IF reader.hasText() AND NOT reader.isIgnorableWhitespace() THEN
          DISPLAY "CHARACTERS :",reader.getText()
        END IF
      WHEN "COMMENT"
        IF reader.hasText() THEN
          DISPLAY "Comment :",reader.getText()
        END IF
    END CASE
  END WHILE
END TRY
```
XML serialization classes
The XML serialization classes convert BDL variables to XML and XML to BDL variables.

The Serializer class
The xml.Serializer class provides methods to manage options for the serializer engine, and to use the serializer engine to serialize variables and XML element nodes.

This class is a static class and does not have to be instantiated.

The STATUS variable is set to zero after a successful method call.
xml.Serializer methods
Methods for the xml.Serializer class.
Table 669: Class methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xml.Serializer.CreateXmlSchemas</code></td>
<td>Creates XML schemas corresponding to the given variable, and fills a dynamic array with xml.DomDocument objects each representing a XML schema.</td>
</tr>
<tr>
<td><code>xml.Serializer.DomToStax</code></td>
<td>Serializes a XML DomNode object to a StaxWriter object.</td>
</tr>
<tr>
<td><code>xml.Serializer.DomToVariable</code></td>
<td>Serializes a XML element node into a BDL variable using a DomNode object.</td>
</tr>
<tr>
<td><code>xml.Serializer.GetOption</code></td>
<td>Gets a global option value from the serializer engine.</td>
</tr>
<tr>
<td><code>xml.Serializer.OptimizedDomToVariable</code></td>
<td>Serializes a XML element node into a BDL variable using a DomNode object.</td>
</tr>
<tr>
<td><code>xml.Serializer.OptimizedSoapSection5ToVariable</code></td>
<td>Serializes an XML element node into a BDL variable in Soap Section 5 encoding.</td>
</tr>
<tr>
<td><code>xml.Serializer.OptimizedStaxToVariable</code></td>
<td>Serializes an XML element node into a BDL variable using a StaxReader object.</td>
</tr>
<tr>
<td><code>xml.Serializer.OptimizedVariableToDom</code></td>
<td>Serializes a BDL variable into a XML element node using a DomNode object.</td>
</tr>
<tr>
<td><code>xml.Serializer.OptimizedVariableToSoapSection5</code></td>
<td>Serializes a BDL variable into a XML element node in Soap Section 5 encoding.</td>
</tr>
<tr>
<td><code>xml.Serializer.OptimizedVariableToStax</code></td>
<td>Serializes a BDL variable into a XML element node using a StaxWriter object.</td>
</tr>
<tr>
<td><code>xml.Serializer.SetOption</code></td>
<td>Sets a global option value for the serializer engine</td>
</tr>
<tr>
<td><code>xml.Serializer.SoapSection5ToVariable</code></td>
<td>Serializes an XML element node into a BDL variable.</td>
</tr>
<tr>
<td><code>xml.Serializer.StaxToVariable</code></td>
<td>Serializes an XML element node into a BDL variable using a StaxReader object.</td>
</tr>
<tr>
<td><code>xml.Serializer.StaxToDom</code></td>
<td>Serializes an XML element node into a DomNode object using a StaxReader object.</td>
</tr>
<tr>
<td><code>xml.Serializer.VariableToDom</code></td>
<td>Serializes a BDL variable into a XML element node using a DomNode object.</td>
</tr>
<tr>
<td><code>xml.Serializer.VariableToSoapSection5</code></td>
<td>Serializes a BDL variable into a XML element node in Soap Section 5 encoding.</td>
</tr>
<tr>
<td><code>xml.Serializer.VariableToStax</code></td>
<td>Serializes a BDL variable into a XML element node using a StaxWriter object.</td>
</tr>
</tbody>
</table>
xml.Serializer.CreateXmlSchemas

Creates XML schemas corresponding to the given variable, and fills a dynamic array with xml.DomDocument objects each representing a XML schema.

**Syntax**

```plaintext
xml.Serializer.CreateXmlSchemas (  
    var RECORD,  
    schemas RECORD )
```

1. `var` is a given variable.
2. `schemas` is a dynamic array of xml.DomDocument objects, each representing an XML schema.

**Usage**

Use this method to create XML schemas corresponding to the given variable `var`, and fill the dynamic array `schemas` with xml.DomDocument objects each representing a XML schema.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.Serializer.DomToStax

Serializes a XML DomNode object to a StaxWriter object.

**Syntax**

```plaintext
xml.Serializer.DomToStax (  
    node xml.DomNode,  
    stax xml.StaxWriter )
```

1. `node` is a DomNode object.
2. `stax` is a StaxWriter object.

**Usage**

Use this method to serialize a xml.DomNode object to a StaxWriter object.

The resulting XML element node of the serialization process will be added at the current cursor position of the StaxWriter object.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.Serializer.DomToVariable

Serializes a XML element node into a BDL variable using a DomNode object.

**Syntax**

```plaintext
xml.Serializer.DomToVariable (  
    node xml.DomNode,  
    var RECORD )
```

1. `node` is a DomNode object of type ELEMENT_NODE.
2. `var` is any Genero BDL variable with optional XML mapping attributes.
Usage
Use this method to serialize a XML element node defined by `node` as a DomNode object into a BDL variable.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

xml.Serializer.getOption
Gets a global option value from the serializer engine.

Syntax
```
xml.Serializer.GetOption(
    str STRING )
RETURNS STRING
```

1. `str` defines the option flag.

Usage
This method gets an option value from the serializer engine for an option defined by `str`. It returns the value of the flag.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

xml.Serializer.OptimizedDomToVariable
Serializes a XML element node into a BDL variable using a DomNode object.

This API implements the XML-binary Optimized packaging specification. See https://www.w3.org/TR/xop10/.

Syntax
```
xml.Serializer.OptimizedDomToVariable(
    node xml.DomNode,
    var RECORD,
    xopTable RECORD )
```

1. `node` is a `DomNode` object of type `ELEMENT_NODE`.
2. `var` is any Genero BDL variable with optional XML mapping attributes.
3. `xopTable` is a dynamic array, defined as follows:
```
DEFINE XOPTable DYNAMIC ARRAY OF RECORD
    cid STRING, # Content-ID to identify the part in a XML Optimized document
    data BYTE, # Blob handled as part in a XML Optimized document
    file STRING # Name of the file handled as part in an XML Optimized document
END RECORD
```

The `XOPTable` dynamic array is necessary to keep the relation between the data to be handled as separate part in an XML Optimized document via an `href` attribute containing the Content-ID value. This parameter can be NULL.

Usage
In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.
Optimized APIs

Optimized APIs work in the same method as the non-Optimized APIs, with the addition that the optimized API supports XML-binary Optimized format, and return in the XOPTable (if not NULL) the BYTE or the file on disk to handle as a separate part based on the Content-ID.

For instance, if a BYTE has to be sent as an attachment via MTOM, the API will create an XML Optimized node with an href containing the Content-ID returned in the XOP table. This node will include a reference to that BYTE.

For example, given this example of an XML-optimized document:

```xml
<ms:data xmlns:m='http://example.org/stuff'>
  <m:photo>
    <xop:Include xmlns:xop='http://www.w3.org/2004/08/xop/include' href='cid:myref@tempuri.org'/>
  </m:photo>
</m:data>
```

If you have NOT used XMLOptimizedContent, the XOPTable will contain one element where:

- `cid` contains "myref@tempuri.org".
- `data` contains the BYTE to be sent or to be received as an attachment.
- `file` contains NULL.

If you have used XMLOptimizedContent, the XOPtable contains one element where:

- `cid` contains "myref@tempuri.org".
- `data` contains NULL.
- `file` contains the file name to be sent or to be received as an attachment.

xml.Serializer.OptimizedSoapSection5ToVariable

Serializes an XML element node into a BDL variable in Soap Section 5 encoding.

This API implements the XML-binary Optimized packaging specification. See [https://www.w3.org/TR/xop10/](https://www.w3.org/TR/xop10/).

Syntax

```c
xml.Serializer.OptimizedSoapSection5ToVariable(
    node xml.DomNode,
    var RECORD,
    xopTable RECORD)
```

1. `node` is a DomNode object of type ELEMENT_NODE.
2. `var` is any Genero BDL variable with optional XML mapping attributes.
3. `xopTable` is a dynamic array, defined as follows:

   ```c
   DEFINE XOPTable DYNAMIC ARRAY OF RECORD
   cid STRING,  # Content-ID to identify the part in a XML Optimized document
   data BYTE,   # Blob handled as part in a XML Optimized document
   file STRING  # Name of the file handled as part in an XML Optimized document
   END RECORD
   ```

   The XOPTable dynamic array is necessary to keep the relation between the data to be handled as separate part in an XML Optimized document via an href attribute containing the Content-ID value. This parameter can be NULL.
Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

Optimized APIs

Optimized APIs work in the same method as the non-Optimized APIs, with the addition that the optimized API supports XML-binary Optimized format, and return in the XOPTable (if not NULL) the BYTE or the file on disk to handle as a separate part based on the Content-ID.

For instance, if a BYTE has to be sent as an attachment via MTOM, the API will create an XML Optimized node with an href containing the Content-ID returned in the XOP table. This node will include a reference to that BYTE.

For example, given this example of an XML-optimized document:

```xml
<m:data xmlns:m='http://example.org/stuff'>
  <m:photo>
    <xop:Include xmlns:xop='http://www.w3.org/2004/08/xop/include' href='cid:myref@tempuri.org'/>
  </m:photo>
</m:data>
```

If you have NOT used XMLOptimizedContent, the XOPTable will contain one element where:

- `cid` contains "myref@tempuri.org".
- `data` contains the BYTE to be sent or to be received as an attachment.
- `file` contains NULL.

If you have used XMLOptimizedContent, the XOPtable contains one element where:

- `cid` contains "myref@tempuri.org".
- `data` contains NULL.
- `file` contains the file name to be sent or to be received as an attachment.

xml.Serializer.OptimizedStaxToVariable

Serializes an XML element node into a BDL variable using a StaxReader object.

This API implements the XML-binary Optimized packaging specification. See https://www.w3.org/TR/xop10/.

Syntax

```sql
xml.Serializer.OptimizedStaxToVariable (  
  stax xml.StaxReader,  
  var RECORD,  
  xopTable RECORD )
```

1. `stax` is a StaxReader object where the cursor points to an XML Element node.
2. `var` is any Genero BDL variable with optional XML mapping attributes.
3. `xopTable` is a dynamic array, defined as follows:

```
DEFINE XOPTable DYNAMIC ARRAY OF RECORD
  cid STRING, # Content-ID to identify the part in a XML Optimized document
  data BYTE, # Blob handled as part in a XML Optimized document
  file STRING # Name of the file handled as part in an XML Optimized document
END RECORD
```
The XOPTable dynamic array is necessary to keep the relation between the data to be handled as separate part in an XML Optimized document via an href attribute containing the Content-ID value. This parameter can be NULL.

Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

Optimized APIs

Optimized APIs work in the same method as the non-Optimized APIs, with the addition that the optimized API supports XML-binary Optimized format, and return in the XOPTable (if not NULL) the BYTE or the file on disk to handle as a separate part based on the Content-ID.

For instance, if a BYTE has to be sent as an attachment via MTOM, the API will create an XML Optimized node with an href containing the Content-ID returned in the XOP table. This node will include a reference to that BYTE.

For example, given this example of an XML-optimized document:

```xml
<m:photo xmlns:m='http://example.org/stuff'>
  <xop:Include xmlns:xop='http://www.w3.org/2004/08/xop/include'
    href='cid:myref@tempuri.org'/>
</m:photo>
</m:data>
```

If you have NOT used XMLOptimizedContent, the XOPTable will contain one element where:

- `cid` contains "myref@tempuri.org".
- `data` contains the BYTE to be sent or to be received as an attachment.
- `file` contains NULL.

If you have used XMLOptimizedContent, the XOPtable contains one element where:

- `cid` contains "myref@tempuri.org".
- `data` contains NULL.
- `file` contains the file name to be sent or to be received as an attachment.

xml.Serializer.OptimizedVariableToDom

Serializes a BDL variable into a XML element node using a DomNode object.

This API implements the XML-binary Optimized packaging specification. See https://www.w3.org/TR/xop10/.

Syntax

```haskell
xml.Serializer.OptimizedVariableToDom(
  var fgl-type,
  node xml.DomNode,
  xopTable RECORD )
```

1. `var` is any Genero BDL variable with optional XML mapping attributes.
2. `node` is a DomNode object of type ELEMENT_NODE or DOCUMENT_FRAGMENT_NODE.
3. `xopTable` is a dynamic array, defined as follows:

```
DEFINE XOPTable DYNAMIC ARRAY OF RECORD
  cid STRING, # Content-ID to identify the part in a XML Optimized document
  data BYTE, # Blob handled as part in a XML Optimized document
```
The XOPTable dynamic array is necessary to keep the relation between the data to be handled as separate part in an XML Optimize document via an href attribute containing the Content-ID value. This parameter can be NULL.

**Usage**

The resulting XML element node of the serialization process will be added at the current cursor position of the StaxWriter object.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLErrorM register. See Error handling in GWS calls (STATUS) on page 3554.

**Optimized APIs**

Optimized APIs work in the same method as the non-Optimized APIs, with the addition that the optimized API supports XML-binary Optimized format, and return in the XOPTable (if not NULL) the BYTE or the file on disk to handle as a separate part based on the Content-ID.

For instance, if a BYTE has to be sent as an attachment via MTOM, the API will create an XML Optimized node with an href containing the Content-ID returned in the XOP table. This node will include a reference to that BYTE.

For example, given this example of an XML-optimized document:

```xml
<data xmlns:m='http://example.org/stuff'>
  <m:photo>
    <xop:Include xmlns:xop='http://www.w3.org/2004/08/xop/include' href='cid:myref@tempuri.org'/>
  </m:photo>
</data>
```

If you have NOT used XMLOptimizedContent, the XOPTable will contain one element where:

1. cid contains "myref@tempuri.org".
2. data contains the BYTE to be sent or to be received as an attachment.
3. file contains NULL.

If you have used XMLOptimizedContent, the XOPtable contains one element where:

1. cid contains "myref@tempuri.org".
2. data contains NULL.
3. file contains the file name to be sent or to be received as an attachment.

xml.Serializer.OptimizedVariableToSoapSection5

Serializes a BDL variable into a XML element node in Soap Section 5 encoding.

This API implements the XML-binary Optimized packaging specification. See https://www.w3.org/TR/xop10/.

**Syntax**

```bsh
xml.Serializer.OptimizedVariableToSoapSection5( var ,
  node xml.DomNode, 
  xopTable RECORD)
```

1. var is any Genero BDL variable with optional XML mapping attributes.
2. node is a DomNode object of type ELEMENT_NODE or DOCUMENT_FRAGMENT_NODE.
3. **xopTable** is a dynamic array, defined as follows:

```plaintext
DEFINE XOPTable DYNAMIC ARRAY OF RECORD
    cid STRING,  # Content-ID to identify the part in a XML Optimized document
    data BYTE,   # Blob handled as part in a XML Optimized document
    file STRING  # Name of the file handled as part in an XML Optimized document
END RECORD
```

The XOPTable dynamic array is necessary to keep the relation between the data to be handled as separate part in an XML Optimized document via an href attribute containing the Content-ID value. This parameter can be NULL.

**Usage**

The resulting XML element node of the serialization process will be added at the current cursor position of the StaxWriter object.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**Optimized APIs**

Optimized APIs work in the same method as the non-Optimized APIs, with the addition that the optimized API supports XML-binary Optimized format, and return in the XOPTable (if not NULL) the BYTE or the file on disk to handle as a separate part based on the Content-ID.

For instance, if a BYTE has to be sent as an attachment via MTOM, the API will create an XML Optimized node with an href containing the Content-ID returned in the XOP table. This node will include a reference to that BYTE.

For example, given this example of an XML-optimized document:

```xml
<m:data xmlns:m='http://example.org/stuff'>
    <m:photo>
        <xop:Include xmlns:xop='http://www.w3.org/2004/08/xop/include' href='cid:myref@tempuri.org'/>
    </m:photo>
</m:data>
```

If you have NOT used XMLOptimizedContent, the XOPTable will contain one element where:

- **cid** contains "myref@tempuri.org".
- **data** contains the BYTE to be sent or to be received as an attachment.
- **file** contains NULL.

If you have used XMLOptimizedContent, the XOPtable contains one element where:

- **cid** contains "myref@tempuri.org".
- **data** contains NULL.
- **file** contains the file name to be sent or to be received as an attachment.

```plaintext
xml.Serializer.OptimizedVariableToStax
```

Serializes a BDL variable into a XML element node using a StaxWriter object.

This API implements the XML-binary Optimized packaging specification. See [https://www.w3.org/TR/xop10/](https://www.w3.org/TR/xop10/).

**Syntax**

```plaintext
xml.Serializer.OptimizedVariableToStax (var RECORD, stax xml.StaxWriter,
```
1. `var` is any Genero BDL variable with optional XML mapping attributes.
2. `stax` is a StaxWriter object.
3. `xopTable` is a dynamic array, defined as follows:

```gdscript
DEFINE XOPTable DYNAMIC ARRAY OF RECORD
    cid STRING, # Content-ID to identify the part in a XML Optimized document
    data BYTE, # Blob handled as part in a XML Optimized document
    file STRING # Name of the file handled as part in an XML Optimized document
END RECORD
```

The `XOPTable` dynamic array is necessary to keep the relation between the data to be handled as separate part in an XML Optimized document via an href attribute containing the Content-ID value. This parameter can be NULL.

**Usage**

The resulting XML element node of the serialization process will be added at the current cursor position of the StaxWriter object.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**Optimized APIs**

Optimized APIs work in the same method as the non-Optimized APIs, with the addition that the optimized API supports XML-binary Optimized format, and return in the `XOPTable` (if not NULL) the BYTE or the file on disk to handle as a separate part based on the Content-ID.

For instance, if a BYTE has to be sent as an attachment via MTOM, the API will create an XML Optimized node with an href containing the Content-ID returned in the XOP table. This node will include a reference to that BYTE.

For example, given this example of an XML-optimized document:

```xml
<m:data xmlns:m='http://example.org/stuff'>
    <m:photo>
        <xop:Include xmlns:xop='http://www.w3.org/2004/08/xop/include'
                     href='cid:myref@tempuri.org'/>
    </m:photo>
</m:data>
```

If you have NOT used XMLOptimizedContent, the `XOPTable` will contain one element where:

- `cid` contains "myref@tempuri.org".
- `data` contains the BYTE to be sent or to be received as an attachment.
- `file` contains NULL.

If you have used XMLOptimizedContent, the `XOPtable` contains one element where:

- `cid` contains "myref@tempuri.org".
- `data` contains NULL.
- `file` contains the file name to be sent or to be received as an attachment.
Sets a global option value for the serializer engine

**Syntax**

```xml
xml.Serializer.SetOption(
    optionName STRING,
    optionValue STRING )
```

1. `optionName` specifies the name of the option flag.
2. `optionValue` defines the value of the flag.

**Usage**

This method sets the specified option value for the serializer engine for an option defined by `optionName`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**xml.Serializer.SoapSection5ToVariable**

Serializes an XML element node into a BDL variable in Soap Section 5 encoding.

**Syntax**

```xml
xml.Serializer.SoapSection5ToVariable(
    node xml.DomNode,
    var RECORD )
```

1. `node` is a `DomNode` object of type `ELEMENT_NODE`.
2. `var` is any Genero BDL variable with optional XML mapping attributes.

**Usage**

Use this method to serialize a XML element node defined by `node` as a `DomNode` object into a BDL variable in Soap Section 5 encoding.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**xml.Serializer.StaxToDom**

Serializes an XML element node into a `DomNode` object using a StaxReader object.

**Syntax**

```xml
xml.Serializer.StaxToDom(
    stax xml.StaxReader,
    node xml.DomNode )
```

1. `stax` is a `StaxReader` object where the cursor points to an XML Element node.
2. `node` is a `DomNode` object of type `ELEMENT_NODE` or `DOCUMENT_FRAGMENT_NODE`.

**Usage**

This method serializes a XML element node into a `DomNode` object using a StaxReader object. The resulting XML element node of the serialization process will be appended to the last child of the given node.
In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

`xml.Serializer.StaxToVariable`  
Serializes an XML element node into a BDL variable using a StaxReader object.  

**Syntax**  

```java  
xml.Serializer.StaxToVariable(  
    stax xml.StaxReader,  
    var RECORD  
)  
```

1. `stax` is a `StaxReader` object where the cursor points to an XML Element node.  
2. `var` is any Genero BDL variable with optional XML mapping attributes.  

**Usage**  
This method serializes a XML element node into a BDL variable using a StaxReader object. 

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

`xml.Serializer.VariableToDom`  
Serializes a BDL variable into a XML element node using a DomNode object.  

**Syntax**  

```java  
xml.Serializer.VariableToDom(  
    var RECORD,  
    node xml.DomNode  
)  
```

1. `var` is any Genero BDL variable with optional XML mapping attributes.  
2. `node` is a `DomNode` object of type `ELEMENT_NODE` or `DOCUMENT_FRAGMENT_NODE`.  

**Usage**  
This method serializes a BDL variable into a XML element node using a DomNode object. The resulting XML element node of the serialization process will be appended to the last child of the given node.  

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

`xml.Serializer.VariableToSoapSection5`  
Serializes a BDL variable into a XML element node in Soap Section 5 encoding.  

**Syntax**  

```java  
xml.Serializer.VariableToSoapSection5(  
    var fgl-type,  
    node xml.DomNode  
)  
```

1. `var` is any Genero BDL variable with optional XML mapping attributes.  
2. `node` is a `DomNode` object of type `ELEMENT_NODE` or `DOCUMENT_FRAGMENT_NODE`.  


Usage

Use this method to serialize a BDL variable into a XML element node defined by node as a DomNode object using Soap Section 5 encoding. The resulting XML element node of the serialization process will be appended to the last child of the given node.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.Serializer.VariableToStax
Serializes a BDL variable into a XML element node using a StaxWriter object.

Syntax

```plaintext
xml.Serializer.VariableToStax(
    var RECORD,
    stax xml.StaxWriter
)
```

1. `var` is any Genero BDL variable with optional XML mapping attributes.
2. `stax` is a StaxWriter object.

Usage

This method serializes a BDL variable into a XML element node using a StaxReader object.

The resulting XML element node of the serialization process will be added at the current cursor position of the StaxWriter object.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

Serialization option flags
Serialization option flags for the xml.Serializer class.

Table 670: Serialization option flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xml_ignoretimezone</code></td>
<td>Defines that the serializer ignores the time zone information, during the</td>
</tr>
<tr>
<td></td>
<td>marshalling and un-marshalling process of a BDL DATETIME data type.</td>
</tr>
<tr>
<td></td>
<td>A value of zero means FALSE. The default is FALSE.</td>
</tr>
<tr>
<td></td>
<td>Throws an exception in case of errors, and updates status with an error code.</td>
</tr>
<tr>
<td><code>xml_ignoreunknownelements</code></td>
<td>Force the XML serializer to ignore unexpected elements that are not defined in the XML schema. See XML to BDL serialization options on page 3499.</td>
</tr>
<tr>
<td></td>
<td>A value of zero means FALSE. The default is FALSE.</td>
</tr>
<tr>
<td></td>
<td>Throws an exception in case of errors, and updates status with an error code.</td>
</tr>
<tr>
<td><code>xml_ignoreunknownattributes</code></td>
<td>Force the XML serializer to ignore unexpected attributes that are not defined in the XML schema. See XML to BDL serialization options on page 3499.</td>
</tr>
<tr>
<td></td>
<td>A value of zero means FALSE. The default is FALSE.</td>
</tr>
<tr>
<td></td>
<td>Throws an exception in case of errors, and updates status with an error code.</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xml_usetypedefinition</td>
<td>Defines whether the serializer must specify the type of data during serialization. This will add an &quot;xsi:type&quot; attribute to each XML data type.</td>
</tr>
<tr>
<td></td>
<td>A value of zero means FALSE. The default is FALSE.</td>
</tr>
<tr>
<td></td>
<td>Throws an exception in case of errors, and updates status with an error code.</td>
</tr>
<tr>
<td>xml_useutctime</td>
<td>Defines that BDL DATETIME data type is converted to UTC time during the serializer marshalling process.</td>
</tr>
<tr>
<td></td>
<td>A value of zero means FALSE. The default is FALSE.</td>
</tr>
<tr>
<td></td>
<td>Throws an exception in case of errors, and updates status with an error code.</td>
</tr>
<tr>
<td>xop_threshold</td>
<td>When using the optimized serializer APIs, you can set a size that determines whether the BYTE is handled as a separate part (with an XML-Binary Optimized document created for it) or whether it is transmitted inline and encoded in base64 format.</td>
</tr>
<tr>
<td>CALL</td>
<td><strong>xml.serializer.setOption(&quot;xop_threshold&quot;,5000)</strong></td>
</tr>
<tr>
<td></td>
<td>In this example, BYTE variables whose size is greater than 5000 bytes are handled as a separate part, otherwise they are handled inline and encoded in base64 format.</td>
</tr>
<tr>
<td></td>
<td>By default, the size is zero (0), and all BYTE variables are handled as separate parts.</td>
</tr>
<tr>
<td>xs_processcontents</td>
<td>Defines the way to generate wildcard elements and attributes in XML schemas via the XML schema <code>processContents</code> tag. See Table 671: <code>Values for xs_processcontents</code> on page 3047</td>
</tr>
<tr>
<td></td>
<td>Throws an exception in case of errors, and updates status with an error code.</td>
</tr>
</tbody>
</table>

**Table 671: Values for xs_processcontents**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No processContents tag will be generated (default)</td>
</tr>
<tr>
<td>1</td>
<td>Generation of <code>processContents=&quot;skip&quot;</code>.</td>
</tr>
<tr>
<td>2</td>
<td>Generation of <code>processContents=&quot;lax&quot;</code>.</td>
</tr>
<tr>
<td>3</td>
<td>Generation of <code>processContents=&quot;strict&quot;</code>.</td>
</tr>
</tbody>
</table>

**XML security classes**

XML Security classes handle encryption and signature of XML documents entirely in memory with keys and certificates.

**The CryptoKey class**

The `xml.CryptoKey` class provides methods to manipulate HMAC, symmetric and asymmetric keys needed for signing, verifying, encrypting and decrypting XML documents or document fragments.

It follows the XML-Signature and XML-Encryption specifications.

The `STATUS` variable is set to zero after a successful method call.
**xml.CryptoKey methods**  
Methods for the xml.CryptoKey class.

### Table 672: Class methods: Creation

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xml.CryptoKey.Create( url STRING ) RETURNS xml.CryptoKey</td>
<td>Initializes a xml.CryptoKey object. Constructor of an empty CryptoKey object based on a URL.</td>
</tr>
<tr>
<td>xml.CryptoKey.CreateDerivedKey( url STRING ) RETURNS xml.CryptoKey</td>
<td>Constructor of an empty CryptoKey object based on an URL. The crypto key must be derived before use.</td>
</tr>
</tbody>
</table>

### Table 673: Object methods: Access

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>compareTo( toCompare xml.CryptoKey ) RETURNS INTEGER</td>
<td>Compares a CryptoKey object to a second key.</td>
</tr>
<tr>
<td>getSHA1() RETURNS STRING</td>
<td>Returns the SHA1 encoded key identifier in a base64 encoded STRING.</td>
</tr>
<tr>
<td>getSize() RETURNS INTEGER</td>
<td>Returns the size of the key in bits.</td>
</tr>
<tr>
<td>getType() RETURNS STRING</td>
<td>Returns the type of key.</td>
</tr>
<tr>
<td>getUsage() RETURNS STRING</td>
<td>Returns the usage of the key.</td>
</tr>
<tr>
<td>getUrl() RETURNS STRING</td>
<td>Returns the key identifier as an URL.</td>
</tr>
</tbody>
</table>

See also [The Diffie-Hellman key agreement algorithm](#) on page 3318.
Table 674: Object methods: Modify

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>deriveKey</code></td>
<td>Derives the symmetric or HMAC CryptoKey object using the given method identifier and concatenating the optional label, the mandatory seed value and the optional created date as initial random value.</td>
</tr>
<tr>
<td><code>generateKey</code></td>
<td>Generates a random key of given size (in bits).</td>
</tr>
<tr>
<td><code>setKey</code></td>
<td>Defines the value of a HMAC or Symmetric key.</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>computeKey</td>
<td>Computes the shared secret based on the given modulus, generator, the private key, and the other peer's public key. The returned key can be any symmetric/HMAC or symmetric/encryption key type. It can be used for symmetric signature or symmetric encryption.</td>
</tr>
<tr>
<td>loadBIN</td>
<td>Loads a symmetric or HMAC key from a file in raw format.</td>
</tr>
<tr>
<td>loadDER</td>
<td>Loads an asymmetric DSA key, an asymmetric RSA key, or Diffie-Hellman parameters from a file in DER format.</td>
</tr>
<tr>
<td>loadFromString</td>
<td>Loads the given key in BASE64 string format into a CryptoKey object.</td>
</tr>
<tr>
<td>loadPEM</td>
<td>Loads an asymmetric DSA key, an asymmetric RSA key, or Diffie-Hellman parameters from a file in PEM format.</td>
</tr>
<tr>
<td>loadPrivate</td>
<td>Loads the private asymmetric RSA key from the given XML document.</td>
</tr>
<tr>
<td>loadPublic</td>
<td>Loads the public part of an asymmetric RSA or DSA CryptoKey object, or the parameters and the public key of the Diffie-Hellman object from a XML document.</td>
</tr>
<tr>
<td>loadPublicFromString</td>
<td>Populate the current CryptoKey object with the passed public key.</td>
</tr>
<tr>
<td>savePrivate</td>
<td>Saves the private key part of an asymmetric RSA CryptoKey object into a XML document according to the XKMS2.0 specification.</td>
</tr>
<tr>
<td>savePublic</td>
<td>Saves the public part of an asymmetric RSA or DSA CryptoKey object, or the parameters and the public key of the Diffie-Hellman object into a XML document.</td>
</tr>
<tr>
<td>savePublicToString</td>
<td>Save the current xml.CryptoKey's public part in the returned base64 string.</td>
</tr>
<tr>
<td>saveToString</td>
<td>Saves the CryptoKey object into a BASE64 string format.</td>
</tr>
</tbody>
</table>
### Table 676: Object methods: Feature

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getFeature(feature STRING )</code></td>
<td>Returns the value of the given feature for this CryptoKey object, or NULL.</td>
</tr>
<tr>
<td><code>setFeature(feature STRING, value STRING )</code></td>
<td>Sets or resets the value of a feature for a CryptoKey object.</td>
</tr>
</tbody>
</table>

xml.CryptoKey.compareTo

Comparis a CryptoKey object to a second key.

**Syntax**

```plaintext
compareTo(toCompare xml.CryptoKey )
RETURNS INTEGER
```

1. `toCompare` defines the `xml.CryptoKey` object to use for comparison to the current CryptoKey object.

**Usage**

The method verifies if the key's URL, type, size, usage and value are the same. If they are the same, the two identical keys will produce the same encryption cipher.

The key features are not taken into account during comparison.

Returns `TRUE` if they are identical, `FALSE` if they are not identical.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**Related reference**

CryptoKey Features on page 3065
Features of the `xml.CryptoKey` class.

xml.CryptoKey.computeKey

Computes the shared secret based on the given modulus, generator, the private key, and the other peer's public key.

The returned key can be any symmetric/HMAC or symmetric/encryption key type. It can be used for symmetric signature or symmetric encryption.

**Syntax**

```plaintext
computeKey( 
  pub xml.CryptoKey, 
  url STRING 
) 
RETURNS xml.CryptoKey
```

1. `pub` defines the other peer's public key (`xml.CryptoKey`).
2. `url` defines the shared secret key type as an URL identifier (`STRING`).
Usage

Important: This method is for Diffie-Hellman key-agreement algorithm only.

Returns a `xml.CryptoKey sharedSecret`: a `xml.CryptoKey` object of the specified type.

In the 3DES case, no key weakness test is done. If the compound shared secret is weak, the other peer involved in the communication may raise an error. It depends on the language used on the other side.

In order to be able to compute an AES256 shared secret of the Java side, you need to add or replace the files `local_policy.jar` and `US_export_policy.jar` located in `$JDK_HOME/jre/lib/security` by the Java Cryptographic Extension corresponding to your JDK version. You can find this extension at [http://www.oracle.com/technetwork/java/javase/downloads/index.html](http://www.oracle.com/technetwork/java/javase/downloads/index.html).

If the shared secret key length is less than the Diffie-Hellman key length, only the first needed bytes will be taken. For example, if the Diffie-Hellman is 512 bits length and the shared secret is a 3DES key, then only the first 192 bits will be used by the computation. In a 3DES shared secret case, `xml.CryptoKey.computeKey()` is calculated, whereas in AES shared secret case, the Diffie-Hellman key is truncated.

If the shared secret key length is bigger than the Diffie-Hellman key length, an error is raised.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See [Error handling in GWS calls (STATUS)](page 3554).

xml.CryptoKey.Create

Initializes a `xml.CryptoKey` object. Constructor of an empty CryptoKey object based on a URL.

**Syntax**

```java
xml.CryptoKey.Create (url STRING )
RETURNS xml.CryptoKey
```

1. `url` defines a key identifier based on the XML-Signature and XML-Encryption specification or the Diffie-Hellman specification.

**Usage**

Returns a `xml.CryptoKey` object or NULL.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See [Error handling in GWS calls (STATUS)](page 3554).

xml.CryptoKey.CreateDerivedKey

Constructor of an empty CryptoKey object based on an URL. The crypto key must be derived before use.

**Syntax**

```java
xml.CryptoKey.CreateDerivedKey (url STRING )
RETURNS xml.CryptoKey
```

1. `url` defines a key identifier based on the XML-Signature and XML-Encryption specification.

**Usage**

Returns a `xml.CryptoKey` object or NULL. Only symmetric and HMAC keys can be derived.
In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**Related concepts**

`xml.CryptoKey.deriveKey` on page 3053

Derives the symmetric or HMAC CryptoKey object using the given `method` identifier and concatenating the optional `label`, the mandatory `seed` value and the optional `created` date as initial random value.

`xml.CryptoKey.CreateFromNode`  
Constructor of a new CryptoKey object based on a URL, from a XML node based on the XML-Signature and XML-Encryption specification.

**Syntax**

```ruby
define CreateFromNode (  
  url STRING,  
  node xml.DomNode  
)  
RETURNS xml.CryptoKey
```

1. `url` defines a key identifier restricted to PUBLIC/PRIVATE keys.  
2. `node` defines an `ELEMENT` node whose local name is either:  
   - `DSAKeyValue` or `RSAKeyValue` belonging to the XML-Signature namespace `http://www.w3.org/2000/09/xmldsig#`  
   - `RSAKeyPair` belonging to the XKMS 2.0 namespace `http://www.w3.org/2002/03/xkms#`

**Usage**

Returns a CryptoKey object or NULL.

If the local name is RSAKeyValue or RSAKeyPair, the URL must be a RSA key. If the local name is DSAKeyValue, the URL must be a DSA key.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

`xml.CryptoKey.deriveKey`  
Derives the symmetric or HMAC CryptoKey object using the given `method` identifier and concatenating the optional `label`, the mandatory `seed` value and the optional `created` date as initial random value.

**Syntax**

```ruby
define deriveKey (  
  url STRING,  
  label STRING,  
  seed STRING,  
  created STRING,  
  offset INTEGER,  
  bytes INTEGER  
)
```

1. `url` defines the `identifier` of the algorithm to apply to the password and its inputs.  
2. `label` defines the optional label input.  
3. `seed` defines the mandatory seed input as a valid Base64 string representing random binary data obtained with the `security.RandomGenerator.CreateRandomNumber` on page 3127 helper method.  
4. `created` defines the optional created date input.  
5. `offset` defines the number of bytes the resulting octet stream must be shifted to obtain the derived key.  
6. `bytes` defines the number of bytes of the resulting derived key.
**Usage**

If it is a symmetric key, the size can be 0, or must match the original key depending on the identifier of the key type. See Derived keys on page 3065 for more details.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.CryptoKey.generateKey
Generates a random key of given size (in bits).

**Syntax**

```plaintext
generateKey(
    keySize INTEGER
)
```

1. **keySize** defines the size of the key to generate.

**Usage**

For symmetric keys, the size is fixed by the key identifier and cannot be changed. The only authorized values are the real key size or NULL.

For Diffie-Hellman, the input parameter (size INTEGER) is the size of the Diffie-Hellman modulus. If the given size is greater than zero (0), it populates the Diffie-Hellman object by randomly generating a modulus of the given size and a private key, and computes the public key. The used generator is two (2). If the given size is zero (0), it completes the Diffie-Hellman object by choosing a private key and computing the public key based on the previously loaded parameters. For more details on loading parameters, see Table 683: Object methods: Load and save on page 3069.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.CryptoKey.getFeature
Returns the value of the given feature for this CryptoKey object, or NULL.

**Syntax**

```plaintext
getFeature(
    feature STRING
) RETURNS STRING
```

1. **feature** defines the CryptoKey feature.

**Usage**

Returns NULL if the feature is not set.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**Related reference**

CryptoKey Features on page 3065
Features of the xml.CryptoKey class.

xml.CryptoKey.getSHA1
Returns the SHA1 encoded key identifier in a base64 encoded STRING.

**Syntax**

```
getSHA1()
RETURNS STRING
```

**Usage**

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.CryptoKey.getSize

Returns the size of the key in bits.

**Syntax**

```
getSize()
RETURNS INTEGER
```

**Usage**

For a Diffie-Hellman key, it returns the size of the key; the size of a Diffie-Hellman key is actually the size of the modulus. If the modulus is not available (null or equal to zero), the method returns zero. In this situation, a return of zero does NOT mean the key is corrupt or unusable.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**Related reference**

[Supported kind of keys](#) on page 3061
Types of keys supported by the xml.CryptoKey class.

xml.CryptoKey.getType

Returns the type of the key.

**Syntax**

```
getType()
RETURNS STRING
```

**Usage**

This method returns the key type. It corresponds to the key type used by the identifier.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**Related reference**

[Supported kind of keys](#) on page 3061
Types of keys supported by the xml.CryptoKey class.

xml.CryptoKey.getUrl
Returns the key identifier as an URL.

**Syntax**

```
getUrl() 
RETURNS STRING
```

**Usage**

This method returns the cryptographic key as an URL. The key is defined in the XML-Signature and XML-Encryption specification.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**Related reference**

Supported kind of keys on page 3061

Types of keys supported by the `xml.CryptoKey` class.

xml.CryptoKey.getUsage

Returns the usage of the key.

**Syntax**

```
geUsage() 
RETURNS STRING
```

**Usage**

This method returns the usage of the key as defined by the `identifier`.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**Related reference**

Supported kind of keys on page 3061

Types of keys supported by the `xml.CryptoKey` class.

xml.CryptoKey.loadBIN

Loads a symmetric or HMAC key from a file in raw format.

**Syntax**

```
loadBIN( 
    filename STRING )
```

1. `filename` defines the file name or an entry in the FGLPROFILE file.

**Usage**

Raw format means that data in the file is read without any transformation, and will be stored as is in the key.

For instance, if your file contains "hello", it has the same effect as calling `xml.CryptoKey.setKey()` with "hello" as parameter.
In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.CryptoKey.loadDER

Loads an asymmetric DSA key, an asymmetric RSA key, or Diffie-Hellman parameters from a file in DER format.

**Syntax**

```
loadDER(
    filename STRING )
```

1. *filename* defines the file name or an entry in the FGLPROFILE file.

**Usage**

If the DSA or RSA private key or Diffie-Hellman parameters is protected with a password, the recommended way is to unprotect it with the openssl tool and to put the key file on a restricted file system. However, you can use a script or the fglpass agent to provide the password to the application.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.CryptoKey.loadFromString

Loads the given key in BASE64 string format into a CryptoKey object.

**Syntax**

```
loadFromString(
    str STRING )
```

1. *str* defines the string to load.

**Usage**

For Diffie-Hellman, the input parameter is a base64 encoded string containing the Diffie-Hellman parameters. This method populates the Diffie-Hellman key with the modulus and generator in the base64 encoded string. This is useful for the parameters exchange step between two peers.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.CryptoKey.loadPEM

Loads an asymmetric DSA key, an asymmetric RSA key, or Diffie-Hellman parameters from a file in PEM format.

**Syntax**

```
loadPEM(
    filename STRING )
```

1. *filename* defines the file name or an entry in the FGLPROFILE file.

**Usage**

If the DSA or RSA private key or Diffie-Hellman parameters are protected with a password, the recommended way is to unprotect it with the openssl tool and to put the key file on a restricted file system. However, you can use a script or the fglpass agent to provide the password to the application.
In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**xml.CryptoKey.loadPrivate**

Loads the private asymmetric RSA key from the given XML document.

**Syntax**

```
loadPrivate(
    doc xml.DomDocument  )
```

1. `doc` defines a `xml.DomDocument` object.

**Usage**

This method loads the private asymmetric RSA key contained in the given `xml.DomDocument` into the private key element of the CryptoKey object. The RSA private key is based on the **XKMS2.0 specification**.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**xml.CryptoKey.loadPublic**

Loads the public part of an asymmetric RSA or DSA CryptoKey object, or the parameters and the public key of the Diffie-Hellman object from a XML document.

**Syntax**

```
loadPublic(
    doc xml.DomDocument  )
```

1. `doc` defines a `xml.DomDocument` object.

**Usage**

This method populates the RSA or DSA CryptoKey object with the public key parts contained in the given `xml.DomDocument`.

The asymmetric RSA CryptoKey object public key part is based on the XML-Signature specification for **RSA**. The DSA CryptoKey object public key part is based on the **DSA** specification.

For Diffie-Hellman, the input parameter is a `xml.DomDocument` object containing a representation of the Diffie-Hellman key based on the XML-Signature specification for the Diffie-Hellman key values. This method populates the Diffie-Hellman object with the parameters and the public key contained in the given `xml.DomDocument`.

If the public key node exists in the `xml.DomDocument` but is empty, it won't be possible to use the key unless the document contains valid modulus and generator parameters and you call `xml.CryptoKey.generateKey` with a size of zero (0). In this case, you won't be in possession of the other peer's public key.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**xml.CryptoKey.loadPublicFromString**

Populate the current CryptoKey object with the passed public key.

**Syntax**

```
loadPublicFromString(
    key string )
```

1. `key` is a string containing the public key.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.
1. *str* defines the public part of the key in base64 form.

**Usage**

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See [Error handling in GWS calls (STATUS)](page 3554).

```java
xml.CryptoKey.savePublic
Saves the public part of an asymmetric RSA CryptoKey object into a XML document according to the XKMS2.0 specification.
```

**Syntax**

```java
savePublic()
RETURNS xml.DomDocument
```

**Usage**

Returns a xml.DomDocument object containing the private key part of an asymmetric RSA key.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See [Error handling in GWS calls (STATUS)](page 3554).

```java
xml.CryptoKey.savePrivate
Saves the private key part of an asymmetric RSA CryptoKey object into a XML document according to the XKMS2.0 specification.
```

**Syntax**

```java
savePrivate()
RETURNS xml.DomDocument
```

**Usage**

This method saves the public key parts of an RSA or DSA CryptoKey object in an xml.DomDocument.

The asymmetric RSA CryptoKey object public key part is based on the XML-Signature specification for RSA. The DSA CryptoKey object public key part is based on the DSA specification.

For Diffie-Hellman, the method is used for the public key exchanged between the two peers.

See also the CryptoKey RetrievalMethod feature.

Returns a xml.DomDocument object.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See [Error handling in GWS calls (STATUS)](page 3554).

```java
xml.CryptoKey.savePublicToString
Save the current xml.CryptoKey's public part in the returned base64 string.
```

**Syntax**

```java
savePublicToString()
RETURNS STRING
```
**Usage**

Returns the public part of the key in base64 form (STRING).

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

```java
xml.CryptoKey.saveToString
```

Saves the CryptoKey object into a BASE64 string format.

**Syntax**

```java
saveToString()
RETURNS STRING
```

**Usage**

For Diffie-Hellman, this method returns the Diffie-Hellman key's modulus and generator in a base64 encoded string. This is used for the parameters exchange step between the two peers.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

```java
xml.CryptoKey.setFeature
```

Sets or resets the value of a feature for a CryptoKey object.

**Syntax**

```java
setFeature(
    feature STRING,
    value STRING )
```

1. `feature` defines the name of the feature.
2. `value` defines the value to set for the named feature.

**Usage**

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**Related reference**

CryptoKey Features on page 3065
Features of the xml.CryptoKey class.

```java
xml.CryptoKey.setKey
```

Defines the value of a HMAC or Symmetric key.

**Syntax**

```java
setKey(
    key STRING )
```

1. `key` defines the value.
**Usage**

The value can be a password and must be of the size corresponding to the key identifier.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**Supported kind of keys**

Types of keys supported by the `xml.CryptoKey` class.

**Table 677: Supported kind of keys**

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
<th>Usage</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.w3.org/2000/09/xmldsig#dsa-sha1">http://www.w3.org/2000/09/xmldsig#dsa-sha1</a></td>
<td>Asymmetric DSA key with SHA1 for signature purposes.</td>
<td>SIGNATURE</td>
<td>PUBLIC or PRIVATE</td>
</tr>
<tr>
<td></td>
<td>Uses a private DSA key for signature and needs an associated public DSA key or X509 certificate containing it, to verify it.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>See specification for details.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><a href="http://www.w3.org/2000/09/xmldsig#rsa-sha1">http://www.w3.org/2000/09/xmldsig#rsa-sha1</a></td>
<td>Asymmetric RSA key with SHA1 for signature purposes.</td>
<td>SIGNATURE</td>
<td>PUBLIC or PRIVATE</td>
</tr>
<tr>
<td></td>
<td>Uses a private RSA key for signature and needs an associated public RSA key or X509 certificate containing it, to verify it.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>See specification for details.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><a href="http://www.w3.org/2001/04/xmldsig-more#rsa-sha256">http://www.w3.org/2001/04/xmldsig-more#rsa-sha256</a></td>
<td>Asymmetric RSA key with SHA256 for signature purposes.</td>
<td>SIGNATURE</td>
<td>PUBLIC or PRIVATE</td>
</tr>
<tr>
<td></td>
<td>Uses a private RSA key for signature and needs an associated public RSA key or X509 certificate containing it, to verify it.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>See specification for details.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Identifier</td>
<td>Description</td>
<td>Usage</td>
<td>Type</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td><a href="http://www.w3.org/2001/04/xmldsig-more#rsa-sha384">http://www.w3.org/2001/04/xmldsig-more#rsa-sha384</a></td>
<td>Asymmetric RSA key with SHA384 for signature purposes. Uses a private RSA key for signature and needs an associated public RSA key or X509 certificate containing it, to verify it. See specification for details.</td>
<td>SIGNATURE</td>
<td>PUBLIC or PRIVATE</td>
</tr>
<tr>
<td><a href="http://www.w3.org/2001/04/xmldsig-more#rsa-sha512">http://www.w3.org/2001/04/xmldsig-more#rsa-sha512</a></td>
<td>Asymmetric RSA key with SHA512 for signature purposes. Uses a private RSA key for signature and needs an associated public RSA key or X509 certificate containing it, to verify it. See specification for details.</td>
<td>SIGNATURE</td>
<td>PUBLIC or PRIVATE</td>
</tr>
<tr>
<td><a href="http://www.w3.org/2000/09/xmldsig#hmac-sha1">http://www.w3.org/2000/09/xmldsig#hmac-sha1</a></td>
<td>Message Authentication Code key with SHA1 for signature purposes. Uses the same password for signature and to verify it, and key size is free. See specification for details.</td>
<td>SIGNATURE</td>
<td>HMAC</td>
</tr>
<tr>
<td><a href="http://www.w3.org/2001/04/xmldsig-more#hmac-sha256">http://www.w3.org/2001/04/xmldsig-more#hmac-sha256</a></td>
<td>Message Authentication Code key with SHA256 for signature purposes. Uses the same password for signature and to verify it, and key size is free. See specification for details.</td>
<td>SIGNATURE</td>
<td>HMAC</td>
</tr>
<tr>
<td><a href="http://www.w3.org/2001/04/xmldsig-more#hmac-sha384">http://www.w3.org/2001/04/xmldsig-more#hmac-sha384</a></td>
<td>Message Authentication Code key with SHA384 for signature purposes. Uses the same password for signature and to verify it, and key size is free. See specification for details.</td>
<td>SIGNATURE</td>
<td>HMAC</td>
</tr>
<tr>
<td>Identifier</td>
<td>Description</td>
<td>Usage</td>
<td>Type</td>
</tr>
<tr>
<td>---------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td><a href="http://www.w3.org/2001/04/xmldsig-more#hmac-sha512">http://www.w3.org/2001/04/xmldsig-more#hmac-sha512</a></td>
<td>Message Authentication Code key with SHA512 for signature purposes. Uses the same password for signature and to verify it, and key size is free. See specification for details.</td>
<td>SIGNATURE</td>
<td>HMAC</td>
</tr>
<tr>
<td><a href="http://www.w3.org/2001/04/xmlenc#aes128-cbc">http://www.w3.org/2001/04/xmlenc#aes128-cbc</a></td>
<td>Symmetric AES128 key for encryption purposes. Uses a common key of 128bits for encrypting and decrypting XML documents. See specification for details.</td>
<td>ENCRYPTION</td>
<td>SYMMETRIC</td>
</tr>
<tr>
<td><a href="http://www.w3.org/2001/04/xmlenc#aes192-cbc">http://www.w3.org/2001/04/xmlenc#aes192-cbc</a></td>
<td>Symmetric AES192 key for encryption purposes. Uses a common key of 192bits for encrypting and decrypting XML documents. See specification for details.</td>
<td>ENCRYPTION</td>
<td>SYMMETRIC</td>
</tr>
<tr>
<td><a href="http://www.w3.org/2001/04/xmlenc#aes256-cbc">http://www.w3.org/2001/04/xmlenc#aes256-cbc</a></td>
<td>Symmetric AES256 key for encryption purposes. Uses a common key of 256bits for encrypting and decrypting XML documents. See specification for details.</td>
<td>ENCRYPTION</td>
<td>SYMMETRIC</td>
</tr>
<tr>
<td><a href="http://www.w3.org/2001/04/xmlenc#tripledes-cbc">http://www.w3.org/2001/04/xmlenc#tripledes-cbc</a></td>
<td>Symmetric TripleDes key for encryption purposes. Uses a common key of 192bits for encrypting and decrypting XML documents. See specification for details.</td>
<td>ENCRYPTION</td>
<td>SYMMETRIC</td>
</tr>
<tr>
<td>Identifier</td>
<td>Description</td>
<td>Usage</td>
<td>Type</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>----------------------</td>
<td>----------------</td>
</tr>
<tr>
<td><a href="http://www.w3.org/2001/04/xmlenc#kw-aes128">http://www.w3.org/2001/04/xmlenc#kw-aes128</a></td>
<td>Symmetric AES128 key wrap for key encryption purposes.</td>
<td>KEY ENCRYPTION</td>
<td>SYMMETRIC</td>
</tr>
<tr>
<td></td>
<td>Uses a common key of 128bits for encrypting and decrypting a symmetric key.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>See specification for details.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><a href="http://www.w3.org/2001/04/xmlenc#kw-aes192">http://www.w3.org/2001/04/xmlenc#kw-aes192</a></td>
<td>Symmetric AES192 key wrap for key encryption purposes.</td>
<td>KEY ENCRYPTION</td>
<td>SYMMETRIC</td>
</tr>
<tr>
<td></td>
<td>Uses a common key of 192bits for encrypting and decrypting a symmetric key.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>See specification for details.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><a href="http://www.w3.org/2001/04/xmlenc#kw-aes256">http://www.w3.org/2001/04/xmlenc#kw-aes256</a></td>
<td>Symmetric AES256 key wrap for key encryption purposes.</td>
<td>KEY ENCRYPTION</td>
<td>SYMMETRIC</td>
</tr>
<tr>
<td></td>
<td>Uses a common key of 256bits for encrypting and decrypting a symmetric key.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>See specification for details.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><a href="http://www.w3.org/2001/04/xmlenc#kw-tripledes">http://www.w3.org/2001/04/xmlenc#kw-tripledes</a></td>
<td>Symmetric TripleDes key wrap for key encryption purposes.</td>
<td>KEY ENCRYPTION</td>
<td>SYMMETRIC</td>
</tr>
<tr>
<td></td>
<td>Uses a common key of 192bits for encrypting and decrypting a symmetric key.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>See specification for details.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><a href="http://www.w3.org/2001/04/xmlenc#rsa-1_5">http://www.w3.org/2001/04/xmlenc#rsa-1_5</a></td>
<td>Asymmetric RSA key for key encryption purposes.</td>
<td>KEY ENCRYPTION</td>
<td>PUBLIC or PRIVATE</td>
</tr>
<tr>
<td></td>
<td>Uses a public RSA key or a X509 certificate containing it to encrypt a symmetric key, and needs the associated private RSA key to decrypt it.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>See specification for details.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Identifier</td>
<td>Description</td>
<td>Usage</td>
<td>Type</td>
</tr>
<tr>
<td>---------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------</td>
<td>------------------------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td><a href="http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p">http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p</a></td>
<td>Asymmetric RSA key for key encryption purposes. Uses a public RSA key or a X509 certificate containing it to encrypt a symmetric key, and needs the associated private RSA key to decrypt it.</td>
<td>KEY ENCRYPTION</td>
<td>PUBLIC or PRIVATE</td>
</tr>
<tr>
<td>Diffie-Hellman identifier: <a href="http://www.w3.org/2001/04/xmlenc#DHKeyValue">http://www.w3.org/2001/04/xmlenc#DHKeyValue</a></td>
<td>Diffie-Hellman key agreement algorithm. Derives a shared secret. The resulting shared secret is a HMAC or symmetric key for encryption purposes.</td>
<td>KEY AGREEMENT</td>
<td>PUBLIC or PRIVATE</td>
</tr>
</tbody>
</table>

**Derived keys**

Key derivation is used on symmetric or HMAC keys to avoid the direct usage of a shared secret password in secured operations. If two parties share a secret password that is successfully hacked by a third party, any future operations become insecure, and the initial two parties do not even realize that their exchanges are unsafe. However, if a different password based on that shared secret password is used for each new secured operation, even if one operation is compromised, it will only be insecure for that operation, but not other operations.

The derivation consists of applying an algorithm with some additional inputs (such as a random seed value) to a password in order to obtain another password that is then used in one secured operation. Of course, the algorithm and its additional inputs must also be shared to enable the computation of the same derived key for the decryption of the message by the person it is intended for.

Note that passwords are often only composed of alphanumeric characters, which makes the job of a hacker a little bit easier, whereas a derived key is composed of any binary data produced by the algorithm used for the derivation.

**Table 678: Derived keys methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://schemas.xmlsoap.org/ws/2005/02/sc/dk/p_sha1">http://schemas.xmlsoap.org/ws/2005/02/sc/dk/p_sha1</a></td>
<td>Only algorithm supported. See specification for details.</td>
</tr>
</tbody>
</table>

**CryptoKey Features**

Features of the xml.CryptoKey class.

**Table 679: CryptoKey Features**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KeyName</td>
<td>Defines or returns whether a user-defined key name is added during a XML signature or encryption in order to identify it to other applications, or by the xml.key store. The default value is NULL, meaning that no key name is used.</td>
</tr>
<tr>
<td>See W3C KeyName specification for details.</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>KeyValue</strong></td>
<td>Defines or returns whether the public part of the asymmetric key is added during a XML signature or encryption. Only for RSA and DSA keys. The default value is FALSE, meaning that no key value is used.</td>
</tr>
</tbody>
</table>
| **RetrievalMethod** | Defines or returns the URL where the XML form of:  
- a DSA or RSA public key will be set during a XML signature, and loaded during a XML verification process.  
- a RSA public key will be set and used to encrypt a XML node during XML encryption  
- a symmetric key with encryption usage will be used to encrypt a XML node or decrypt it back  
The default value is NULL, meaning that no retrieval method is used.  
The XML form of a DSA or RSA public key can be obtained by the `xml.CryptoKey.savePublic` method.  
The XML form of a symmetric key can be obtained by the `xml.Encryption.encryptKey` method. |

### Examples

**xml.CryptoKey usage examples.**

**Loading an asymmetric RSA key**

```xml
IMPORT xml

MAIN
DEFINE key xml.CryptoKey
LET key = xml.CryptoKey.Create("http://www.w3.org/2001/04/xmlenc#rsa-1_5")
TRY
CALL key.loadPEM("RSA1024Key.pem")
CALL key.setFeature("KeyName","MyRsaKey")
DISPLAY "Key size (in bits) : ",key.getSize() # displays 1024 (bits)
DISPLAY "Key type : ",key.getType() # displays PRIVATE or PUBLIC
DISPLAY "Key usage : ",key.getUsage() # displays KEYENCRYPTION
CATCH
DISPLAY "Unable to load key :",STATUS
END TRY
END MAIN
```

**Note:** All keys in PEM or DER format were created with the OpenSSL tool.

**Generating a symmetric AES256 key**

```xml
IMPORT xml

MAIN
DEFINE key xml.CryptoKey
LET key = xml.CryptoKey.Create("http://www.w3.org/2001/04/xmlenc#aes256-cbc")
```
TRY
    CALL key.generateKey(NULL)
    DISPLAY "Key size (in bits) : ",key.getSize() # displays 256 (bits)
    DISPLAY "Key type : ",key.getType() # displays SYMMETRIC
    DISPLAY "Key usage : ",key.getUsage() # displays ENCRYPTION
CATCH
    DISPLAY "Unable to generate key :",STATUS
END TRY
END MAIN

Note: All keys in PEM or DER format were created with the OpenSSL tool.

Setting a HMAC key

IMPORT xml

MAIN
    DEFINE key xml.CryptoKey
    LET key = xml.CryptoKey.Create("http://www.w3.org/2000/09/xmldsig#hmac-sha1")
    TRY
        CALL key.setKey("secretpassword")
        # displays 112 (size of secretpassword in bits)
        DISPLAY "Key size (in bits) : ",key.getSize()
        DISPLAY "Key type : ",key.getType() # displays HMAC
        DISPLAY "Key usage : ",key.getUsage() # displays SIGNATURE
CATCH
    END TRY
END MAIN

Note: All keys in PEM or DER format were created with the OpenSSL tool.

Deriving a HMAC key

IMPORT xml
IMPORT security

MAIN
    DEFINE key xml.CryptoKey
    # will contain a random binary data encoded in Base64
    DEFINE seedBase64 STRING
    LET key = xml.CryptoKey.CreateDerivedKey("http://www.w3.org/2000/09/xmldsig#hmac-sha1")
    TRY
        # Creates a random 24 bytes long binary data encoded into a Base64 form string
        CALL key.setKey("secretpassword")
        # Derives the 14 bytes long "secretpassword" into a 64 bytes long key
        # from a random 24 bytes long seed value and shifting the resulting key
        # from 255 bytes
        LET seedBase64 = security.RandomGenerator.CreateRandomString(24)
        CALL key.deriveKey("http://schemas.xmlsoap.org/ws/2005/02/sc/dk/p_sha1",
        NULL,seedBase64,NULL,255,64)
        # Displays 512 (size of 'secretpassword' derivation in bits)
        DISPLAY "Key size (in bits) : ",key.getSize()
        # Note: Key is derived and can be used in
        # any encryption or signature function
CATCH
    END TRY
END MAIN
Note: All keys in PEM or DER format were created with the OpenSSL tool.

Computing the shared secret with Diffie-Hellman

Load the Diffie-Hellman parameters from a PEM file, the other peer's public key from an XML file, and compute the shared secret.

Function `generateKey` is called with a 0, parameters are already filled.

```
IMPORT xml

FUNCTION BuildSharedSecret(DHdoc)
    DEFINE myKey, othersPubKey, sharedSecret xml.CryptoKey
    DEFINE DHdoc xml.DomDocument
    LET myKey =
        xml.CryptoKey.Create("http://www.w3.org/2001/04/xmlenc#DHKeyValue")
    LET othersPubKey =
        xml.CryptoKey.Create("http://www.w3.org/2001/04/xmlenc#DHKeyValue")
    TRY
        CALL othersPubKey.loadPublic(DHdoc)
        # populate myKey with the parameters previously generated by the
        # other peer.
        CALL myKey.loadPEM("DHParam.pem")
        # Randomly generate a private key and compute the public key. Key
        # length is the parameters length.
        CALL myKey.generateKey(0)
        LET sharedSecret = myKey.computeKey(othersPubKey,
            "http://www.w3.org/2000/09/xmldsig#hmac-sha1")
    CATCH
        DISPLAY "ERROR : should not raise exception"
        EXIT PROGRAM (-1)
    END TRY
END FUNCTION
```

The CryptoX509 class

The `xml.CryptoX509` class provides methods to manipulate X509 certificates needed for identification of individual persons, groups or any entities during XML encryption or signature process.

It also provides additional load and save functions to interact with other applications in XML or in BASE64, such as in WS-Security compliant applications. It follows the `XML-Signature` and `XML-Encryption` specifications.

The `STATUS` variable is set to zero after a successful method call.

**xml.CryptoX509 methods**

Methods for the `xml.CryptoX509` class.

**Table 680: Class methods: Creation**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 681: Object methods: Access</td>
<td></td>
</tr>
<tr>
<td>----------------------------------</td>
<td></td>
</tr>
<tr>
<td><strong>Name</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><code>getIdentifier()</code></td>
<td>GETS the identification part of a X509 certificate</td>
</tr>
<tr>
<td><code>getThumbprintSHA1()</code></td>
<td>GETS the SHA1 encoded thumbprint identifying the X509 certificate.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 682: Object methods: Modify</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name</strong></td>
</tr>
<tr>
<td><code>createPublicKey(url STRING)</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 683: Object methods: Load and save</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name</strong></td>
</tr>
<tr>
<td><code>load(doc xml.DomDocument)</code></td>
</tr>
<tr>
<td><code>loadDER(filename STRING)</code></td>
</tr>
<tr>
<td><code>loadFromString(str STRING)</code></td>
</tr>
<tr>
<td><code>loadPEM(filename STRING)</code></td>
</tr>
<tr>
<td><code>save()</code></td>
</tr>
<tr>
<td><code>saveToString()</code></td>
</tr>
</tbody>
</table>
### Table 684: Object methods: Feature

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getFeature</code></td>
<td>Get the value of a given feature of a CryptoX509 object.</td>
</tr>
<tr>
<td><code>setFeature</code></td>
<td>Sets or resets the given feature for this CryptoX509 object.</td>
</tr>
</tbody>
</table>

xml.CryptoX509.Create

Constructor of an empty CryptoX509 object.

**Syntax**

```java
xml.CryptoX509.Create() 
RETURNS xml.CryptoX509
```

**Usage**

Returns a xml.CryptoX509 object or NULL.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

xml.CryptoX509.CreateFromNode

Constructor of a new CryptoX509 object from a XML X509 certificate node.

**Syntax**

```java
xml.CryptoX509.CreateFromNode( 
   node xml.DomNode  ) 
RETURNS xml.CryptoX509
```

1. `node` defines an element in xml.DomNode node with `X509Data` as local name, which is based on the XML-Signature specification namespace "http://www.w3.org/2000/09/xmldsig#".

**Usage**

Returns a xml.CryptoX509 object or NULL.

If the X509 certificate is incomplete, the certificate will be created from the application global certificate list if one of `SubjectName` or `Issuer` matches. (See xml.KeyStore.addCertificate for more details.)

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

xml.CryptoX509.createPublicKey

Creates a new public CryptoKey object for the given URL.

**Syntax**

```java
createPublicKey(
```
url STRING )
RETURNS xml.CryptoX509

1. url defines the given url.

Usage
This method creates a new public CryptoKey object for the given URL, from the public key embedded in this certificate if any; NULL otherwise.
Returns a CryptoX509 object.
In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.CryptoX509.getFeature
Get the value of a given feature of a CryptoX509 object.

Syntax
getFeature ( feature STRING )
RETURNS STRING

1. feature defines a feature of the CryptoX509 object.

Usage
This method returns the value of the given feature for the CryptoX509 object, or NULL if feature is not set.
In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

Related reference
CryptoX509 Features on page 3074
Features of the xml.CryptoX509 class.

xml.CryptoX509.getIdentifier
Gets the identification part of a X509 certificate

Syntax
getIdentifier ()
RETURNS STRING

Usage
Returns the identification part of the X509 certificate in a string.
Example: /C=FR/ST=France/L=Schiltigheim/O=MC/OU=My Company Name/CN=cert
In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.CryptoX509.getThumbprintSHA1
Gets the SHA1 encoded thumbprint identifying the X509 certificate.

**Syntax**

```java
getThumbprintSHA1()
RETURNS STRING
```

**Usage**

This method returns the SHA1 encoded thumbprint identifying the X509 certificate in a BASE64-encoded string.

Example: CM4y6z7zzLmTGMelIE46RKIKAPI=

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

---

**xml.CryptoX509.load**

Loads the given XML document with ds:X509Data as root node in a CryptoX509 object.

**Syntax**

```java
load(
    doc xml.DomDocument
)
```

1. `doc` defines a `xml.DomDocument` object.

**Usage**

This method loads an XML document with ds:X509Data based on the XML-Signature specification as root node in a CryptoX509 object. See the w3.org site for more information on the XML-Signature specification for ds:X509Data as root node.

If the X509 certificate is incomplete, the certificate will be created from the application global certificate list if one of SubjectName or Issuer matches. (See `xml.KeyStore.addCertificate` for more details.)

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

---

**xml.CryptoX509.loadDER**

Loads a X509 certificate from a file in DER format.

**Syntax**

```java
loadDER(
    filename STRING
)
```

1. `filename` defines the file name or an entry in the FGLPROFILE file.

**Usage**

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

---

**xml.CryptoX509.loadFromString**
Loads the given X509 certificate in BASE64 string format into this CryptoX509 object.

**Syntax**

```java
loadFromString(
    str STRING )
```

1. `str` defines the X509 certificate in BASE64 string format to load.

**Usage**

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

xml.CryptoX509.loadPEM

Loads a X509 certificate from a file in PEM format.

**Syntax**

```java
loadPEM(
    filename STRING )
```

1. `filename` defines the file name or an entry in the FGLPROFILE file.

**Usage**

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

xml.CryptoX509.save

Saves the CryptoX509 certificate into a XML document with ds:X509Data element as root node.

**Syntax**

```java
save()
RETURNS xml.DomDocument
```

**Usage**

This method saves a CryptoX509 certificate in an XML document with the ds:X509Data element as root node. See the w3.org site for more information on the XML-Signature specification for ds:X509Data as root node.

(See also the RetrievalMethod feature)

Returns a `xml.DomDocument` object.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

xml.CryptoX509.saveToString

Saves the CryptoX509 certificate into a BASE64 string format.

**Syntax**

```java
saveToString()
RETURNS STRING
```
Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.CryptoX509.setFeature

Sets or resets the given feature for this CryptoX509 object.

Syntax

```java
setFeature(
    feature STRING,
    value STRING
)
```

1. `feature` defines the feature to be set.
2. `value` defines the value to set.

Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

Related reference

CryptoX509 Features on page 3074
Features of the xml.CryptoX509 class.

CryptoX509 Features
Features of the xml.CryptoX509 class.

Table 685: CryptoX509 Features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X509Certificate</td>
<td>Defines or returns whether the complete X509 certificate is added during XML signature or encryption. Default value is FALSE.</td>
</tr>
<tr>
<td>See specification for details.</td>
<td></td>
</tr>
<tr>
<td>X509SubjectName</td>
<td>Defines or returns whether the subject name of the X509 certificate is added during XML signature or encryption. Default value is FALSE.</td>
</tr>
<tr>
<td>See specification for details.</td>
<td></td>
</tr>
<tr>
<td>X509IssuerSerial</td>
<td>Defines or returns whether the issuer name and serial number of the X509 certificate is added during XML signature or encryption. Default value is FALSE.</td>
</tr>
<tr>
<td>See specification for details.</td>
<td></td>
</tr>
</tbody>
</table>
### Feature Description

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
</table>
| RetrievalMethod | Defines or returns the URL where the XML form of the X509 certificate will be set during a XML signature, and loaded during a XML verification process, and based on that CryptoX509 object. Default value is NULL, meaning that no retrieval method is used.

**Note:** The XML form of a X509 certificate can be obtain by the `xml.CryptoX509.save()` method.

#### Examples

**xml.CryptoX509 usage examples.**

**Loading a certificate from a PEM file**

```plaintext
IMPORT xml

MAIN
  DEFINE x509 xml.CryptoX509
  LET x509 = xml.CryptoX509.Create()
  TRY
    CALL x509.loadPEM("Certificate.crt");
    DISPLAY "Id : ", x509.getIdentifier()
  CATCH
    DISPLAY "Unable to load certificate : ", STATUS
  END TRY
END MAIN

**Note:** All certificates in PEM format were created with the OpenSSL tool.

**Creating a public key for signature verification from a certificate**

```plaintext
IMPORT xml

MAIN
  DEFINE x509 xml.CryptoX509
  DEFINE key xml.CryptoKey
  LET x509 = xml.CryptoX509.Create()
  TRY
    CALL x509.loadPEM("RSA1024Certificate.crt");
    CATCH
      DISPLAY "Unable to load certificate : ", STATUS
      EXIT PROGRAM
  END TRY
  TRY
    LET key = x509.createPublicKey("http://www.w3.org/2000/09/xmldsig#rsa-shal")
    DISPLAY "Key size (in bytes) : ", key.getSize() # displays 1024 (bits)
    DISPLAY "Key type : ", key.getType() # displays PUBLIC
    DISPLAY "Key usage : ", key.getUsage() # displays SIGNATURE
    CATCH
      DISPLAY "Unable to create public key : ", STATUS
  END TRY
END MAIN

**Note:** All certificates in PEM format were created with the OpenSSL tool.

**Saving the subjectName of a certificate in XML**

```plaintext
IMPORT xml
```
MAIN
DEFINE x509 xml.CryptoX509
DEFINE doc xml.DomDocument
LET x509 = xml.CryptoX509.Create()
TRY
  CALL x509.loadPEM("RSA1024Certificate.crt");
  CATCH
    DISPLAY "Unable to load certificate :",STATUS
    EXIT PROGRAM
END TRY
TRY
  CALL x509.setFeature("X509SubjectName",TRUE)
  LET doc = x509.save()
  CALL doc.setFeature("format-pretty-print",TRUE)
  CALL doc.save("RSAX509SubjectName.xml")
  CATCH
    DISPLAY "Unable to save certificate :",STATUS
END TRY
END MAIN

Note: All certificates in PEM format were created with the OpenSSL tool.

The Signature class
The xml.Signature class provides methods to create detached, enveloped or enveloping XML signatures of one or more references of XML documents or document fragments, and to determine whether a signed referenced document has been modified afterwards.

It follows the XML-Signature specifications.

The STATUS variable is set to zero after a successful method call.

xml.Signature methods
Methods for the xml.Signature class.

Table 686: Class methods: Creation

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xml.Signature.Create()</td>
<td>Constructor of a blank Signature object.</td>
</tr>
<tr>
<td>RETURNS xml.Signature</td>
<td></td>
</tr>
<tr>
<td>xml.Signature.CreateFromNode( node xml.DomNode )</td>
<td>Constructor of a new Signature object from a XML Signature node, based on the XML-Signature specification.</td>
</tr>
<tr>
<td>RETURNS xml.Signature</td>
<td></td>
</tr>
</tbody>
</table>

Table 687: Class methods: Object access

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xml.Signature.RetrieveObjectDataListFromSignatureNode( signNode xml.DomNode, index INTEGER )</td>
<td>Returns a DomNodeList containing all embedded XML nodes related to the signature object</td>
</tr>
<tr>
<td>RETURNS xml.DomNodeList</td>
<td></td>
</tr>
</tbody>
</table>

Note: In addition to this class method categorized under Object Access, there are also object methods. These are listed in Table 694: Object methods: Object access on page 3079.
### Table 688: Object methods: Key and certificate

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setCertificate</code></td>
<td>Defines the X509 certificate to be added to the signature object when signing a document.</td>
</tr>
<tr>
<td><code>setKey</code></td>
<td>Defines the key used for signing or validation.</td>
</tr>
</tbody>
</table>

#### Example Code:
```java
setCertificate(cert xml.CryptoX509)
setKey(key xml.CryptoKey)
```

### Table 689: Object methods: Modifier

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setCanonicalization</code></td>
<td>Sets the canonicalization method to use for the signature.</td>
</tr>
<tr>
<td><code>setId</code></td>
<td>Sets an ID value for the signature.</td>
</tr>
</tbody>
</table>

#### Example Code:
```java
setCanonicalization(url STRING)
setId(id STRING)
```

### Table 690: Object methods: Access

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getCanonicalization</code></td>
<td>Returns the canonicalization identifier of the signature.</td>
</tr>
<tr>
<td><code>getDocument</code></td>
<td>Returns a new DomDocument object representing the signature in XML.</td>
</tr>
<tr>
<td><code>getID</code></td>
<td>Returns the ID value of the signature.</td>
</tr>
<tr>
<td><code>getSignatureMethod</code></td>
<td>Returns the algorithm method of the signature.</td>
</tr>
<tr>
<td><code>getType</code></td>
<td>Returns a string with the type of the signature object.</td>
</tr>
</tbody>
</table>

#### Example Code:
```java
getCanonicalization() RETURNS STRING
getDocument() RETURNS xml.DomDocument
getID() RETURNS STRING
getSignatureMethod() RETURNS STRING
getType() RETURNS STRING
```
### Table 691: Object methods: Reference modifier

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`appendReferenceTransformation(</td>
<td>Appends transformations related to the specified reference index.</td>
</tr>
<tr>
<td>referenceIndex INTEGER,</td>
<td></td>
</tr>
<tr>
<td>method STRING,</td>
<td></td>
</tr>
<tr>
<td>args ... )</td>
<td></td>
</tr>
<tr>
<td>`createReference(</td>
<td>Creates a new reference that will be signed with the <code>compute()</code> method</td>
</tr>
<tr>
<td>uri STRING,</td>
<td></td>
</tr>
<tr>
<td>digest STRING )</td>
<td></td>
</tr>
<tr>
<td>RETURNS INTEGER</td>
<td></td>
</tr>
<tr>
<td>`setReferenceID(</td>
<td>Sets an ID for the signature reference in the specified signature object.</td>
</tr>
<tr>
<td>index INTEGER,</td>
<td></td>
</tr>
<tr>
<td>id STRING )</td>
<td></td>
</tr>
</tbody>
</table>

### Table 692: Object methods: Reference access

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getReferenceCount()</code></td>
<td>Returns the number of references in this Signature object.</td>
</tr>
<tr>
<td>RETURNS INTEGER</td>
<td></td>
</tr>
<tr>
<td>`getReferenceDigest(</td>
<td>Returns the digest algorithm identifier of the reference.</td>
</tr>
<tr>
<td>index INTEGER )</td>
<td></td>
</tr>
<tr>
<td>RETURNS STRING</td>
<td></td>
</tr>
<tr>
<td>`getReferenceURI(</td>
<td>Returns the URI of the reference in this signature object.</td>
</tr>
<tr>
<td>index INTEGER )</td>
<td></td>
</tr>
<tr>
<td>RETURNS STRING</td>
<td></td>
</tr>
<tr>
<td>`getReferenceID(</td>
<td>Returns the ID value of the reference in this signature object.</td>
</tr>
<tr>
<td>index INTEGER )</td>
<td></td>
</tr>
<tr>
<td>RETURNS STRING</td>
<td></td>
</tr>
<tr>
<td>`getReferenceTransformation(</td>
<td>Gets the transformation identifier related to the reference of index <code>referenceIndex</code>.</td>
</tr>
<tr>
<td>referenceIndex INTEGER,</td>
<td></td>
</tr>
<tr>
<td>index INTEGER )</td>
<td></td>
</tr>
<tr>
<td>RETURNS STRING</td>
<td></td>
</tr>
<tr>
<td>`getReferenceTransformationCount(</td>
<td>Returns the number of transformations referenced in this signature object.</td>
</tr>
<tr>
<td>referenceIndex INTEGER )</td>
<td></td>
</tr>
<tr>
<td>RETURNS INTEGER</td>
<td></td>
</tr>
</tbody>
</table>
### Table 693: Object methods: Object modifier

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>appendObjectData(index INTEGER, node xml.DomNode )</code></td>
<td>Appends a copy of a XML DomNode to the signature object index.</td>
</tr>
<tr>
<td><code>createObject()</code> RETURNS INTEGER</td>
<td>Creates a new object that will embed additional XML nodes.</td>
</tr>
<tr>
<td><code>setObjectID(index INTEGER, id STRING )</code></td>
<td>Sets an ID for the signature object.</td>
</tr>
</tbody>
</table>

### Table 694: Object methods: Object access

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getObjectCount()</code> RETURNS INTEGER</td>
<td>Returns the number of objects in this Signature object.</td>
</tr>
<tr>
<td><code>getObjectId(index INTEGER )</code> RETURNS STRING</td>
<td>Returns the ID value of the signature object.</td>
</tr>
</tbody>
</table>

**Note:** In addition to these object methods categorized under Object Access, there is also a class method. It is listed in Table 687: Class methods: Object access on page 3076.
Table 695: Object methods: Signature computation and verification

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>compute(doc xml.DomDocument )</code></td>
<td>Computes the signature of all references set in this Signature object.</td>
</tr>
<tr>
<td><code>signString(key xml.CryptoKey, str STRING )</code></td>
<td>Sign the passed string according to the specified key.</td>
</tr>
<tr>
<td><code>verify(doc xml.DomDocument )</code></td>
<td>Verifies that all references in this signature object have not changed.</td>
</tr>
<tr>
<td><code>verifyString(key xml.CryptoKey, originalStr STRING, signature STRING )</code></td>
<td>Verify the signature is consistent with the given key and the original message.</td>
</tr>
</tbody>
</table>

`xml.Signature.appendObjectData`
Append a copy of a XML DomNode to the signature object index.

**Syntax**

```java
appendObjectData(
    index INTEGER,
    node xml.DomNode
)
```

1. `index` defines the index in this Signature object.
2. `node` defines the `xml.DomNode`.

**Usage**

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

`xml.Signature.appendReferenceTransformation`
Appends transformations related to the specified reference index.

**Syntax**

```java
appendReferenceTransformation(
    referenceIndex INTEGER,
    method STRING,
    args ...
)
```

1. `referenceIndex` defines the index in this Signature object.
2. `method` represents an URL as identifier of the transformation algorithm.
3. `args` defines a list of transformations.
Usage

This method appends a reference transformation that is executed before any computation.

A transformation modifies the reference URI before signing or validating it. Several transformations are executed one after another, and only once the last transformation has been applied, is the reference really signed or verified.

Depending on the transformation identifier, additional parameters are necessary.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.Signature.compute

Computes the signature of all references set in this Signature object.

Syntax

```java
compute(
    doc xml.DomDocument )
```

1. `doc` defines the XML document.

Usage

If the signature type is:

- Enveloping: then `doc` must be NULL because all document fragment references are inside the Signature itself
- Enveloped: then `doc` must be the XML document where the signature must be added afterwards to get a valid enveloped signature
- Detached: then `doc` can be NULL if all references are absolute, otherwise it can be the XML document fragment references that are referencing

See XML Signature concepts for more details.

Also, see Windows®.NET special recommendation.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.Signature.Create

Constructor of a blank Signature object.

Syntax

```java
xml.Signature.Create() 
RETURNS xml.Signature
```

Usage

Returns a XML Signature object or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.Signature.CreateFromNode
Constructor of a new Signature object from a XML Signature node, based on the XML-Signature specification.

Syntax

```
xml.Signature.CreateFromNode(
    node xml.DomNode )
RETURNS xml.Signature
```

1. `node` defines the XML Signature node.

Usage

Returns a XML Signature object or NULL.

The `node` must be an element node with Signature as the local name, and it must belong to the XML-Signature namespace http://www.w3.org/2000/09/xmldsig#, as defined in www.w3.org.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

xml.Signature.createObject

Creates a new object that will embed additional XML nodes.

Syntax

```
createObject ()
RETURNS INTEGER
```

Usage

The returned value represents the index for any further manipulation of this signature object.

Note: An object is enveloping additional XML nodes, but is not necessarily signed unless there is a reference on it.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

xml.Signature.createReference

Creates a new reference that will be signed with the `compute()` method.

Syntax

```
createReference (
    uri STRING,
    digest STRING )
RETURNS INTEGER
```

1. `uri` represents the data to be signed.
2. `digest` defines a URL as identifier for the hash algorithm.

Usage

The returned value represents the index for any further manipulation of this reference.

The `uri` can be:
• An absolute URL such as http://, https://, tcp://, tcps://, file:/// and alias:// (see FGLPROFILE Configuration for more details about URL mapping with aliases), and where the data can be a XML document or any kind of data such as images or HTML pages.
• NULL to sign the whole document, but only one NULL is allowed in the entire signature.
• A fragment like #tobesigned. Note that a DOM node fragment is identified via the value of an attribute of type ID such as xml:id or any attribute whose type was changed to ID with xml.DomNode.setIdAttribute() or xml.DomNode.setIdAttributeNS().

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

Related concepts
xml.Signature.compute on page 3081
Computes the signature of all references set in this Signature object.

xml.Signature.getCanonicalization
Returns the canonicalization identifier of the signature.

Syntax

getCanonicalization()  
RETURNS STRING

Usage
Returns one of the four canonicalization identifiers of the signature.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.Signature.getDocument
Returns a new DomDocument object representing the signature in XML.

Syntax

getDocument()  
RETURNS xml.DomDocument

Usage
Returns a xml.DomDocument object.

If the signature type is enveloped, it's up to the user to add it at the right place in the XML document it is intended to sign.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.Signature.getID
Returns the ID value of the signature.

Syntax

getID()  
RETURNS STRING
Usage

This method returns the ID value of the signature.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.Signature.getObjectCount

Returns the number of objects in this Signature object.

Syntax

```java
getObjectCount ()
RETURNS INTEGER
```

Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.Signature.getObjectId

Returns the ID value of the signature object.

Syntax

```java
getObjectId (index INTEGER )
RETURNS STRING
```

1. `index` defines the index in this Signature object.

Usage

This method returns the ID value of the index (index) in this signature object.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.Signature.getReferenceCount

Returns the number of references in this Signature object.

Syntax

```java
getReferenceCount ()
RETURNS INTEGER
```

Usage

This method returns the number of references in the specified signature object.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.Signature.getReferenceDigest
Returns the digest algorithm identifier of the reference.

### Syntax

```java
getReferenceDigest (  
    index INTEGER  
)  
RETURNS STRING
```

1. `index` defines the index in this signature object.

#### Usage

Returns the digest algorithm identifier of the reference of index (`index`) in this signature object.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (`STATUS`) on page 3554.

xml.Signature.getReferenceID

Returns the ID value of the reference in this signature object.

### Syntax

```java
getReferenceID (  
    index INTEGER  
)  
RETURNS STRING
```

1. `index` is the index in this Signature object.

#### Usage

This method returns the ID value of the reference of index (`index`) in this signature object, or NULL if there is none.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (`STATUS`) on page 3554.

xml.Signature.getReferenceTransformation

Gets the transformation identifier related to the reference of index `referenceIndex`.

### Syntax

```java
getReferenceTransformation (  
    referenceIndex INTEGER,  
    index INTEGER  
)  
RETURNS STRING
```

1. `referenceIndex` is the index in this Signature object.
2. `index` defines the position in the list of transformations.

#### Usage

Returns the transformation identifier related to the reference of index `referenceIndex`, at the position specified by `index` in the list of transformations.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (`STATUS`) on page 3554.

xml.Signature.getReferenceTransformationCount
Returns the number of transformations referenced in this signature object.

**Syntax**

```plaintext
getReferenceTransformationCount ( referenceIndex INTEGER )
RETURNS INTEGER
```

1. `referenceIndex` defines the index in this signature object.

**Usage**

This method returns the number of transformations related to the reference of index `referenceIndex` in this signature object.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**xml.Signature.getReferenceURI**

Returns the URI of the reference in this signature object.

**Syntax**

```plaintext
getReferenceURI ( index INTEGER )
RETURNS STRING
```

1. `index` defines the index in this signature object.

**Usage**

This method returns the URI of the reference specified in the index `index` in this signature object.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**xml.Signature.getSignatureMethod**

Returns the algorithm method of the signature.

**Syntax**

```plaintext
getSignatureMethod ()
RETURNS STRING
```

**Usage**

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**xml.Signature.getType**

Returns a string with the type of the signature object.

**Syntax**

```plaintext
getType ()
```
xml.Signature.RetrieveObjectDataListFromSignatureNode

Returns a DomNodeList containing all embedded XML nodes related to the signature object.

Syntax

```xml
xml.Signature.RetrieveObjectDataListFromSignatureNode(
    signNode xml.DomNode,
    index INTEGER
) RETURNS xml.DomNodeList
```

1. `signNode` defines the XML Signature xml.DomNode.
2. `index` defines the index of the signature object.

Usage

This method returns a xml.DomNodeList containing all embedded XML nodes related to the signature object specified by the index `index` within the XML signature node, `signNode`.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.Signature.setCanonicalization

Sets the canonicalization method to use for the signature.

Syntax

```xml
setCanonicalization(
    url STRING
)
```

1. `url` defines one of the four canonicalization identifier.

Usage

This method sets the canonicalization method to use for the signature. The default value is the `c14n` method.

**Note:** Windows®.NET default `c14n` canonicalization method is not compatible with the W3C standard, therefore it is recommended to use the `exc-c14n` method when interoperating with a Windows® system.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.Signature.setCertificate

Defines the X509 certificate to be added to the signature object when signing a document.

Syntax

```xml
setCertificate(
)
```
1. `cert defines the X509 certificate to be added.``

**Usage**

If NULL, no certificate is added.

During the computation of the signature, some certificate information can be added based on the `feature` set on that CryptoX509 object. If no features are set, the complete X509 certificate is automatically added.

During the verification of a signature the certificate set with the `setCertificate` method isn't used. See **XML Signature concepts** for more details.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See **Error handling in GWS calls (STATUS)** on page 3554.

**xml.Signature.setID**

Sets an ID value for the signature.

**Syntax**

```plaintext
setId(id STRING )
```

1. `id` defines the ID value to be set.

**Usage**

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See **Error handling in GWS calls (STATUS)** on page 3554.

**xml.Signature.setKey**

Defines the key used for signing or validation.

**Syntax**

```plaintext
setKey(key xml.CryptoKey )
```

1. `key` defines the `key` to be used for signing or validation.

**Usage**

Only RSA, DSA or HMAC keys intended for `SIGNATURE` are allowed.

During the computation of the signature, some key information can be added depending on the `feature` set on that CryptoKey object. If no features are set, nothing is added. See **XML Signature concepts** for more details.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See **Error handling in GWS calls (STATUS)** on page 3554.

**xml.Signature.setObjectID**

Sets an ID for the signature object.

**Syntax**

```plaintext
setObjectID(
```
xml.Signature.setReferenceID
Sets an ID for the signature reference in the specified signature object.

Syntax

```java
setReferenceID(
    index INTEGER,
    id STRING )
```

1. `index` defines the index value.
2. `id` defines the value to be set.

Usage
This method sets the ID defined by `id` value for the signature object specified in the index (`index`).

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.Signature.signString
Sign the passed string according to the specified key.

Syntax

```java
signString(
    key xml.CryptoKey,
    str STRING
) RETURNS STRING
```

1. `key` defines the key to be used for the signature.
2. `str` defines the string to be signed.

Usage
The key can be a HMAC key, a RSA private key, or a DSA private key. The signing process is performed with SHA-1 digest, as recommended by the XMLSec specification.

Returns `sig`, or the signature in base64 format.

This method does not belong to the XML encryption specification.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.
xml.Signature.verify

Verifies that all references in this signature object have not changed.

**Syntax**

```plaintext
verify(
    doc xml.DomDocument
)
RETURNS INTEGER
```

1. `doc` defines the XML document.

**Usage**

Returns TRUE if valid, FALSE otherwise.

If the signature type is:

- Enveloping: then `doc` must be NULL because all document fragment references are inside the Signature itself
- Enveloped: then `doc` must be the XML document where the signature was enveloped
- Detached: then `doc` can be null if all references are absolute, otherwise it can be the XML document the fragment references are referencing

See XML Signature concepts for more details.

By default, the validation process uses the CryptoKey set with `xml.Signature.setKey()` to verify the signature. However, if the signature contains a X509 certificate or a X509 retrieval method, it uses the list of trusted certificates, or if the signature contains a RSA or DSA retrieval method, it uses the RSA or DSA public key automatically loaded.

**Note:** See Windows® .NET special recommendation.

Before loading the XML document to verify the signature, you might need to set some options to retrieve the "id" nodes with the `xml.DomDocument.setFeature()` method:

```plaintext
DEFINE doc xml.DomDocument
...
CALL doc.setFeature(feature, TRUE)
...
```

Here `feature` must be "auto-id-attribute" if the "id" attribute has no namespace, or "auto-id-qualified-attribute", when "id" has a namespace.

This is especially needed when you encounter error messages such as:

```plaintext
Xml security operation failed : libxml2 library function failed : expr=xpointer(id('id-1436767651')).
```

Meaning that the parser was unable to find the "id" attribute in the XML document.

Note that the "auto-id-*" features will declare all XML attributes where the name is "id", "ID", "Id" or "iD" to be of type ID, and thus be usable via `xml.DomDocument.getElementById()` method used during signature validation.

If needed, you can also set features for a specific attribute with the `xmlDOMNode.setIdAttribute()` method, or with the `xmlDOMNode.setIdAttributeNS()` method.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.Signature.verifyString
Verify the signature is consistent with the given key and the original message.

**Syntax**

```java
verifyString(
    key xml.CryptoKey,
    originalStr STRING,
    signature STRING )
RETURNS INTEGER
```

1. `key` defines the key to use for verification.
2. `originalStr` defines the signed string in its clear form.
3. `toBeVerifiedStr` defines the signature to be verified.

**Usage**

The key can be a HMAC key, a RSA private key or a DSA private key. The HMAC key must be the same as the one used for signing. The public RSA and DSA key must be the public key corresponding to the private key used for signing.

Returns 1 when verification is successful; 0 (zero) is returned if verification fails.

This method does not belong to the XML encryption specification.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**XML Signature concepts**

The purpose of a signature is to guarantee the integrity of a XML document, that it was not altered, and that it still contains the same data as when it was created. An additional purpose of a signature is to authenticate the author of the document. There are different ways to achieve this guarantee.

Sign and verify with a common shared HMAC key

Use if the sender of the XML document and the receiver share a common secret key.

**How to sign**

1. Create a HMAC key with the constructor of the CryptoKey class.
2. Set or load the common shared key value in the CryptoKey object.
3. Create a blank signature with the constructor of the Signature class.
4. Assign the CryptoKey object to the Signature object.
5. Create one or more references to be signed.
6. Compute the signature.
7. Retrieve the XML signature document from the Signature object.

**How to verify**

1. Create a HMAC key with the constructor of the CryptoKey.
2. Set or load the common shared key value in the CryptoKey object.
3. Create a signature with the constructor of the Signature class and from a XML signature node obtained after the above compute operation.
4. Assign the CryptoKey object to the Signature object.
5. Verify the signature validity.

**Related concepts**

Examples on page 3099
xml.Signature usage examples.

Sign with the originator private RSA or DSA key, and verify with the originator public RSA or DSA key
Use if the receiver of the XML document has the RSA or DSA public key of the sender.

Only the originator can sign a message with this specific pair of keys. Any other peer needs the corresponding public key and does not have access to the private key.

How to sign
1. Create a RSA or DSA key with the constructor of the CryptoKey class.
2. Load the RSA or DSA private key into the CryptoKey object.
3. Create a blank signature with the constructor of the Signature class.
4. Assign the CryptoKey object to the Signature object.
5. Create one or more references to be signed.
6. Compute the signature.
7. Retrieve the XML signature document from the Signature object.

How to verify
1. Create a RSA or DSA key with the constructor of the CryptoKey class.
2. Load the RSA or DSA public key into the CryptoKey object.
3. Create a signature with the constructor of the Signature class and from a XML signature node obtained after the above compute operation.
4. Assign the CryptoKey object to the Signature object.
5. Verify the signature validity.

Related concepts
Examples on page 3099
xml.Signature usage examples.

Sign with the originator private RSA or DSA key, and verify with a RSA or DSA retrieval method
Use if the sender of the XML document provides the public RSA or DSA key in XML form (and via http, tcp or a file protocol).

Only the originator can sign a message with this specific pair of keys. Any other peer needs the corresponding public key and does not have access to the private key.

How to sign
1. Create a RSA or DSA key with the constructor of the CryptoKey class.
2. Load the RSA or DSA private key into the CryptoKey object.
3. Set the RetrievalMethod feature on the CryptoKey object with the URL where the XML form of the public RSA or DSA key is available.
4. Create a blank signature with the constructor of the Signature class.
5. Assign the CryptoKey object to the Signature object.
6. Create one or more references to be signed.
7. Compute the signature.
8. Retrieve the XML signature document from the Signature object.

How to verify
1. Create a signature with the constructor of the Signature class and from a XML signature node obtained after the above compute operation.
2. Verify the signature validity.

Note: There is no key nor certificate to set in the Signature object during validation.
Sign with the originator private RSA or DSA key, and verify with the originator X509 certificate associated to the private RSA or DSA key.

Use if the receiver of the XML document has the X509 certificate associated to the RSA or DSA private key.

Only the originator can sign a message with this specific pair of keys. Any other peer needs the corresponding public key and does not have access to the private key.

**How to sign**

1. Create a RSA or DSA key with the constructor of the CryptoKey class.
2. Load the RSA or DSA private key into the CryptoKey object.
3. Create a blank signature with the constructor of the Signature class.
4. Assign the CryptoKey object to the Signature object.
5. Create one or more references to be signed.
6. Compute the signature.
7. Retrieve the XML signature document from the Signature object.

**How to verify**

1. Create a X509 certificate with the constructor of the CryptoX509 class.
2. Load the X509 certificate into the CryptoKey object.
3. Create the RSA or DSA public key from the X509 certificate of the CryptoX509 object.
4. Create a signature with the constructor of the Signature class and from a XML signature node obtained after the above compute operation.
5. Assign the CryptoKey object containing the public key to the Signature object.
6. Verify the signature validity.

Sign with the originator private RSA or DSA key, and verify with trusted X509 certificates.

Use if the sender of the XML document adds a X509 certificate that was signed by another trusted X509 certificate.

Only the originator can sign a message with this specific pair of keys. Any other peer needs the corresponding public key and does not have access to the private key.

**How to sign**

1. Create a RSA or DSA key with the constructor of the CryptoKey class.
2. Load the RSA or DSA private key into the CryptoKey object.
3. Create a X509 certificate with the constructor of the CryptoX509 class.
4. Load the X509 certificate associated to the RSA or DSA private key into the CryptoKey object.
5. Create a blank signature with the constructor of the Signature class.
6. Assign the CryptoKey object to the Signature object.
7. Assign the CryptoX509 object to the Signature object.
8. Create one or more references to be signed.
9. Compute the signature.
10. Retrieve the XML signature document from the Signature object.
How to verify

1. Create a X509 certificate with the constructor of the CryptoX509 class.
2. Load the X509 certificate that was used to sign the originator X509 certificate into the CryptoX509 object.
3. Add the X509 certificate as trusted certificate to the application.
4. Create a signature with the constructor of the Signature class and from a XML signature node obtained after the above compute operation.
5. Verify the signature validity.

Note: Point 1 to 3 can be omitted if entry xml.application.calist has been set in FGLPROFILE file with the trusted certificate.

Note: There is no key nor certificate to set in the Signature object during validation.

Related concepts

Examples on page 3099
xml.Signature usage examples.

Sign with the originator private RSA or DSA key, and verify with a X509 certificate retrieval method and trusted X509 certificates

Use if the sender of the XML document adds a X509 retrieval method that was signed by another trusted X509 certificate.

Only the originator can sign a message with this specific pair of keys. Any other peer needs the corresponding public key and does not have access to the private key.

How to sign

1. Create a RSA or DSA key with the constructor of the CryptoKey class.
2. Load the RSA or DSA private key into the CryptoKey object.
3. Create a X509 certificate with the constructor of the CryptoX509 class.
4. Set the RetrievalMethod feature on the CryptoX509 object with the URL where the XML form of the originator X509 certificate is available.
5. Create a blank signature with the constructor of the Signature class.
6. Assign the CryptoKey object to the Signature object.
7. Assign the CryptoX509 object to the Signature object.
8. Create one or more references to be signed.
9. Compute the signature.
10. Retrieve the XML signature document from the Signature object.

How to verify

1. Create a X509 certificate with the constructor of the CryptoX509 class.
2. Load the X509 certificate that was used to sign the originator X509 certificate into the CryptoX509 object.
3. Add the X509 certificate as trusted certificate to the application.
4. Create a signature with the constructor of the Signature class and from a XML signature node obtained after the above compute operation.
5. Verify the signature validity.

Note: Steps 1 - 3 can be omitted if entry xml.application.calist has been set in FGLPROFILE file with the trusted certificate.

Note: There is no key or certificate to set in the Signature object during validation.

Related concepts

Examples on page 3099
xml.Signature usage examples.

Sign with a named key and verify using the keystore

Use if the sender and the receiver exchange multiple XML documents signed with different keys.

How to sign

1. Create a HMAC, RSA or DSA key with the constructor of the CryptoKey class.
2. Set the HMAC key or load the RSA or DSA key in the CryptoKey object.
3. Set the KeyName feature with the name identifying the key.
4. Create a blank signature with the constructor of the Signature class.
5. Assign the CryptoKey object to the Signature object.
6. Create one or more references to be signed.
7. Compute the signature.
8. Retrieve the XML signature document from the Signature object.

How to verify

1. Create a HMAC, RSA or DSA key with the constructor of the CryptoKey.
2. Set the HMAC key or load the RSA or DSA key in the CryptoKey object.
3. Set the KeyName feature with the name identifying the key.
4. Register the key to be used by key name for any signature verification.
5. Create a signature with the constructor of the Signature class and from a XML signature node obtained after the above compute operation.
6. Verify the signature validity.

Note: It is recommended that steps 1 to 4 are done once at application start-up for each key used in the application. Steps 5 - 6 can then be executed quickly for any XML signature to be checked.

Related concepts

Examples on page 3099
xml.Signature usage examples.

Digest identifier

Table 696: Digest identifiers

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.w3.org/2000/09/xmlsig#sha1">http://www.w3.org/2000/09/xmlsig#sha1</a></td>
<td>Computes the digest of the reference set with createReference(), by applying a hash operation using a SHA algorithm of 160 bits.</td>
</tr>
<tr>
<td>See specification for details.</td>
<td>Note: It is the only digest algorithm recommended by the W3C.</td>
</tr>
<tr>
<td><a href="http://www.w3.org/2001/04/xmlenc#sha512">http://www.w3.org/2001/04/xmlenc#sha512</a></td>
<td>Computes the digest of the reference set with createReference(), by applying a hash operation using a SHA algorithm of 512 bits.</td>
</tr>
<tr>
<td>See specification for details.</td>
<td></td>
</tr>
<tr>
<td><a href="http://www.w3.org/2001/04/xmlsig-more#sha384">http://www.w3.org/2001/04/xmlsig-more#sha384</a></td>
<td>Computes the digest of the reference set with createReference(), by applying a hash operation using a SHA algorithm of 384 bits.</td>
</tr>
<tr>
<td>See specification for details.</td>
<td></td>
</tr>
<tr>
<td><a href="http://www.w3.org/2001/04/xmlenc#sha256">http://www.w3.org/2001/04/xmlenc#sha256</a></td>
<td>Computes the digest of the reference set with createReference(), by applying a hash operation using a SHA algorithm of 256 bits.</td>
</tr>
<tr>
<td>See specification for details.</td>
<td></td>
</tr>
<tr>
<td>Identifier</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><a href="http://www.w3.org/2001/04/xmldsig-more#sha224">http://www.w3.org/2001/04/xmldsig-more#sha224</a></td>
<td>Computes the digest of the reference set with createReference(), by applying a hash operation using a SHA algorithm of 224 bits.</td>
</tr>
<tr>
<td>See specification for details.</td>
<td></td>
</tr>
<tr>
<td><a href="http://www.w3.org/2001/04/xmldsig-more#md5">http://www.w3.org/2001/04/xmldsig-more#md5</a></td>
<td>Computes the digest of the reference set with createReference(), by applying a hash operation using a MD5 algorithm.</td>
</tr>
<tr>
<td>See specification for details.</td>
<td></td>
</tr>
<tr>
<td><a href="http://www.w3.org/2001/04/xmlenc#ripemd160">http://www.w3.org/2001/04/xmlenc#ripemd160</a></td>
<td>Computes the digest of the reference set with createReference(), by applying a hash operation using a RIPEMD algorithm.</td>
</tr>
<tr>
<td>See specification for details.</td>
<td></td>
</tr>
</tbody>
</table>

**Related concepts**

xml.Signature.createReference on page 3082

Creates a new reference that will be signed with the compute() method

**Transformation identifier**

**Table 697: Transformation identifiers**

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
<th>Additional Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.w3.org/2000/09/xmldsig#base64">http://www.w3.org/2000/09/xmldsig#base64</a></td>
<td>Transforms the output from the previous transformation (or the reference if there is no previous transformation), into the raw data associated to a BASE64 encoded form. This is intended to sign the raw data associated with the BASE64 encoded content of an element. See specification for details.</td>
<td>No</td>
</tr>
<tr>
<td>See specification for details.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| http://www.w3.org/TR/2001/REC-xml-c14n-20010315                            | Transforms the output from the previous transformation (or the reference if there is no previous transformation), into a canonicalized XML document without any XML comments. This is intended to transform two equivalent XML documents into a standardized XML representation in order to obtain the same hash value. For instance: The following two XML nodes are equivalent but would produce different hash values if not canonicalized.
   • <tag Attr1="hello" Attr2="world" />
   • <tag Attr2="world" Attr1="hello" />                                                                 | No                    |
<p>| See specification for details.                                            |                                                                                                                                                                                                            |                       |</p>
<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
<th>Additional Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.w3.org/TR/2001/REC-xml-c14n-20010315">http://www.w3.org/TR/2001/REC-xml-c14n-20010315</a> #WithComments</td>
<td>Transforms the output from the previous transformation (or the reference if there is no previous transformation), into a canonicalized XML document keeping all XML comments. This is intended to transform two equivalent XML documents into a standardized XML representation in order to obtain the same hash value. For instance: The following two XML nodes are equivalent but would produce different hash values if not canonicalized. • <code>&lt;tag Attr1=&quot;hello&quot; Attr2=&quot;world&quot;/&gt;</code> • <code>&lt;tag Attr2=&quot;world&quot; Attr1=&quot;hello&quot; /&gt;</code></td>
<td>No</td>
</tr>
<tr>
<td><a href="http://www.w3.org/2001/10/xml-exc-c14n#">http://www.w3.org/2001/10/xml-exc-c14n#</a></td>
<td>Transforms the output from the previous transformation (or the reference if there is no previous transformation), into a canonicalized XML document without any XML comments, and removing all unused namespaces declaration. This is intended to transform two equivalent XML documents into a standardized XML representation in order to obtain the same hash value. For instance: The following two XML nodes are equivalent but would produce different hash values if not canonicalized. • <code>&lt;tag Attr1=&quot;hello&quot; Attr2=&quot;world&quot;/&gt;</code> • <code>&lt;tag Attr2=&quot;world&quot; Attr1=&quot;hello&quot; /&gt;</code></td>
<td>No</td>
</tr>
<tr>
<td><a href="http://www.w3.org/2001/10/xml-exc-c14n">http://www.w3.org/2001/10/xml-exc-c14n</a> #WithComments</td>
<td>Transforms the output from the previous transformation (or the reference if there is no previous transformation), into a canonicalized XML document keeping all XML comments, and removing all unused namespaces declaration. This is intended to transform two equivalent XML documents into a standardized XML representation in order to obtain the same hash value. For instance: Following two XML nodes are equivalent but would produce different hash values if not canonicalized. • <code>&lt;tag Attr1=&quot;hello&quot; Attr2=&quot;world&quot;/&gt;</code> • <code>&lt;tag Attr2=&quot;world&quot; Attr1=&quot;hello&quot; /&gt;</code></td>
<td>No</td>
</tr>
<tr>
<td>Identifier</td>
<td>Description</td>
<td>Additional Parameters</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td><a href="http://www.w3.org/2000/09/xmldsig">http://www.w3.org/2000/09/xmldsig</a> #enveloped-signature</td>
<td>Transforms the output from the previous transformation (or the reference if there is no previous transformation), into the same XML document or fragment, but without the Signature node. This is intended to create enveloped signatures where the <code>&lt;dsig:Signature&gt;</code> node is inside the document, but without taking it into account during signature computation or verification. See specification for details.</td>
<td>This method can take an optional parameter that is the node where the enveloped signature will be appended.</td>
</tr>
<tr>
<td><a href="http://www.w3.org/TR/1999/REC-xpath-19991116">http://www.w3.org/TR/1999/REC-xpath-19991116</a></td>
<td>Transforms the output from the previous transformation (or the reference if there is no previous transformation), into a XML document based on the XPath filtering expression applied to each node of the input document, where the expression represents a predicate to the XPath expression `//.</td>
<td>//@*</td>
</tr>
</tbody>
</table>

```java
call s.appendReferenceTransformation( i, "http://www.w3.org/TR/1999/REC-xpath-19991116", "ancestor-or-self::MyCode", NULL)
```

See specification for details.
<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
<th>Additional Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.w3.org/2002/06/xmldsig-filter2">http://www.w3.org/2002/06/xmldsig-filter2</a></td>
<td>Transforms the output from the previous transformation (or the reference if there is no previous transformation), into a XML document according to a XPath filtering 2.0 expression applied to the entire document at once. This is intended to identify the nodes to be signed using a XPath expression instead of an attribute of type ID, and to perform fast and more complex operations such as intersect, subtract or union. For instance: The following samples output the entire document without the MyCode node child of the MyElement root node:</td>
<td>XPathFilter2.0 type (intersect, subtract or union), followed by the XPath expression, followed by NULL or a list of prefix, namespace matching the XPath expression.</td>
</tr>
<tr>
<td>See specification for details.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Examples**

xml.Signature usage examples.
Create a detached signature using a HMAC key

```xml
IMPORT xml

MAIN
DEFINE doc xml.DomDocument
DEFINE sig xml.Signature
DEFINE key xml.CryptoKey
DEFINE index INTEGER
# Create DomDocument object
LET doc = xml.DomDocument.Create()
# Notice that whitespaces are significant in cryptography, therefore it is recommended that you remove unnecessary ones
CALL doc.setFeature("whitespace-in-element-content",FALSE)
TRY
    # Load document to be signed
    CALL doc.load("MyDocument.xml")
    # Create HMAC key
```
LET key = xml.CryptoKey.Create("http://www.w3.org/2000/09/xmldsig#hmac-sha1")
CALL key.setKey("secretpassword")
LET sig = xml.Signature.Create()
CALL sig.setKey(key)
LET index = sig.createReference('#code', "http://www.w3.org/2000/09/xmldsig#sha1")
CALL sig.appendReferenceTransformation(index, "http://www.w3.org/2001/10/xml-exc-c14n")
LET doc = xml.DomDocument.Create()
CALL doc.setFeature("whitespace-in-element-content",FALSE)
TRY
CALL doc.load("MyDocumentDetachedSignature.xml")
LET sig = xml.Signature.CreateFromNode(doc.getDocumentElement())
CALL key.setKey("secretpassword")
LET doc = xml.DomDocument.Create()
CALL doc.setFeature("whitespace-in-element-content",FALSE)
CALL doc.load("MyDocument.xml")
LET isVerified = sig.verify(doc)
IF isVerified THEN
DISPLAY "Signature OK"
ELSE
DISPLAY "Signature FAILED"
ENDIF
Note: All keys or certificates in PEM or DER format were created with the OpenSSL tool.

Create an enveloping signature using a DSA key

```plaintext
IMPORT xml

MAIN
DEFINE doc xml.DomDocument
DEFINE sig xml.Signature
DEFINE key xml.CryptoKey
DEFINE index INTEGER
DEFINE objInd INTEGER
# Create DomDocument object
LET doc = xml.DomDocument.Create()
# Notice that whitespaces are significant in cryptography,
# therefore it is recommended to remove unnecessary ones
CALL doc.setFeature("whitespace-in-element-content",FALSE)
TRY
    # Load document to be signed
    CALL doc.load("MyDocument.xml")
    # Create DSA key and load it from file
    LET key = xml.CryptoKey.Create(
          "http://www.w3.org/2000/09/xmldsig#dsa-sha1")
    CALL key.loadPEM("DSAKey.pem")
    # Create signature object with the key to use
    LET sig = xml.Signature.Create()
    CALL sig.setKey(key)
    # Create an object inside the signature to envelop the root node
    LET objInd = sig.createObject()
    # Set the object id to get a reference
    CALL sig setObjectId(objInd,"data")
    # Copy the enveloping node from the document
    CALL sig.appendObjectData(objInd,doc.getDocumentElement())
    # Set the reference to be signed on the object node.
    # In our case, the object node with attribute 'data'
    LET index = sig.createReference("#data",
          "http://www.w3.org/2000/09/xmldsig#sha1")
    # Set canonicalization method on the enveloping object to be signed.
    CALL sig.appendReferenceTransformation(index,
          "http://www.w3.org/2001/10/xml-exc-c14n#")
    # Compute enveloping signature
    CALL sig.compute(NULL)
    # Retrieve signature document
    LET doc=sig.getDocument()
    # Save signature on disk
    CALL doc.setFeature("format-pretty-print",TRUE)
    CALL doc.save("MyDocumentEnvelopingSignature.xml")
CATCH
    DISPLAY "Unable to create an enveloping signature ":STATUS
END TRY
END MAIN
```

Note: All keys or certificates in PEM or DER format were created with the OpenSSL tool.

Verify an enveloping signature using a X509 certificate

```plaintext
IMPORT xml

MAIN
```
MAIN
DEFINE doc xml.DomDocument
DEFINE sig xml.Signature
DEFINE cert xml.CryptoX509
DEFINE pub xml.CryptoKey
DEFINE isVerified INTEGER
# Create DomDocument object
LET doc = xml.DomDocument.Create()
# Notice that whitespaces are significant in cryptography,
# therefore it is recommended to remove unnecessary ones
CALL doc.setFeature("whitespace-in-element-content",FALSE)
TRY
  # Load Signature into a DomDocument object
  CALL doc.load("MyDocumentEnvelopingSignature.xml")
  # Create signature object from DomDocument root node
  LET sig = xml.Signature.CreateFromNode(doc.getDocumentElement())
  # Create X509 certificate
  LET cert = xml.CryptoX509.Create()
  CALL cert.loadPEM("DSACertificate.crt")
  # Create public key from that X509 certificate
  LET pub = cert.createPublicKey("http://www.w3.org/2000/09/xmldsig#dsa-sha1")
  # Assign it to the signature
  CALL sig.setKey(pub)
  # Verify enveloping signature validity
  LET isVerified = sig.verify(NULL)
  # Notice that if something has been modified in the signature
  # or if the certificate isn't associated to the
  # private DSA key of example 3,
  # the program will display "FAILED".
  IF isVerified THEN
    DISPLAY "Signature OK"
  ELSE
    DISPLAY "Signature FAILED"
  END IF
CATCH
  DISPLAY "Unable to verify the enveloping signature :",STATUS
END TRY
END MAIN

Note: All keys or certificates in PEM or DER format were created with the OpenSSL tool.

Create an enveloped signature using a RSA key

IMPORT xml

MAIN
DEFINE doc xml.DomDocument
DEFINE doc2 xml.DomDocument
DEFINE root xml.DomNode
DEFINE node xml.DomNode
DEFINE signNode xml.DomNode
DEFINE sig xml.Signature
DEFINE key xml.CryptoKey
DEFINE index INTEGER
# Create DomDocument object
LET doc = xml.DomDocument.Create()
# Notice that whitespaces are significant in cryptography,
# therefore it is recommended to remove unnecessary ones
CALL doc.setFeature("whitespace-in-element-content",FALSE)
TRY
  # Load document to be signed
  CALL doc.load("MyDocument.xml")

# Create rsa key
LET key = xml.CryptoKey.Create("http://www.w3.org/2000/09/xmldsig#rsa-sha1")
CALL key.loadPEM("RSAKey.pem")
# Create signature object with the key to use
LET sig = xml.Signature.Create()
CALL sig.setKey(key)
# Set XML node to be signed. In our case, the node with
# attribute 'xml:id="code"
LET index = sig.createReference("#code",
    "http://www.w3.org/2000/09/xmldsig#sha1")
# Add enveloped method to not take the XML signature node into account
# when computing the entire document.
CALL sig.appendReferenceTransformation(index,
# Set canonicalization method on the XML fragment to be signed.
CALL sig.appendReferenceTransformation(index,
    "http://www.w3.org/2001/10/xml-exc-c14n#")
# Compute enveloped signature
CALL sig.compute(doc)
# Retrieve signature document
LET doc2=sig.getDocument()()
# Append the signature node to the original document to get
# a valid enveloped signature
# Notice that the enveloped signature can be added anywhere in the
# original document
LET signNode = doc2.getDocumentElement() # Get Signature node
# Import it into the original document
LET node = doc.importNode(signNode,true)
# Retrieve the original document root node
LET root = doc.getDocumentElement()
# Append the signature node as last child of the original document
CALL root.appendChild(node)
# Save document with enveloped signature back to disk
CALL doc.setFeature("format-pretty-print",TRUE)
CALL doc.save("MyDocumentEnvelopedSignature.xml")
CATCH
    DISPLAY "Unable to create an enveloped signature ":,STATUS
END TRY
END MAIN

Note: All keys or certificates in PEM or DER format were created with the OpenSSL tool.

Verify an enveloped signature using a RSA key

IMPORT xml

MAIN
DEFINE doc xml.DomDocument
DEFINE node xml.DomNode
DEFINE sig xml.Signature
DEFINE key xml.CryptoKey
DEFINE list xml.DomNodeList
DEFINE isVerified INTEGER
# Create DomDocument object
LET doc = xml.DomDocument.Create()
# Notice that whitespaces are significant in cryptography,
# therefore it is recommended to remove unnecessary ones
CALL doc.setFeature("whitespace-in-element-content",false)
TRY
    # Load original document with enveloped signature into a DomDocument object
CALL doc.load("MyDocumentEnvelopedSignature.xml")
# Because the signature can be anywhere in the original document,
# we must first retrieve it
LET list = doc.getElementsByTagNameNS("Signature",
    "http://www.w3.org/2000/09/xmldsig#")
IF list.getCount() != 1 THEN
    DISPLAY "Unable to find one Signature node"
    EXIT PROGRAM (-1)
ELSE
    LET node = list.getItem(1)
END IF
# Create RSA key
LET key = xml.CryptoKey.Create(
    "http://www.w3.org/2000/09/xmldsig#rsa-sha1")
CALL key.loadPEM("RSAKey.pem")
# Create signature object from DomNode object and set RSA key to use
LET sig = xml.Signature.CreateFromNode(node)
CALL sig.setKey(key)
# Verify enveloped signature validity of original document
LET isVerified = sig.verify(doc)
# Notice that if something has been modified in the node with
# attribute 'xml:id="code"' of the original XML document with the
# enveloped signature, the program will display "FAILED".
IF isVerified THEN
    DISPLAY "Signature OK"
ELSE
    DISPLAY "Signature FAILED"
END IF
CATCH
    DISPLAY "Unable to verify the enveloped signature :",STATUS
END TRY
END MAIN

Note: All keys or certificates in PEM or DER format were created with the OpenSSL tool.

The Encryption class
The xml.Encryption class provides methods to encrypt and decrypt XML documents, nodes or symmetric keys.

It follows the XML-Encryption specifications.

The STATUS variable is set to zero after a successful method call.

xml.Encryption methods
Methods for the xml.Encryption class.

### Table 698: Class methods: Creation

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RETURNS xml.Encryption</td>
<td></td>
</tr>
</tbody>
</table>
### Table 699: Class methods: String encryption and decryption

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xml.Encryption.DecryptString(key xml.CryptoKey, str STRING)</code></td>
<td>Decrypts an encrypted string encoded in BASE64, using the specified symmetric key, and returns the string in clear text.</td>
</tr>
<tr>
<td><code>xml.Encryption.EncryptString(key xml.CryptoKey, str STRING)</code></td>
<td>Encrypts the specified string using the symmetric key, and returns the encrypted string encoded in BASE64.</td>
</tr>
<tr>
<td><code>xml.Encryption.RSADecrypt(filename STRING, str STRING)</code></td>
<td>Decrypts the BASE64 encrypted string using the RSA key and returns it in clear text</td>
</tr>
<tr>
<td><code>xml.Encryption.RSAEncrypt(filename STRING, str STRING)</code></td>
<td>Encrypts the specified string using the RSA key and returns it encoded in BASE64.</td>
</tr>
</tbody>
</table>

The methods listed in Table 699: Class methods: String encryption and decryption do not belong to the XML encryption specification, but are helper functions to allow BDL application to encrypt and decrypt short passwords with RSA keys, or big strings by using symmetric keys. Notice that a common way to encrypt data is to use symmetric keys, and to use RSA keys to encrypt the symmetric key value.

### Table 700: Object methods: Key and certificate setting

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getEmbeddedKey()</code></td>
<td>Get a copy of the embedded symmetric key that was used in the last decryption operation.</td>
</tr>
<tr>
<td><code>setCertificate(cert xml.CryptoX509)</code></td>
<td>Assigns a copy of the X509 certificate to this encryption object.</td>
</tr>
<tr>
<td><code>setKey(key xml.CryptoKey)</code></td>
<td>Assigns a copy of the symmetric key to this encryption object.</td>
</tr>
<tr>
<td><code>setKeyEncryptionKey(key xml.CryptoKey)</code></td>
<td>Assigns a copy of the key-encryption key to this encryption object.</td>
</tr>
</tbody>
</table>
### Table 701: Object methods: XML elements encryption and decryption

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>decryptElement</code></td>
<td>Decrypts the encrypted data DomNode using the symmetric key.</td>
</tr>
<tr>
<td><code>decryptElementContent</code></td>
<td>Decrypts the encrypted data DomNode using the symmetric key.</td>
</tr>
<tr>
<td><code>encryptElement</code></td>
<td>Encrypts the element DomNode and all its children using the symmetric key.</td>
</tr>
<tr>
<td><code>encryptElementContent</code></td>
<td>Encrypts all child nodes of the element DomNode using the symmetric key.</td>
</tr>
</tbody>
</table>

### Table 702: Object methods: Detached XML elements encryption and decryption

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>decryptElementDetached</code></td>
<td>Decrypts the specified encrypted data DomNode using the symmetric key, and returns it in a new element <code>xml.DomNode</code>.</td>
</tr>
<tr>
<td><code>decryptElementContentDetached</code></td>
<td>Decrypts the encrypted data DomNode using the symmetric key, and returns all its children in one new document fragment type <code>xml.DomNode</code>.</td>
</tr>
<tr>
<td><code>encryptElementDetached</code></td>
<td>Encrypts the element DomNode and all its children using the symmetric key, and returns them as one new encrypted-data node.</td>
</tr>
<tr>
<td><code>encryptElementContentDetached</code></td>
<td>Encrypts all child nodes of the element DomNode using the symmetric key, and returns them as one new encrypted-data node.</td>
</tr>
</tbody>
</table>

### Table 703: Object methods: Key encryption and decryption

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>decryptKey</code></td>
<td>Decrypts the <code>EncryptedKey</code> as root in the given XML document, and returns a new CryptoKey of the given kind.</td>
</tr>
<tr>
<td><code>encryptKey</code></td>
<td>Encrypts the given symmetric or HMAC key as an encrypted-key node and returns it as root node of a new XML document.</td>
</tr>
</tbody>
</table>
xml.Encryption.Create

Constructor of an Encryption object.

**Syntax**

```csharp
xml.Encryption.Create()
    RETURNS xml.Encryption
```

**Usage**

Returns an Encryption object or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.Encryption.decryptElement

Decrypts the encrypted data DomNode using the symmetric key.

**Syntax**

```csharp
decryptElement (node xml.DomNode )
```

1. `node` defines the encrypted DomNode.

**Usage**

The EncryptedData DomNode node is replaced at the same place in the XML document with the resulting decrypted element DomNode and its children.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.Encryption.decryptElementContent

Decrypts the encrypted data DomNode using the symmetric key.

**Syntax**

```csharp
decryptElementContent (node xml.DomNode )
```

1. `node` defines the encrypted DomNode.

**Usage**

The EncryptedData DomNode node is replaced at the same place in the XML document with the resulting child nodes.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.
Decodes the encrypted data DomNode using the symmetric key, and returns all its children in one new document fragment type xml.DomNode.

Syntax

```
decryptElementContentDetached(
    node xml.DomNode
) RETURNS xml.DomNode
```

1. `node` defines the encrypted DomNode.

Usage

This method decrypts the EncryptedData DomNode referenced by `node` and returns all its children in one new DOCUMENT_FRAGMENT_NODE node.

The resulting child nodes are not added at any place in the XML document. It's up to the user to insert them in the right place, and to remove the encrypted node.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.Encryption.decryptElementDetached

Decrypts the specified encrypted data DomNode using the symmetric key, and returns it in a new element xml.DomNode.

Syntax

```
decryptElementDetached(
    node xml.DomNode
) RETURNS xml.DomNode
```

1. `node` defines the encrypted DomNode.

Usage

This method decrypts the EncryptedData DomNode `node`. The resulting element DomNode and its children are not added at any place in the XML document. It's up to the user to insert it at the right place, and to remove the encrypted node.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.Encryption.decryptKey

Decrypts the EncryptedKey as root in the given XML document, and returns a new CryptoKey of the given kind.

Syntax

```
decryptKey(
    doc xml.DomDocument,
    url STRING
) RETURNS xml.CryptoKeY
```

1. `doc` defines the DomDocument object.
2. `url` defines the string.
Usage
Returns a new CryptoKey of the given kind.
Only symmetric or HMAC keys are allowed.
In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.Encryption.DecryptString
Decrypts an encrypted string encoded in BASE64, using the specified symmetric key, and returns the string in clear text.

Syntax

```plaintext
xml.Encryption.DecryptString(
  key xml.CryptoKey,
  str STRING
) RETURNS STRING
```

1. `key` defines the symmetric key to use for decryption.
2. `str` defines the encrypted string for decryption.

Usage
This method decrypts the encrypted string `str` encoded in BASE64, using the symmetric key specified in `key`, and returns the string in clear text.
The key's usage must be for encryption, see Supported kind of keys on page 3061.
In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.Encryption.encryptElement
Encrypts the element DomNode and all its children using the symmetric key.

Syntax

```plaintext
encryptElement(
  node xml.DomNode
)
```

1. `node` defines the ELEMENT DomNode to encrypt.

Usage
The element DomNode specified in `node` and all its children are replaced at the same place in the XML document with the resulting EncryptedData node.
In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.Encryption.encryptElementContent
Encrypts all child nodes of the element DomNode using the symmetric key.

Syntax

```plaintext
encryptElementContent(
  node xml.DomNode
)
```
1. `node` defines the element `DomNode` to encrypt.

**Usage**

The child nodes of the element `DomNode` specified in `node` are replaced at the same place in the XML document with the resulting `EncryptedData` node.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

```plaintext
xm.Encryption.encryptElementContentDetached
```

Encrypts all child nodes of the element `DomNode` using the symmetric key, and returns them as one new encrypted-data node.

**Syntax**

```plaintext
encryptElementContentDetached(
    node xml.DomNode
) RETURNS xml.DomNode
```

1. `node` defines the ELEMENT `DomNode` to encrypt.

**Usage**

Encrypts all child nodes of the element `DomNode` specified in `node` using the symmetric key, and returns them as one new `EncryptedData` node.

The resulting `DomNode` is not added at any place in the XML document. It's up to the user to insert it at the right place, and to remove the nodes in clear form.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

```plaintext
xm.Encryption.encryptElementDetached
```

Encrypts the element `DomNode` and all its children using the symmetric key, and returns them as one new encrypted-data node.

**Syntax**

```plaintext
encryptElementDetached(
    node xml.DomNode
) RETURNS xml.DomNode
```

1. `node` defines the ELEMENT `DomNode` to encrypt.

**Usage**

Encrypts the element `DomNode` specified in `node` and all its children using the symmetric key, and returns them as one new `EncryptedData` node.

The resulting `DomNode` is not added at any place in the XML document. It's up to the user to insert it at the right place, and to remove the nodes in clear form.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

```plaintext
xm.Encryption.encryptKey
```
Encrypts the given symmetric or HMAC key as an encrypted-key node and returns it as root node of a new XML document.

**Syntax**

```plaintext
encryptKey(
    symkey xml.CryptoKey
) RETURNS xml.DomDocument
```

1. `symkey` defines the given symmetric or HMAC key as an EncryptedKey node.

**Usage**

This method encrypts the symmetric or HMAC key given in `symkey` as an encrypted-key node and returns it as root node of a new XML document. The encryption key must have been set otherwise it will fail.

Depending on the feature set on the key-encryption key, the returned XML document will contain an additional KeyInfo node.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

---

xml.Encryption.EncryptString

Encrypts the specified string using the symmetric key, and returns the encrypted string encoded in BASE64.

**Syntax**

```plaintext
xml.Encryption.EncryptString(
    key xml.CryptoKey,
    str STRING
) RETURNS STRING
```

1. `key` defines the key.
2. `str` defines the string to be encrypted.

**Usage**

This method encrypts the string specified in `str` using the symmetric key defined by `key`, and returns the encrypted string encoded in BASE64.

The key's usage must be for encryption, see Supported kind of keys on page 3061.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

---

xml.Encryption.getEmbeddedKey

Get a copy of the embedded symmetric key that was used in the last decryption operation.

**Syntax**

```plaintext
getEmbeddedKey() RETURNS xml.CryptoKey
```

**Usage**

Returns a copy of the embedded symmetric key that was used in the last decryption operation, or NULL if there is none.
An embedded symmetric key is always encrypted, and needs therefore a key-encryption key to be set in order to
decrypt it.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-
readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls
(STATUS) on page 3554.

xml.Encryption.RSADecrypt
Decysts the BASE64 encrypted string using the RSA key and returns it in clear text

Syntax

```java
xml.Encryption.RSADecrypt (  
    filename STRING,
    str STRING )
RETURNS STRING
```

1. filename defines the file name of a RSA private key in PEM format or an entry in the FGLPROFILE file.
2. str defines a string that was encrypted with the fglpass tool or with the xml.Encryption.RSAEncrypt
   method.

Usage

RSA decryption is only intended for short strings that cannot exceed the size of the RSA key minus 12 bytes. For
instance, if you have a RSA key of 512 bits, your password cannot exceed 512/8-12 = 52 bytes. If you need to handle
big strings, you must use symmetric keys and the DecryptString method. However, you can use RSA keys to
decrypt symmetric key values.

Important: You must ensure that access to the RSA private key file is restricted only to the authorized person or
group of persons.

If the RSA private key is protected with a password, the recommended way is to unprotect it with the openssl
 tool and to put the key file on a restricted file system. But you can also use a script or the fglpass agent to provide the
password to the application.

For example, you can encrypt a database password with the fglpass tool and store it in the FGLPROFILE
file, then you can decrypt it with the base.Application.getResourceEntry and the
xml.Encryption.RSADecrypt method to connect to the database.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-
readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls
(STATUS) on page 3554.

xml.Encryption.RSAEncrypt
Encrypts the specified string using the RSA key and returns it encoded in BASE64.

Syntax

```java
xml.Encryption.RSAEncrypt (  
    filename STRING,
    str STRING )
RETURNS STRING
```

1. filename defines the file name of a RSA public or private key in PEM format or an entry in the FGLPROFILE file.
2. str defines the string to be encrypted.

Usage

RSA encryption is only intended for short strings that cannot exceed the size of the RSA key minus 12 bytes. For
instance, if you have a RSA key of 512 bits, your password cannot exceed 512/8-12 = 52 bytes. If you need to handle
big strings, you must use symmetric keys and the `xml.Encryption.EncryptString` method. However, you can use RSA keys to encrypt symmetric key values.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (`STATUS`) on page 3554.

`xml.Encryption.setCertificate`

Assigns a copy of the X509 certificate to this encryption object.

**Syntax**

```java
setCertificate(
    cert xml.CryptoX509)
```

1. `cert` defines the copy of the X509 certificate.

**Usage**

The certificate will then be added to any further XML document or node encryption.

- NULL is allowed to avoid the certificate being added.
- To encrypt using a certificate, you must use the `createPublicKey` method of the X509 class to obtain the public key embedded in the certificate, and then provide it to the encryption object with `setKeyEncryptionKey` method.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (`STATUS`) on page 3554.

`xml.Encryption.setKey`

Assigns a copy of the symmetric key to this encryption object.

**Syntax**

```java
setKey(
    key xml.CryptoKey)
```

1. `key` defines the symmetric key.

**Usage**

Any further XML document or node encryption or decryption will use that symmetric key.

When decrypting a XML document that has an embedded symmetric key, the embedded key will be used instead.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (`STATUS`) on page 3554.

`xml.Encryption.setKeyEncryptionKey`

Assigns a copy of the key-encryption key to this encryption object.

**Syntax**

```java
setKeyEncryptionKey(
    key xml.CryptoKey)
```

1. `key` defines the key-encryption key.
Usage

Any further XML encryption will use that key-encryption key to encrypt the symmetric key set with $\text{setKey}()$ within the resulting XML, and any further XML decryption will use that key-encryption key to decrypt the embedded symmetric key.

- NULL is allowed, meaning that embedded symmetric keys will not be encrypted nor decrypted anymore, assuming that they have been exchanged in another way.
- Only public or private RSA keys, or key-wrap keys are allowed.
- Public RSA keys can encrypt but not decrypt.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

Examples

xml.Encryption usage examples.

Encrypt a XML node with a symmetric AES128 key

```java
IMPORT xml
MAIN
  DEFINE doc xml.DomDocument
  DEFINE root xml.DomNode
  DEFINE enc xml.Encryption
  DEFINE symkey xml.CryptoKey
  LET doc = xml.DomDocument.Create()
  # Notice that whitespaces are significant in cryptography,
  # therefore it is recommended that you remove unnecessary ones
  CALL doc.setFeature("whitespace-in-element-content",FALSE)
  TRY
    # Load XML file to be encrypted
    CALL doc.load("XMLFileToBeEncrypted.xml")
    LET root = doc.getDocumentElement()
    # Create symmetric AES128 key for XML encryption purposes
    LET symkey = xml.CryptoKey.Create( "http://www.w3.org/2001/04/xmlenc#aes128-cbc")
    CALL symkey.setKey("&secretpassword<") # password of 128 bits
    CALL symKey.setFeature("KeyName","MySecretKey") # Name the password
    # in order to identify the key (Not mandatory)
    # Encrypt the entire document
    LET enc = xml.Encryption.Create()
    CALL enc.setKey(symkey) # Set the symmetric key to be used
    CALL enc.encryptElement(root) # Encrypt
    # Save encrypted document back to disk
    CALL doc.setFeature("format-pretty-print",TRUE)
    CALL doc.save("EncryptedXMLFile.xml")
  CATCH
    DISPLAY "Unable to encrypt XML file :",STATUS
  END TRY
END MAIN
```

Note: All keys or certificates in PEM or DER format were created with the OpenSSL tool.

Decrypt a XML node with a symmetric AES128 key

```java
IMPORT xml
MAIN
  DEFINE doc xml.DomDocument
  DEFINE node xml.DomNode
  DEFINE enc xml.Encryption
  DEFINE symkey xml.CryptoKey
```
DEFINE list xml.DomNodeList
DEFINE str String
LET doc = xml.DomDocument.Create()
# Notice that whitespaces are significant in cryptography, 
# therefore it is recommended to remove unnecessary ones
CALL doc.setFeature("whitespace-in-element-content",FALSE)
TRY
# Load encrypted XML file
CALL doc.load("EncryptedXMLFile.xml")
# Retrieve encrypted node (if any) from the document
LET list = doc.getElementsByTagNameNS( 
   "EncryptedData","http://www.w3.org/2001/04/xmlenc#")
IF list.getCount()==1 THEN 
   LET node = list.getItem(1)
ELSE
   DISPLAY "No encrypted node found"
   EXIT PROGRAM
END IF
# Check if symmetric key name matches the expected "MySecretKey" 
   (Not mandatory)
LET list = node.selectByXPath("dsig:KeyInfo/dsig:KeyName[position()=1 and 
   text()="MySecretKey"]","dsig",http://www.w3.org/2000/09/xmldsig#")
IF list.getCount()!=1 THEN
   DISPLAY "Key name doesn't match"
   EXIT PROGRAM
END IF
# Create symmetric AES128 key for XML decryption purpose
LET symkey = xml.CryptoKey.Create(
   http://www.w3.org/2001/04/xmlenc#aes128-cbc")
CALL symkey.setKey(">secretpassword<") # password of 128 bits
# Decrypt the entire document
LET enc = xml.Encryption.Create()
CALL enc.setKey(symkey) # Set the symmetric key to be used
CALL enc.decryptElement(node) # Decrypt
# Save encrypted document back to disk
CALL doc.setFeature("format-pretty-print",TRUE)
CALL doc.save("DecryptedXMLFile.xml")
CATCH
   DISPLAY "Unable to decrypt XML file 
END TRY
END MAIN

Note: All keys or certificates in PEM or DER format were created with the OpenSSL tool.
Encrypt a XML node with a generated symmetric key protected with the public RSA key within a X509 certificate

IMPORT xml
MAIN
DEFINE doc xml.DomDocument
DEFINE root xml.DomNode
DEFINE enc xml.Encryption
DEFINE symkey xml.CryptoKey
DEFINE kek xml.CryptoKey
DEFINE cert xml.CryptoX509
LET doc = xml.DomDocument.Create()
# Notice that whitespaces are significant in cryptography, 
# therefore it is recommended to remove unnecessary ones
CALL doc.setFeature("whitespace-in-element-content",FALSE)
TRY
# Load XML file to be encrypted
CALL doc.load("XMLFileToBeEncrypted.xml")
LET root = doc.getDocumentElement()
# Load the X509 certificate and retrieve the public RSA key
# for key-encryption purpose
LET cert = xml.CryptoX509.Create()
CALL cert.loadPEM("RSA1024Certificate.crt")
LET kek = cert.createPublicKey(
    "http://www.w3.org/2001/04/xmlenc#rsa-1_5")
# Generate symmetric key for XML encryption purpose
LET symkey = xml.CryptoKey.Create(
    "http://www.w3.org/2001/04/xmlenc#aes256-cbc")
CALL symkey.generateKey(NULL)
# Encrypt the entire document
LET enc = xml.Encryption.Create()
CALL enc.setKey(symkey) # Set the symmetric key to be used
CALL enc.setKeyEncryptionKey(kek) # Set the key-encryption key to
    # be used for protecting the symmetric key
CALL enc.setCertificate(cert) # Set the certificate to be added
    # (not mandatory)
CALL enc.encryptElement(root) # Encrypt
# Save encrypted document back to disk
CALL doc.setFeature("format-pretty-print",TRUE)
CALL doc.save("EncryptedXMLFile.xml")
CATCH
  DISPLAY "Unable to encrypt XML file ",STATUS
END TRY
END MAIN

Note: All keys or certificates in PEM or DER format were created with the OpenSSL tool.

Decrypt a XML node encrypted with a symmetric key protected with a private RSA key

IMPORT xml

MAIN
  DEFINE doc xml.DomDocument
  DEFINE node xml.DomNode
  DEFINE enc xml.Encryption
  DEFINE symkey xml.CryptoKey
  DEFINE kek xml.CryptoKey
  DEFINE list xml.DomNodeList
  LET doc = xml.DomDocument.Create()
  # Notice that whitespaces are significant in cryptography,
  # therefore it is recommended to remove unnecessary ones
  CALL doc.setFeature("whitespace-in-element-content",FALSE)
  TRY
    # Load encrypted XML file
    CALL doc.load("EncryptedXMLFile.xml")
    # Retrieve encrypted node (if any) from the document
    LET list = doc.getElementsByTagNameNS("EncryptedData",
        "http://www.w3.org/2001/04/xmlenc#")
    IF list.getCount()==1 THEN
      LET node = list.getItem(1)
    ELSE
      DISPLAY "No encrypted node found"
      EXIT PROGRAM
    END IF
    # Load the private RSA key
    LET kek = xml.CryptoKey.create(
        "http://www.w3.org/2001/04/xmlenc#rsa-1_5")
    CALL kek.loadPEM("RSA1024Key.pem")
    # Decrypt the entire document
    LET enc = xml.Encryption.Create()
CALL enc.setKeyEncryptionKey(kek) # Set the key-encryption key to decrypted the protected symmetric key
CALL enc.decryptElement(node) # Decrypt
# Retrieve the embedded symmetric key for further usage and display info about it
LET symkey = enc.getEmbeddedKey()
DISPLAY "Key size (in bytes) : ", symkey.getSize() # displays 1024
DISPLAY "Key type : ", symkey.getType() # displays SYMMETRIC
DISPLAY "Key usage : ", symkey.getUsage() # displays ENCRYPTION
# Encrypted document back to disk
CALL doc.setFeature("format-pretty-print", TRUE)
CALL doc.save("DecryptedXMLFile.xml")
CATCH
  DISPLAY "Unable to decrypt XML file : ", STATUS
END TRY
END MAIN

Note: All keys or certificates in PEM or DER format were created with the OpenSSL tool.

The KeyStore class
The xml.KeyStore class provides static methods to handle a key store that is global for the entire application. It enables you to register X509 and trusted certificates, and any kind of key by name for automatic XML signature validation or XML decryption.

The STATUS variable is set to zero after a successful method call.

xml.KeyStore methods
Methods for the xml.KeyStore class.

Table 704: Class methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| xml.KeyStore.AddCertificate(  
  global xml.CryptoX509 ) | Registers in the keystore the given X509 certificate as a certificate for the application. |
| xml.KeyStore.AddKey(  
  global xml.CryptoX509 ) | Registers in the keystore the given key by name for the application. |
| xml.KeyStore.AddTrustedCertificate(  
  global xml.CryptoX509 ) | Registers in the keystore the given X509 certificate as a trusted certificate for the application. |

xml.KeyStore.AddCertificate
Registers in the keystore the given X509 certificate as a certificate for the application.

Syntax

xml.KeyStore.AddCertificate(  
  global xml.CryptoX509 )

1. **global** defines the X509 certificate to register.

Usage

This method registers a given X509 certificate to be used when an incomplete X509 certificate is detected during signature or encryption. To complete the process the certificate’s issuer name and serial number are checked.

The method has the same effect as the FGLPROFILE entry xml.keystore.x509list.
In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.KeyStore.AddKey
Registers in the keystore the given key by name for the application.

Syntax

```xml
xml.KeyStore.AddKey(
    global xml.CryptoX509
)
```

1. `global` defines the key object `xml.CryptoX509` to add to the keystore.

Usage

This method adds a key to the keystore to be used for XML signature verification or XML decryption when a key name has been specified in the XML KeyInfo node and no other key has been set in the Signature or Encryption object.

The CryptoKey must have the KeyName feature set, and the name must be unique in the application.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

xml.KeyStore.AddTrustedCertificate
Registers in the keystore the given X509 certificate as a trusted certificate for the application.

Syntax

```xml
xml.KeyStore.AddTrustedCertificate(
    global xml.CryptoX509
)
```

1. `global` defines the X509 certificate to register.

Usage

This method registers the given X509 certificate as a trusted certificate for the application. It will be used for signature verification if no other certificate was set for that purpose.

The method has the same effect as the FGLPROFILE entry xml.keystore.calist.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

XML transformation classes

The XML transformation classes transform XML.

Important: The XML transformation classes are not supported on GMI mobile devices.

The XSLTransformer class

The `xml.XSLTransformer` class handles XML transformation following the XSLT version 1.0 specification.

Important: This class is not available on iOS, as it is based on libxslt library, which is not allowed on the Apple® store.
xml.XSLTTransformer methods
Methods for the xml.XSLTTransformer class.

Table 705: Class methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xml.XSLTTransformer.CreateFromDocument (doc xml.DomDocument)</td>
<td>Returns a new instance of a XSLTTransformer object to be used to transform a XML document based on a given style sheet.</td>
</tr>
</tbody>
</table>

Table 706: Object methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xml.XSLTTransformer.setParameter (param STRING, value STRING)</td>
<td>Set parameters in an instance of an XSLTTransformer corresponding to the XSLT entries named by param.</td>
</tr>
<tr>
<td>xml.XSLTTransformer.getParameter (param STRING)</td>
<td>Get the value of a given parameter in an instance of an XSLTTransformer corresponding to the XSLT entry named by param.</td>
</tr>
<tr>
<td>xml.XSLTTransformer.clearParameters ()</td>
<td>Clear all parameters in an instance of an XSLTTransformer object.</td>
</tr>
<tr>
<td>xml.XSLTTransformer.doTransform (doc xml.DomDocument)</td>
<td>Transforms the input source based on the XSLT style sheet used to create the XSLTTransformer instance and returns the transformed document.</td>
</tr>
<tr>
<td>xml.XSLTTransformer.getErrorDescription (index INTEGER)</td>
<td>Returns the exact description of the error referenced by the index.</td>
</tr>
<tr>
<td>xml.XSLTTransformer.getErrorsCount ()</td>
<td>Returns the number of errors.</td>
</tr>
</tbody>
</table>

xml.XSLTTransformer.clearParameters
Clear all parameters in an instance of an XSLTTransformer object.

Syntax

xml.XSLTTransformer.clearParameters()  

Usage

Clears all parameters corresponding to the XSLT entries named by param.

For example, this XSLT sample contains two param definitions:

```xml
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" >
```
In case of error, the method throws an exception and sets the \texttt{STATUS} variable. Depending on the error, a human-readable description of the problem is available in the \texttt{SQLCA.SQLERRM} register. See Error handling in GWS calls (\texttt{STATUS}) on page 3554.

\textbf{Related concepts}

\texttt{xml.XSLTTransformer.getParameter} on page 3123
Get the value of a given parameter in an instance of an \texttt{XSLTTransformer} corresponding to the XSLT entry named by \texttt{param}.

\texttt{xml.XSLTTransformer.setParameter} on page 3122
Set parameters in an instance of an \texttt{XSLTTransformer} corresponding to the XSLT entries named by \texttt{param}.

\texttt{xml.XSLTTransformer.CreateFromDocument}
Returns a new instance of a \texttt{XSLTTransformer} object to be used to transform a XML document based on a given style sheet.

\textbf{Syntax}

\begin{verbatim}
xml.XSLTTransformer.CreateFromDocument (  
doctor xml.DomDocument  )  
RETURNS xml.XSLTTransformer
\end{verbatim}

1. \texttt{doc} defines the given style sheet.

\textbf{Usage}

Returns a new instance of an \texttt{XSLTTransformer} object to be used to transform a XML document based on the given style sheet. It may raise a \texttt{XML_ERROR_LOAD_EXCEPTION} exception if the XML document is not loadable.

There may be non-critical errors when loading a given style sheet document; you can check with the \texttt{xml.XSLTTransformer.getErrorsCount()} method.

In case of error, the method throws an exception and sets the \texttt{STATUS} variable. Depending on the error, a human-readable description of the problem is available in the \texttt{SQLCA.SQLERRM} register. See Error handling in GWS calls (\texttt{STATUS}) on page 3554.

To see these methods in a working example, see Example: Using xml.XSLTTransformer methods.

\texttt{xml.XSLTTransformer.doTransform}
Transforms the input source based on the XSLT style sheet used to create the \texttt{XSLTTransformer} instance and returns the transformed document.

\textbf{Syntax}

\begin{verbatim}
xml.XSLTTransformer.doTransform (  
doctor xml.DomDocument  )  
RETURNS xml.DomDocument
\end{verbatim}
1. *doc* defines the input source.

**Usage**

This method transforms the input source based on the XSLT style sheet used to create the XSLTTransformer instance and returns the transformed document. It raises a `XML_ERROR_FAILED` exception in case of critical error. The operation may return a new document that may be incomplete. You must check for errors that may occur during the transformation with the `xml.XSLTTransformer.getErrorsCount()` method.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

To see these methods in a working example, see Example: Using xml.XSLTTransformer methods.

**xml.XSLTTransformer.getErrorDescription**

Returns the exact description of the error referenced by the index.

**Syntax**

```java
xml.XSLTTransformer.getErrorDescription(
    index INTEGER
) RETURNS STRING
```

1. `index` defines the index of the error returned by `xml.XSLTTransformer.getErrorsCount()`.

**Usage**

This method returns the exact description of the error referenced by the index during the `doTransform()` call or during the `CreateFromDocument()` when the style sheet contains some errors.

This method is typically used in conjunction with `getErrorsCount()`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

To see these methods in a working example, see Example: Using xml.XSLTTransformer methods.

**Related concepts**

- `xml.XSLTTransformer.CreateFromDocument` on page 3120
  Returns a new instance of a XSLTTransformer object to be used to transform a XML document based on a given style sheet.

- `xml.XSLTTransformer.doTransform` on page 3120
  Transforms the input source based on the XSLT style sheet used to create the XSLTTransformer instance and returns the transformed document.

**xml.XSLTTransformer.getErrorsCount**

Returns the number of errors.

**Syntax**

```java
xml.XSLTTransformer.getErrorsCount()
RETURNS INTEGER
```

**Usage**

Return the number of errors during the `doTransform()` call or during the `CreateFromDocument()` when the style sheet contains errors.
It is recommended to check if errors have been encountered after both operations, because even with errors, it is possible to transform a document.

The errors count is reset between CreateFromDocument() and doTransform().

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

To see these methods in a working example, see Example: Using xml.XSLTTransformer methods.

Related concepts
xml.XSLTTransformer.getErrorDescription on page 3121
Returns the exact description of the error referenced by the index.

xml.XSLTTransformer.CreateFromDocument on page 3120
Returns a new instance of a XSLTTransformer object to be used to transform a XML document based on a given style sheet.

xml.XSLTTransformer.doTransform on page 3120
Transforms the input source based on the XSLT style sheet used to create the XSLTTransformer instance and returns the transformed document.

xml.XSLTTransformer.setParameter
Set parameters in an instance of an XSLTTransformer corresponding to the XSLT entries named by param.

Syntax

```
xm1.XSLTTransformer.setParameter(
    param STRING,
    value STRING )
```

1. param defines the name of the parameter.
2. value specifies the value to set the parameter.

Usage

Set parameters corresponding to the XSLT entries named by param.

For example, this XSLT sample contains two param definitions:

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" >

    <xsl:param name="QADIR" />
    <xsl:param name="NODE_NAME"/>
    ...
    <MyNode><xsl:value-of select="$QADIR" /></MyNode>
    ...
    <xsl:when test="name()=$NODE_NAME">
        <xsl:copy>
            <xsl:copy-of select="@*"/>
        </xsl:copy>
    </xsl:when>

</xsl:stylesheet>
```

To set the parameter QADIR:

```
CALL xslt.setParameter("QADIR","" | "/work/tmp" | "") # Literal
```

Note: If a parameter is a literal, you must quote it before calling the setParameter()
To set the parameter \texttt{NODE\_NAME}:

\begin{verbatim}
CALL xslt.setParameter("NODE\_NAME","BOOK")
\end{verbatim}

If the parameter is already set, the previous value will be replaced by the new one.

In case of error, the method throws an exception and sets the \texttt{STATUS} variable. Depending on the error, a human-readable description of the problem is available in the \texttt{SQLCA.SQLERRM} register. See \textit{Error handling in GWS calls (STATUS)} on page 3554.

\textbf{Related concepts}

\texttt{xml.XSLTransformer.getParameter} on page 3123
Get the value of a given parameter in an instance of an \texttt{XSLTransformer} corresponding to the XSLT entry named by \texttt{param}.

\texttt{xml.XSLTransformer.clearParameters} on page 3119
Clear all parameters in an instance of an \texttt{XSLTransformer} object.

\texttt{xml.XSLTransformer.getParameter}
Get the value of a given parameter in an instance of an \texttt{XSLTransformer} corresponding to the XSLT entry named by \texttt{param}.

\textbf{Syntax}

\begin{verbatim}
xml.XSLTransformer.getParameter(
    param STRING)
RETURNS STRING
\end{verbatim}

1. \texttt{param} defines the name of the parameter.

\textbf{Usage}

Get parameters corresponding to the XSLT entries named by \texttt{param}.

For example, this XSLT sample contains two \texttt{param} definitions:

\begin{verbatim}
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" >
  <xsl:param name="QADIR" />
  <xsl:param name="NODE\_NAME"/>
  ...
  <MyNode><xsl:value-of select="$QADIR" /></MyNode>
  ...
  <xsl:when test="name()=$NODE\_NAME">  
      <xsl:copy>
        <xsl:copy-of select="@*"/>
    </xsl:copy>
  </xsl:when>
</xsl:stylesheet>
\end{verbatim}

To get the parameter \texttt{QADIR}:

\begin{verbatim}
DEFINE dir STRING
SET dir=xslt.getParameter("QADIR")
\end{verbatim}

In case of error, the method throws an exception and sets the \texttt{STATUS} variable. Depending on the error, a human-readable description of the problem is available in the \texttt{SQLCA.SQLERRM} register. See \textit{Error handling in GWS calls (STATUS)} on page 3554.
To see these methods in a working example, see Example: Using xml.XSLTTransformer methods.

**Related concepts**

xml.XSLTTransformer.setParameter on page 3122

Set parameters in an instance of an XSLTTransformer corresponding to the XSLT entries named by param.

xml.XSLTTransformer.clearParameters on page 3119

Clear all parameters in an instance of an XSLTTransformer object.

**Example: Using xml.XSLTTransformer methods**

This Genero application provides a working example using methods from the XSLTTransformer class.

```genero
IMPORT XML

MAIN
   DEFINE ok BOOLEAN
   IF num_args() != 3 THEN
      DISPLAY "Usage : DoXslp <stylesheet> <source> <result>"
      EXIT PROGRAM 1
   ELSE
      LET ok = RunXSLP(arg_val(1), arg_val(2), arg_val(3))
      IF NOT ok THEN
         DISPLAY "Error: failed"
         EXIT PROGRAM 1
      ELSE
         DISPLAY "Done"
         EXIT PROGRAM
      END IF
   END IF
END MAIN

FUNCTION RunXSLP(style, src, ret)
   DEFINE style, src, ret STRING
   DEFINE ind INTEGER
   DEFINE xslt xml.XSLTTransformer
   DEFINE styleSheet xml.DomDocument
   DEFINE source xml.DomDocument
   DEFINE result xml.DomDocument

   # Load StyleSheet
   TRY
      LET styleSheet = xml.DomDocument.Create()
      CALL styleSheet.load(style)
   CATCH
      DISPLAY "Error: unable to load stylesheet", style
      RETURN FALSE
   END TRY

   # Create XSLT transformer
   TRY
      LET xslt = xml.XSLTTransformer.CreateFromDocument(styleSheet)
      FOR ind = 1 TO xslt.getErrorsCount()
         DISPLAY "StyleSheet error #" || ind || " : ", xslt.getErrorDescription(ind)
      END FOR
   CATCH
      DISPLAY "Error : unable to create XSLT transformer from ", styleSheet
      RETURN FALSE
   END TRY

   # Load Source
   TRY
      LET source = xml.DomDocument.Create()
      CALL source.load(src)
   CATCH
      DISPLAY "Error: unable to load source", src
      RETURN FALSE
   END TRY
```

```javascript
import XML

class MAIN {
   ok BOOLEAN
   IF num_args() != 3 THEN
      DISPLAY "Usage : DoXslp <stylesheet> <source> <result>"
      EXIT PROGRAM 1
   ELSE
      ok = RunXSLP(arg_val(1), arg_val(2), arg_val(3))
      IF NOT ok THEN
         DISPLAY "Error: failed"
         EXIT PROGRAM 1
      ELSE
         DISPLAY "Done"
         EXIT PROGRAM
      END IF
   END IF
}

class RunXSLP {
   style, src, ret STRING
   ind INTEGER
   xslt xml.XSLTTransformer
   styleSheet xml.DomDocument
   source xml.DomDocument
   result xml.DomDocument

   # Load StyleSheet
   TRY
      styleSheet = xml.DomDocument.Create()
      styleSheet.load(style)
   CATCH
      DISPLAY "Error: unable to load stylesheet", style
      RETURN false
   END TRY

   # Create XSLT transformer
   TRY
      xslt = xml.XSLTTransformer.CreateFromDocument(styleSheet)
      FOR ind = 1 TO xslt.getErrorsCount()
         DISPLAY "StyleSheet error #" || ind || " : ", xslt.getErrorDescription(ind)
      END FOR
   CATCH
      DISPLAY "Error : unable to create XSLT transformer from ", styleSheet
      RETURN false
   END TRY

   # Load Source
   TRY
      source = xml.DomDocument.Create()
      source.load(src)
   CATCH
      DISPLAY "Error: unable to load source", src
      RETURN false
   END TRY
```
OM to XML Migration

Code using the OM classes can be converted to XML classes in most cases.

The build-in om package provides basic XML handling. The Web Services extensions xml package provides full support for XML document processing. You may need to change code from om to xml classes. Before you convert code, make sure that you are using the classes from the appropriate package:

- Classes from the om package exist to manipulate the AUI tree and also provide basic XML handling.
- Classes from the xml package provides classes and methods to handle any kind of XML document processing, and are recommended for XML document manipulation.

Why would you migrate from om to xml classes and methods?

- You need to be able to utilize a feature (such as a StyleSheet) that requires use of methods from the xml library classes.

Note: The DOM API of the om package is designed to handle specific FGL files and to manipulate the user interface tree (the AUI tree). For all other cases, we recommend that you use the DOM API of the Web Services xml package.

OM - XML Mapping

A reference guide to the DOM APIs of the om and xml classes.

Table 707: OM - XML Mapping

<table>
<thead>
<tr>
<th>OM class method</th>
<th>XML class method(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>om.DomDocument.copy</td>
<td>xml.DomNode.clone</td>
</tr>
<tr>
<td>OM class method</td>
<td>XML class method(s)</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>om.DomDocument.getElementById</td>
<td>xml.DomDocument.getElementById + xmlDOMNode.setIdAttribute or xmlDOMNode.setIdAttributeNS</td>
</tr>
<tr>
<td>omDOMNode.appendChild</td>
<td>xmlDOMNode.appendChild</td>
</tr>
<tr>
<td>omDOMNode.createChild</td>
<td>xmlDOMNode.appendChildElement</td>
</tr>
<tr>
<td>omDOMNode.insertBefore</td>
<td>xmlDOMNode.insertBeforeChild</td>
</tr>
<tr>
<td>omDOMNode.removeChild</td>
<td>xmlDOMNode.removeChild</td>
</tr>
<tr>
<td>omDOMNode.replaceChild</td>
<td>xmlDOMNode.replaceChild</td>
</tr>
<tr>
<td>omDOMNode.loadXml</td>
<td>xmlDOMNode.loadFromNode + add it to the DomDocument</td>
</tr>
<tr>
<td>omDOMNode.parse</td>
<td></td>
</tr>
<tr>
<td>omDOMNode.toString</td>
<td>xmlDOMNode.toString</td>
</tr>
<tr>
<td>omDOMNode.writeXml</td>
<td>xmlDOMNode.save</td>
</tr>
<tr>
<td>omDOMNode.write</td>
<td>xmlDOMNode toString</td>
</tr>
<tr>
<td>omDOMNode.getId</td>
<td>N/A</td>
</tr>
<tr>
<td>omDOMNode.getTagRoot</td>
<td>xmlDOMNode.getLocalName</td>
</tr>
<tr>
<td>omDOMNode.setAttribute</td>
<td>xmlDOMNode.setAttribute</td>
</tr>
<tr>
<td>omDOMNode.getAttribute</td>
<td>xmlDOMNode.getAttribute</td>
</tr>
<tr>
<td>omDOMNode.getAttributeInteger</td>
<td>xmlDOMNode.getAttribute + condition for the default value and the cast</td>
</tr>
<tr>
<td>omDOMNode.getAttributeString</td>
<td>xmlDOMNode.getAttribute + condition for the default value and the cast</td>
</tr>
<tr>
<td>omDOMNode.getAttributeName</td>
<td>xmlDOMNode.getAttributeNodeItem + xmlDOMNode.getLocalName</td>
</tr>
<tr>
<td>omDOMNode.getAttributeCount</td>
<td>xmlDOMNode.getAttributeCount</td>
</tr>
<tr>
<td>omDOMNode.getAttributeValue</td>
<td>xmlDOMNode.getAttributeNodeItem + xmlDOMNode.getNodeValue</td>
</tr>
<tr>
<td>omDOMNode.removeAttribute</td>
<td>xmlDOMNode.removeAttribute</td>
</tr>
<tr>
<td>omDOMNode.getChildCount</td>
<td>xmlDOMNode.getChildrenCount</td>
</tr>
<tr>
<td>omDOMNode.getChildByIndex</td>
<td>xmlDOMNode.getChildNodeItem</td>
</tr>
<tr>
<td>omDOMNode.getFirstChild</td>
<td>xmlDOMNode.getFirstChild</td>
</tr>
<tr>
<td>omDOMNode.getLastChild</td>
<td>xmlDOMNode.getLastChild</td>
</tr>
<tr>
<td>omDOMNode.getNext</td>
<td>xmlDOMNode.getNextSibling</td>
</tr>
<tr>
<td>OM class method</td>
<td>XML class method(s)</td>
</tr>
<tr>
<td>-------------------------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>om.DomNode.getPrevious</td>
<td>xml.DomNode.getPreviousSibling</td>
</tr>
<tr>
<td>om.DomNode.selectByTag</td>
<td>xml.DomNode.getElementsByTagName</td>
</tr>
<tr>
<td>om.DomNode.selectByPath</td>
<td>xml.DomNode.selectByXPath</td>
</tr>
<tr>
<td>om.NodeList.item</td>
<td>xml.DomNodeList.getItem</td>
</tr>
<tr>
<td>om.NodeList.getLength</td>
<td>xml.DomNodeList.getCount</td>
</tr>
</tbody>
</table>

For more information on Genero built-in classes (such as the OM class), refer to Built-in packages on page 2378.

The security package

The Genero Web Services security package provides classes and methods to support basic cryptographic features.

Use the IMPORT statement at the top of the module using this library:

```fortran
IMPORT security
```

The RandomGenerator class

The security.RandomGenerator class includes methods for creating random strings or numbers.

security.RandomGenerator methods

Methods of the security.RandomGenerator class.

Table 708: Class methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>security.RandomGenerator.CreateRandomNumber()</td>
<td>Generates an 8-byte strong random number.</td>
</tr>
<tr>
<td>security.RandomGenerator.CreateRandomString( size INTEGER ) RETURNS STRING</td>
<td>Creates a random base64 string.</td>
</tr>
</tbody>
</table>

**security.RandomGenerator.CreateRandomNumber**

Generates an 8-byte strong random number.

**Syntax**

```fortran
security.RandomGenerator.CreateRandomNumber() RETURNS BIGINT
```

**Usage**

This method generates an 8-byte strong random number and returns it as a BIGINT. The generated number can then be used for advanced cryptographic features.
This method is based on OpenSSL, using `/dev/random` on UNIX® and CryptGenRandom() on Microsoft® Windows®, which are based on the cryptographically secure pseudo-random number generator (CSPRNG) specifications.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

security.RandomGenerator.CreateRandomString
Creates a random base64 string.

Syntax

```
security.RandomGenerator.CreateRandomString(
    size INTEGER )
RETURNS STRING
```

1. `size` defines the size of the random string.

Usage

This method generates a random string of binary data of `size` bytes long and returns it in a STRING encoded in a Base64 form.

The size must be greater than 0.

Use this function when randomness is required, such as in xml.CryptoKey.deriveKey() or security.Digest.CreateDigestString().

This method is based on OpenSSL, using `/dev/random` on UNIX® and CryptGenRandom() on Microsoft® Windows®, which are based on the cryptographically secure pseudo-random number generator (CSPRNG) specifications.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

security.RandomGenerator.CreateUUIDString
Creates a new universal unique identifier (UUID).

Syntax

```
security.RandomGenerator.CreateUUIDString()
RETURNS STRING
```

Usage

This method generates an universal unique identifier and returns the value as STRING.

The generated string follows the UUID version 4 specification. Version 4 UUIDs have the form `xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx` where `x` is any hexadecimal digit and `y` is one of 8, 9, A, or B.

Note: This method replaces com.Util.CreateUUIDString().

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.
The Base64 class
The `security.Base64` class includes methods for encoding to base64 or decoding from base64.

security.Base64 methods
Methods of the `security.Base64` class.
Table 709: Class methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>security.Base64.FromByte( data BYTE ) RETURNS STRING</td>
<td>Encodes the given BYTE data in base64.</td>
</tr>
<tr>
<td>security.Base64.FromHexBinary( hexBinVal STRING ) RETURNS STRING</td>
<td>Decodes the given hexadecimal string to base64.</td>
</tr>
<tr>
<td>security.Base64.FromString( clearVal STRING ) RETURNS STRING</td>
<td>Encodes the given string in base64.</td>
</tr>
<tr>
<td>security.Base64.FromStringWithCharset( clearVal STRING, charset STRING ) RETURNS STRING</td>
<td>Encodes the given string in base64, based on a given charset.</td>
</tr>
<tr>
<td>security.Base64.LoadBinary( path STRING ) RETURNS STRING</td>
<td>Reads data from a file and encodes to base64.</td>
</tr>
<tr>
<td>security.Base64.SaveBinary( path STRING, base64Data STRING )</td>
<td>Decodes the given base64 string and writes the data to a file.</td>
</tr>
<tr>
<td>security.Base64.ToByte( val64 STRING, ret BYTE )</td>
<td>Decodes the given base64 string into a BYTE.</td>
</tr>
<tr>
<td>security.Base64.ToHexBinary( val64 STRING ) RETURNS STRING</td>
<td>Decodes the given base64 string to hexadecimal.</td>
</tr>
<tr>
<td>security.Base64.ToString( val64 STRING ) RETURNS STRING</td>
<td>Decodes the given base64 string.</td>
</tr>
<tr>
<td>security.Base64.ToStringWithCharset( val64 STRING, charset STRING ) RETURNS STRING</td>
<td>Decodes the given base64 string, based on a given charset.</td>
</tr>
<tr>
<td>security.Base64.Xor( clearVal1 STRING, clearVal2 STRING ) RETURNS STRING</td>
<td>Computes the exclusive disjunction between two base64 encoded strings.</td>
</tr>
</tbody>
</table>
security.Base64.FromByte
Encodes the given BYTE data in base64.

Syntax

```c
security.Base64.FromByte(
    data BYTE
) RETURNS STRING
```

1. `data` defines the data of type BYTE to be encoded.

Usage

This method encodes the given BYTE data in base64 and returns the string.

**Important:** The BYTE must be located in memory.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

security.Base64.FromHexBinary
Decodes the given hexadecimal string to base64.

Syntax

```c
security.Base64.FromHexBinary(
    hexBinVal STRING
) RETURNS STRING
```

1. `hexBinVal` defines the string in hexadecimal form.

Usage

This method decodes the given hexadecimal string and returns it in base64.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

security.Base64.FromString
Encodes the given string in base64.

Syntax

```c
security.Base64.FromString(
    clearVal STRING
) RETURNS STRING
```

1. `clearVal` defines the string to be encoded.

Usage

This method encodes the given string, `clearVal`, and returns it in base64.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.
**security.Base64.FromStringWithCharset**

Encodes the given string in base64, based on a given charset.

**Syntax**

```java
security.Base64.FromStringWithCharset(
    clearVal STRING,
    charset STRING
) RETURNS STRING
```

1. *clearVal* defines the string to be encoded.
2. *charset* defines the character set to be used.

**Usage**

This method encodes the string, *clearVal*, based on the specified charset, *charset*, and returns it in base64.

Before conversion, the string is converted from the local DVM charset to the specified encoding.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**security.Base64.LoadBinary**

Reads data from a file and encodes to base64.

**Syntax**

```java
security.Base64.LoadBinary(
    path STRING
) RETURNS STRING
```

1. *path* defines the path to the binary file.

**Usage**

Reads the file located at *path* and encodes this binary data in Base64 format.

For example, this method can be used to send images through a network.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**security.Base64.SaveBinary**

Decodes the given base64 string and writes the data to a file.

**Syntax**

```java
security.Base64.SaveBinary(
    path STRING,
    base64Data STRING
)
```

1. *path* defines the path to the binary file.
2. *base64Data* defines base64 string to be written.

**Usage**

Decodes the given Base64 string and writes the binary data to the file defined by *path*. 
This method can be used to save data from a network on the disk.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**security.Base64.ToHexBinary**

Decodes the given base64 string to hexadecimal.

**Syntax**

```plaintext
security.Base64.ToHexBinary ( 
    val64 STRING )
RETURNS STRING
```

1. `val64` defines a string encoded in base64.

**Usage**

Decodes the base64 string, `val64`, and returns it in its hexadecimal form.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**security.Base64.ToByte**

Decodes the given base64 string into a BYTE.

**Syntax**

```plaintext
security.Base64.ToByte ( 
    val64 STRING, 
    ret BYTE )
```

1. `val64` defines the string in base64.
2. `ret` defines a parameter of type BYTE to fill with data.

**Usage**

Decodes the base64 string specified in `val64` and fills the BYTE variable with binary data.

**Important:** The BYTE must be located in memory.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**security.Base64.ToString**

Decodes the given base64 string.

**Syntax**

```plaintext
security.Base64.ToString ( 
    val64 STRING )
RETURNS STRING
```

1. `val64` defines the string in base64.
Usage
Decodes the given base64 string and returns it in its clear (human readable) form.
If the base64 string does not contain human readable data, the method will raise an exception.
If the base64 string contains bytes sequences that do not match a valid character in the current encoding, the method raises a conversion error.
In case of error, the method throws an exception and sets the \texttt{STATUS} variable. Depending on the error, a human-readable description of the problem is available in the \texttt{SQLCA.SQLERRM} register. See \textit{Error handling in GWS calls (STATUS)} on page 3554.

\texttt{security.Base64.ToStringWithCharset}
Decodes the given base64 string, based on a given charset.

Syntax

\begin{verbatim}
security.Base64.ToStringWithCharset ( 
    val64 STRING,
    charset STRING )
RETURNS STRING
\end{verbatim}

1. \texttt{val64} defines the string in base64.
2. \texttt{charset} defines the character set to be used.

Usage
Decodes the given base64 string and returns it in its clear human readable form, based on a given charset.
The original base64 encoded string is first decoded to a string that will be converted from the specified charset to the local DVM charset. In case of charset conversion error, the error -15700 is raised.
If the base64 string does not contain human readable data, the method will raise an exception.
If the base64 string contains bytes sequences that do not match a valid character in the current encoding, the method raises a conversion error.
In case of error, the method throws an exception and sets the \texttt{STATUS} variable. Depending on the error, a human-readable description of the problem is available in the \texttt{SQLCA.SQLERRM} register. See \textit{Error handling in GWS calls (STATUS)} on page 3554.

\texttt{security.Base64.Xor}
Computes the exclusive disjunction between two base64 encoded strings.

Syntax

\begin{verbatim}
security.Base64.Xor ( 
    clearVal1 STRING,
    clearVal2 STRING )
RETURNS STRING
\end{verbatim}

1. \texttt{clearVal1} defines the first string encoded in base64.
2. \texttt{clearVal2} defines the second string encoded in base64.

Usage
Decodes the two given strings and does an exclusive disjunction between the two binary inputs. The result is returned encoded in base64.
In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.
**The HexBinary class**

The `security.HexBinary` class includes methods for encoding to hexadecimal or decoding from hexadecimal.

**security.HexBinary methods**

Methods of the `security.HexBinary` class.
### Table 710: Class methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>security.HexBinary.FromBase64(val64 STRING)</code></td>
<td>Converts a base64 string to the hexadecimal equivalent.</td>
</tr>
<tr>
<td><code>security.HexBinary.FromByte(data BYTE)</code></td>
<td>Encodes BYTE data in hexadecimal.</td>
</tr>
<tr>
<td><code>security.HexBinary.FromString(clearVal STRING)</code></td>
<td>Encodes a given string in hexadecimal.</td>
</tr>
<tr>
<td><code>security.HexBinary.FromStringWithCharset(clearVal STRING, charset STRING)</code></td>
<td>Encodes a given string in hexadecimal, based on a given charset.</td>
</tr>
<tr>
<td><code>security.HexBinary.LoadBinary(path STRING)</code></td>
<td>Reads binary data from a file and converts it to hexadecimal.</td>
</tr>
<tr>
<td><code>security.HexBinary.SaveBinary(path STRING, hexBinData STRING)</code></td>
<td>Decodes an hexadecimal strings and writes the binary data to a file.</td>
</tr>
<tr>
<td><code>security.HexBinary.ToBase64(hexBinVal STRING)</code></td>
<td>Converts an hexadecimal string to the base64 equivalent.</td>
</tr>
<tr>
<td><code>security.HexBinary.ToByte(hex STRING, ret BYTE)</code></td>
<td>Decodes an hexadecimal string into a BYTE variable.</td>
</tr>
<tr>
<td><code>security.HexBinary.ToString(hexVal STRING)</code></td>
<td>Decodes an hexadecimal string to a clear, human-readable string.</td>
</tr>
<tr>
<td><code>security.HexBinary.ToStringWithCharset(hexVal STRING, charset STRING)</code></td>
<td>Decodes an hexadecimal string to a clear, human-readable string, based on a given charset.</td>
</tr>
<tr>
<td><code>security.HexBinary.Xor(hexVal1 STRING, hexVal2 STRING)</code></td>
<td>Computes the exclusive disjunction between two hexadecimal encoded strings.</td>
</tr>
</tbody>
</table>
security.HexBinary.FromBase64
Converts a base64 string to the hexadecimal equivalent.

Syntax

```java
security.HexBinary.FromBase64(
    val64 STRING
) RETURNS STRING
```

1. `val64` defines a string encoded in base64.

Usage
This method decodes the given base64 string and returns it in its hexadecimal form.
In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

security.HexBinary.FromByte
Encodes BYTE data in hexadecimal.

Syntax

```java
security.HexBinary.FromByte(
    data BYTE
) RETURNS STRING
```

1. `data` defines the data of type BYTE to be encoded.

Usage
This method encodes the given BYTE data in hexadecimal and returns the string.

Important: The BYTE must be located in memory.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

security.HexBinary.FromString
Encodes a given string in hexadecimal.

Syntax

```java
security.HexBinary.FromString(
    clearVal STRING
) RETURNS STRING
```

1. `clearVal` defines the string to be encoded.

Usage
Encodes the given string and returns it in hexadecimal.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.
**security.HexBinary.FromStringWithCharset**

Encodes a given string in hexadecimal, based on a given charset.

**Syntax**

```csharp
security.HexBinary.FromStringWithCharset(
    clearVal STRING,
    charset STRING )
RETURNS STRING
```

1. `clearVal` defines the string to be encoded.
2. `charset` defines the character set to be used.

**Usage**

Encodes the given string and returns it in hexadecimal.

Before conversion, the string is converted from the local DVM charset to the specified encoding.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**security.HexBinary.LoadBinary**

Reads binary data from a file and converts it to hexadecimal.

**Syntax**

```csharp
security.HexBinary.LoadBinary( 
    path STRING )
RETURNS STRING
```

1. `path` defines the path to the binary file.

**Usage**

Reads the file located at `path` and returns this binary data in hexadecimal format.

For example, this method can be used to send images through a network.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**security.HexBinary.SaveBinary**

Decodes an hexadecimal string and writes the binary data to a file.

**Syntax**

```csharp
security.HexBinary.SaveBinary(
    path STRING,
    hexBinData STRING )
```

1. `path` defines the path to the binary file.
2. `hexBinData` defines the hexadecimal string to be written.

**Usage**

Decodes the given hexadecimal string and writes the binary data to the file defined by `path`. 
This method can be used to save data from a network on the disk.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

`security.HexBinary.ToBase64`
Converts an hexadecimal string to the base64 equivalent

**Syntax**

```csharp
security.HexBinary.ToBase64(
    hexBinVal STRING )
RETURNS STRING
```

1. `hexBinVal` defines the string in hexadecimal form

**Usage**
Decodes the given hexadecimal string and returns it in base64.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

`security.HexBinary.ToByte`
Decodes an hexadecimal string into a BYTE variable.

**Syntax**

```csharp
security.HexBinary.ToByte(
    hex STRING,
    ret BYTE )
```

1. `hex` defines a string in hexadecimal.
2. `ret` defines the input parameter to fill with data of type BYTE.

**Usage**
Decodes the given hexadecimal string and fills the BYTE variable with binary data.

**Important:** The BYTE must be located in memory.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

`security.HexBinary.ToString`
Decodes an hexadecimal string to a clear, human-readable string.

**Syntax**

```csharp
security.HexBinary.ToString(
    hexVal STRING )
RETURNS STRING
```

1. `hexVal` defines the string in hexadecimal.
**Usage**

Decodes the given hexadecimal string and returns it in its clear, human-readable form. If the hexadecimal string does not contain a human-readable string, the method will raise an exception.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

`security.HexBinary.ToStringWithCharset`

Decodes an hexadecimal string to a clear, human-readable string, based on a given charset.

**Syntax**

```c
security.HexBinary.ToStringWithCharset ( 
    hexVal STRING, 
    charset STRING )
RETURNS STRING
```

1. `hexVal` defines the string in hexadecimal.
2. `charset` defines the character set to be used.

**Usage**

Decodes the given hexadecimal string and returns it in its clear human-readable form, based on a given charset.

The original hexadecimal encoded string is first decoded to a string that will then be converted from the specified charset to the local DVM charset. In case of charset conversion error, the error -15700 is raised.

If the hexadecimal string does not contain a human-readable string, the method will raise an exception.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

`security.HexBinary.Xor`

Computes the exclusive disjunction between two hexadecimal encoded strings.

**Syntax**

```c
security.HexBinary.Xor ( 
    hexVal1 STRING, 
    hexVal2 STRING )
RETURNS STRING
```

1. `hexVal1` defines the first string in hexadecimal.
2. `hexVal2` defines the second string in hexadecimal.

**Usage**

Decodes the two given strings and does an exclusive disjunction between the two binary inputs. The result is returned in hexadecimal.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.
The Digest class

The `security.Digest` class implements digest algorithms to process data.

The class implements several methods that allow you to add data piece by piece and process these data with a specified digest algorithm.

Steps to process data with a digest algorithm:

1. Define the digest algorithm with the `security.Digest.CreateDigest` method.
3. When all data pieces are added, the buffer can be processed by calling methods like `security.Digest.DoBase64Digest` or `security.Digest.DoHexBinaryDigest`.

Alternatively, a simple data string can be processed with the `security.Digest.CreateDigestString` method.

**security.Digest methods**

Methods of the `security.Digest` class.

**Table 711: Class methods**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>security.Digest.CreateDigest (algo STRING) RETURNS security.Digest</code></td>
<td>Defines a new digest context by specifying the algorithm to be used.</td>
</tr>
</tbody>
</table>
### Table 712: Object methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>security.Digest.AddData (toDigest BYTE)</code></td>
<td>Adds data from a BYTE variable to the digest buffer.</td>
</tr>
<tr>
<td><code>security.Digest&gt;AddBase64Data (toDigest STRING)</code></td>
<td>Adds data in base64 format to the digest buffer.</td>
</tr>
<tr>
<td><code>security.Digest&gt;AddHexBinaryData (toDigest STRING)</code></td>
<td>Adds data in hexadecimal format to the digest buffer.</td>
</tr>
<tr>
<td><code>security.Digest&gt;AddStringData (toDigest STRING)</code></td>
<td>Adds a data string to the digest buffer.</td>
</tr>
<tr>
<td><code>security.Digest&gt;AddStringDataWithCharset (toDigest STRING, charset STRING)</code></td>
<td>Adds a data string to the digest buffer, after converting to the specified character set.</td>
</tr>
<tr>
<td><code>security.Digest.CreateDigestString (toDigest STRING, randomBase64 STRING)</code></td>
<td>Creates a SHA1 digest from the given string.</td>
</tr>
<tr>
<td><code>security.Digest.DoBase64Digest ()</code></td>
<td>Creates a digest of the buffered data and returns the result in base64 format.</td>
</tr>
<tr>
<td><code>security.Digest.DoHexBinaryDigest ()</code></td>
<td>Creates a digest of the buffered data and returns the result in hexadecimal format.</td>
</tr>
</tbody>
</table>

**security.Digest.AddData**  
Adds data from a BYTE variable to the digest buffer.

**Syntax**

```plaintext
security.Digest.AddData ( 
    toDigest BYTE )
```

1. `toDigest` defines the binary data to be added to the digest buffer.

**Usage**

Adds the binary data contained in the given BYTE to the digest context.

After adding all the data pieces, the buffer can be processed by calling `security.Digest.DoBase64Digest` or `security.Digest.DoHexBinaryDigest`.

**Important:** The BYTE must be located in memory.
In case of error, the method throws an exception and sets the \texttt{STATUS} variable. Depending on the error, a human-readable description of the problem is available in the \texttt{SQLCA.SQLERRM} register. See \texttt{Error handling in GWS calls (STATUS)} on page 3554.

\texttt{security.Digest.AddBase64Data}

Adds data in base64 format to the digest buffer.

\textbf{Syntax}

\begin{verbatim}
security.Digest.AddBase64Data ( 
    toDigest STRING )
\end{verbatim}

1. \textit{toDigest} defines the base64 data string to be added to the digest buffer.

\textbf{Usage}

Use the method to decode the given base64 string and add the binary data to the digest buffer.

After adding all the data pieces, the buffer can be processed by calling \texttt{security.Digest.DoBase64Digest} or \texttt{security.Digest.DoHexBinaryDigest}.

In case of error, the method throws an exception and sets the \texttt{STATUS} variable. Depending on the error, a human-readable description of the problem is available in the \texttt{SQLCA.SQLERRM} register. See \texttt{Error handling in GWS calls (STATUS)} on page 3554.

\texttt{security.Digest.AddHexBinaryData}

Adds data in hexadecimal format to the digest buffer.

\textbf{Syntax}

\begin{verbatim}
security.Digest.AddHexBinaryData ( 
    toDigest STRING )
\end{verbatim}

1. \textit{toDigest} defines the hexadecimal data string to be added to the digest buffer.

\textbf{Usage}

Use the method to decode the given hexadecimal string and add the binary data to the digest buffer.

After adding all the data pieces, the buffer can be processed by calling \texttt{security.Digest.DoBase64Digest} or \texttt{security.Digest.DoHexBinaryDigest}.

In case of error, the method throws an exception and sets the \texttt{STATUS} variable. Depending on the error, a human-readable description of the problem is available in the \texttt{SQLCA.SQLERRM} register. See \texttt{Error handling in GWS calls (STATUS)} on page 3554.

\texttt{security.Digest.AddStringData}

Adds a data string to the digest buffer.

\textbf{Syntax}

\begin{verbatim}
security.Digest.AddStringData ( 
    toDigest STRING )
\end{verbatim}

1. \textit{toDigest} defines the human-readable character string to be added to the digest buffer.

\textbf{Usage}

This method adds the specified string data to the digest buffer.
After adding all the data pieces, the buffer can be processed by calling `security.Digest.DoBase64Digest` or `security.Digest.DoHexBinaryDigest`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**Related concepts**

- `security.Digest.AddStringDataWithCharset` on page 3145
  Adds a data string to the digest buffer, after converting to the specified character set.

**Syntax**

```java
security.Digest.AddStringDataWithCharset ( 
    toDigest STRING, 
    charset STRING ) 
```

1. `toDigest` defines the human-readable character string to be added to the digest buffer.
2. `charset` defines the charset to be used.

**Usage**

This method adds the specified string data based on the given charset to the digest buffer.

Before adding the string, it is converted from the local DVM charset to the specified encoding.

After adding all the data pieces, the buffer can be processed by calling `security.Digest.DoBase64Digest` or `security.Digest.DoHexBinaryDigest`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

**Related concepts**

- `security.Digest.AddStringData` on page 3144
  Adds a data string to the digest buffer.

**Syntax**

```java
security.Digest.CreateDigest ( 
    algo STRING ) 
RETURNS security.Digest 
```

1. `algo` defines the digest algorithm to be used.

**Usage**

Use this method to create and initialize a digest context to compute data digest based on the specified algorithm.

Available digest algorithms are:

- "SHA1"
- "SHA224"
- "SHA256"
In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**security.Digest.CreateDigestString**

Creates a SHA1 digest from the given string.

**Syntax**

```java
security.Digest.CreateDigestString(
    toDigest STRING,
    randomBase64 STRING
) RETURNS STRING
```

1. `toDigest` defines the password to be digested.
2. `randomBase64` defines a random string in Base64.

**Usage**

Use this method to compute the SHA1 digest from the string in `toDigest` and an optional random Base64 form string, and return it in a string encoded in Base64 form.

The random value must be a valid Base64 string. You typically generate this value with the `security.RandomGenerator.CreateRandomString()` method.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

**Example**

```
DEFINE password, digest STRING ...
LET digest =
    security.Digest.CreateDigestString(
        password,
        security.RandomGenerator.CreateRandomString(16) )
```

**security.Digest.DoBase64Digest**

Creates a digest of the buffered data and returns the result in base64 format.

**Syntax**

```java
security.Digest.DoBase64Digest ()
RETURNS STRING
```

**Usage**

This method processes the digest on all data previously added to the context and encodes it in base64.

After that call, the internal buffer is cleaned and ready to be populated again with new data to be digested.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.
security.Digest.DoHexBinaryDigest
Creates a digest of the buffered data and returns the result in hexadecimal format.

Syntax

```plaintext
security.Digest.DoHexBinaryDigest ()
RETURNS STRING
```

Usage

This method processes the digest on all data previously added to the context and returns it in hexadecimal.

After that call, the internal buffer is cleaned and ready to be populated again with new data to be digested.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 3554.

Example

Computing a hash value of a string.

Program example `ComputeHash.4gl`:

```plaintext
IMPORT SECURITY

MAIN

DEFINE result STRING

IF num_args() != 2 THEN
    DISPLAY "Usage: ComputeHash <string> <hashcode>"
    DISPLAY "  string: the string to digest"
    DISPLAY "  hashcode: SHA1, SHA512, SHA384, SHA256, SHA224, MD5"
ELSE
    LET result = ComputeHash(arg_val(1), arg_val(2))
    IF result IS NOT NULL THEN
        DISPLAY "Hash value is :",result
    ELSE
        DISPLAY "Error"
    END IF
END IF

END MAIN

FUNCTION ComputeHash(toDigest, algo)

DEFINE toDigest, algo, result STRING
DEFINE dgst security.Digest

TRY
    LET dgst = security.Digest.CreateDigest(algo)
    CALL dgst.AddStringData(toDigest)
    LET result = dgst.DoBase64Digest()
CATCH
    DISPLAY "ERROR : ", STATUS, " - ", SQLCA.SQLERRM
    EXIT PROGRAM(-1)
END TRY

RETURN result
END FUNCTION
```

Example execution:
$ fglrun ComputeHash "Hello World" SHA1
Hash value is: Ck1VqNd45QIvq3AZd8XYQLvEhtA=

The PBKDF2 class
The security.PBKDF2 class generates passwords using the Password-Based Key Derivation Function 2 (PBKDF2).

security.PBKDF2 methods
Methods of the security.PBKDF2 class.

Table 713: Class methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>security.PBKDF2.GenerateKey (</td>
<td>Generates a password of a given size based on a human readable password using Password-Based Key Derivation Function 2 (PBKDF2)</td>
</tr>
<tr>
<td>password STRING,</td>
<td></td>
</tr>
<tr>
<td>salt STRING,</td>
<td></td>
</tr>
<tr>
<td>hash STRING,</td>
<td></td>
</tr>
<tr>
<td>iter INTEGER,</td>
<td></td>
</tr>
<tr>
<td>keySize INTEGER )</td>
<td></td>
</tr>
<tr>
<td>RETURNS STRING</td>
<td></td>
</tr>
<tr>
<td>security.PBKDF2.CheckKey (</td>
<td>Validates a hashed key.</td>
</tr>
<tr>
<td>password STRING,</td>
<td></td>
</tr>
<tr>
<td>salt STRING,</td>
<td></td>
</tr>
<tr>
<td>hash STRING,</td>
<td></td>
</tr>
<tr>
<td>iter INTEGER,</td>
<td></td>
</tr>
<tr>
<td>hashedkey STRING )</td>
<td></td>
</tr>
<tr>
<td>RETURNS BOOLEAN</td>
<td></td>
</tr>
</tbody>
</table>

security.PBKDF2.GenerateKey
Generates a password of a given size based on a human readable password using Password-Based Key Derivation Function 2 (PBKDF2)

Syntax

security.PBKDF2.GenerateKey (  
password STRING,  
salt STRING,  
hash STRING,  
iter INTEGER,  
keySize INTEGER )  
RETURNS STRING

1. *password* defines the human readable password to derive using the PBKDF2 method.
2. *salt* defines the base64 random value created using Security.RandomGenerator.CreateRandomString(). Can be NULL.
3. *hash* defines the hash operation. By default, it is "SHA1". Valid values include:
   - SHA1
   - SHA224
   - SHA256
   - SHA384
   - SHA512
   - MD5
4. \textit{iter} defines the number of iterations to compute the derived password. This value must be greater than or equal to zero (\(\geq 0\)).

5. \textit{keySize} defines the size in bytes of the returned key. Must be greater than zero (\(>0\))

\textbf{Usage}

This method generates a password of a given size based on a human readable password using \texttt{PBKDF2}.

In practice, with the same salt value and the same human readable password, the same key can be regenerated in another application and therefore used as symmetric key to decrypt data encrypted in the initial application.

This method may raise exception -15700 (operation failed) or -15701 (invalid parameter).

In case of error, the method throws an exception and sets the \texttt{STATUS} variable. Depending on the error, a human-readable description of the problem is available in the \texttt{SQLCA.SQLERRM} register. See Error handling in GWS calls (\texttt{STATUS}) on page 3554.

For an example using \texttt{PBKDF2} methods, see Example: Using \texttt{security.PBKDF2} methods on page 3150.

\texttt{security.PBKDF2.CheckKey}

Validates a hashed key.

\textbf{Syntax}

\begin{verbatim}
security.PBKDF2.CheckKey (  
    password STRING,  
    salt STRING,  
    hash STRING,  
    iter INTEGER,  
    hashedkey STRING )  
RETURNS BOOLEAN
\end{verbatim}

1. \texttt{password} defines the human readable password to derive using the \texttt{PBKDF2} method.

2. \texttt{salt} defines the base64 random value created using \texttt{Security.RandomGenerator.CreateRandomString()}. Can be \texttt{NULL}.

3. \texttt{hash} defines the hash operation. By default, it is "SHA1". Valid values include:

   - SHA1
   - SHA224
   - SHA256
   - SHA384
   - SHA512
   - MD5

4. \texttt{iter} defines the number of iterations to compute the derived password. This value must be greater than or equal to zero (\(\geq 0\)).

5. \texttt{hashedkey} defines the key created by the \texttt{Security.PBKDF2.GenerateKey()} class method.

\textbf{Usage}

This method validates a hashed key produced by the \texttt{security.PBKDF2.GenerateKey} method.

The method returns \texttt{TRUE}, if the hashed key is valid, and returns \texttt{FALSE}, if the hashed key is not valid.

This method may raise exception -15700 (operation failed) or -15701 (invalid parameter).

In case of error, the method throws an exception and sets the \texttt{STATUS} variable. Depending on the error, a human-readable description of the problem is available in the \texttt{SQLCA.SQLERRM} register. See Error handling in GWS calls (\texttt{STATUS}) on page 3554.
Example: Using security.PBKDF2 methods
This example generates a key size of 128-bits based on a given password.

The 128-bits key can then be used in xml.CryptoKey, for instance.

```sql
IMPORT SECURITY

MAIN
DEFINE salt STRING
DEFINE result STRING
LET salt = Security.RandomGenerator.CreateRandomString(8)
TRY
    CALL Security.PBKDF2.GenerateKey(arg_val(1), salt, "sha1", 1000, 16)
RETURNING result
    DISPLAY "Generate Key of 128bits value is ":result # 128/8==16
CATCH
    IF status == -15700 THEN
        DISPLAY "Generation failed ":SQLCA.SQLERRM
    ELSE
        IF status == -15701 THEN
            DISPLAY "Invalid parameter ":SQLCA.SQLERRM
        ELSE
            DISPLAY "Unknown error ":STATUS
        END IF
    END IF
END TRY
END MAIN
```

The BCrypt class
The security.BCrypt class lets you save passwords as BCrypt results instead of clear text.

If you need to store passwords on a database for instance, you can save them as BCrypt results instead of clear text. This makes them difficult to hack, as the time to generate one is expensive.

security.BCrypt methods
Methods of the security.BCrypt class.

Table 714: Class methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>security.BCrypt.GenerateSalt</code> (&lt;i&gt;cost&lt;/i&gt; INTEGER) RETURNS STRING</td>
<td>Generates the encoded value needed as input to the HashPassword method.</td>
</tr>
<tr>
<td><code>security.BCrypt.HashPassword</code> (&lt;i&gt;password&lt;/i&gt; STRING, &lt;i&gt;salt&lt;/i&gt; STRING) RETURNS STRING</td>
<td>Creates a hash password.</td>
</tr>
<tr>
<td><code>security.BCrypt.CheckPassword</code> (&lt;i&gt;password&lt;/i&gt; STRING, &lt;i&gt;hashedPass&lt;/i&gt; STRING) RETURNS INTEGER</td>
<td>Checks the hash password.</td>
</tr>
</tbody>
</table>
**security.BCrypt.GenerateSalt**
Generates the encoded value needed as input to the HashPassword method.

**Syntax**

```sql
security.BCrypt.GenerateSalt(
    cost INTEGER
) RETURNS STRING
```

1. `cost` defines the number of rounds of hashing to apply. The default value is 10. This value must be between 4 and 30. It represents \(2^{\text{cost}}\) iteration. An iteration above 14 may take several minutes to compute.

   **Warning:** Using a high cost value for the salt is very CPU consuming, and can really slow down the application depending on the system it is running. Be forewarned that this is expected!

**Usage**

This method generates the encoded value needed as input to the HashPassword method.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

For an example using bcrypt methods, see Example: Using security.BCrypt methods on page 3152.

**Related concepts**

- **security.BCrypt.CheckPassword** on page 3152
  Checks the hash password.
- **security.BCrypt.HashPassword** on page 3151
  Creates a hash password.

**security.BCrypt.HashPassword**
Creates a hash password.

**Syntax**

```sql
security.BCrypt.HashPassword(
    password STRING,
    salt STRING
) RETURNS STRING
```

1. `password` defines the password to hash. The password is limited to 72 bytes.
2. `salt` defines an encoded value generated by GenerateSalt() that has a dedicated format. If the same hash value is computed again on another application, the same salt must be used. The format of the salt value follows this example: $2a$cost$modified_base64_encoded (random value of 16-byte length). If `salt` is NULL, the HashPassword method will generate one with a cost of 10.

**Usage**

The resulting hash password is composed of the version, the cost, and the salt+cipher separated by $, as in this example:

```
$2a$12$EXRkfkd9n2gzds2CSitu.MW9.qAVqa9eLS1/RYtYcmB1eLHg.9q
```

where:

- "2a" is the version of BCrypt. The current API supports 2a to 2z.
- "12" is the cost.
- The remainder is the salt + cipher result concatenated and encoded in "modified" base64:
• The first 22 characters ("EXRkfkdXn2gzs2SSitu." in our example) decode to a 16-byte value for the salt.
• The remaining characters ("MW9.gAVqa9eLS1//RYtYcmBleLHg.9q" in our example) are cipher text.

This method may raise exception -15700 (operation failed) or -15701 (invalid parameter).

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

For an example using BCrypt methods, see Example: Using security.BCrypt methods on page 3152.

Related concepts
security.BCrypt.CheckPassword on page 3152
Checks the hash password.

security.BCrypt.GenerateSalt on page 3151
Generates the encoded value needed as input to the HashPassword method.

security.BCrypt.CheckPassword
Checks the hash password.

Syntax

```
security.BCrypt.CheckPassword(
    password STRING,
    hashedPass STRING )
RETURNS  INTEGER
```

1. `password` defines the password to hash. The password is limited to 72 bytes.
2. `hashedPass` defines the hash password created by the HashPassword method.

Usage

The method returns TRUE, the hashed password is valid.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 3554.

For an example using BCrypt methods, see Example: Using security.BCrypt methods on page 3152.

Related concepts
security.BCrypt.HashPassword on page 3151
Creates a hash password.

security.BCrypt.GenerateSalt on page 3151
Generates the encoded value needed as input to the HashPassword method.

Example: Using security.BCrypt methods

This example creates (and checks) a hash password as BCrypt results.

```
IMPORT SECURITY

MAIN
    DEFINE salt STRING
    DEFINE hashed_pass STRING
    LET salt = Security.BCrypt.GenerateSalt(12)
    CALL Security.BCrypt.HashPassword(arg_val(1), salt) RETURNING hashed_pass
    DISPLAY "Hashed password is ":hashed_pass
```
IF Security.BCrypt.CheckPassword(arg_val(1), hashed_pass) THEN
  DISPLAY "OK: password check done."
ELSE
  DISPLAY "KO: password check failed."
END IF
END MAIN

File extensions

This page describes the file extensions used by the language.

Table 715: File extensions

<table>
<thead>
<tr>
<th>Extension</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.4gl</td>
<td>Text</td>
<td>Source module</td>
</tr>
<tr>
<td>.42m</td>
<td>Binary</td>
<td>Compiler p-code module</td>
</tr>
<tr>
<td>.per</td>
<td>Text</td>
<td>Form specification file</td>
</tr>
<tr>
<td>.42f</td>
<td>XML</td>
<td>Compiled form specification file</td>
</tr>
<tr>
<td>.42s</td>
<td>Binary</td>
<td>Compiled localized strings file</td>
</tr>
<tr>
<td>.4st</td>
<td>XML</td>
<td>Presentation styles resource file</td>
</tr>
<tr>
<td>.4sm</td>
<td>XML</td>
<td>Startmenu resource file</td>
</tr>
<tr>
<td>.4tm</td>
<td>XML</td>
<td>Topmenu resource file</td>
</tr>
<tr>
<td>.4tb</td>
<td>XML</td>
<td>Toolbar resource file</td>
</tr>
<tr>
<td>.4ad</td>
<td>XML</td>
<td>Action defaults resource file</td>
</tr>
<tr>
<td>.sch</td>
<td>Text</td>
<td>Database schema file - column types</td>
</tr>
<tr>
<td>.42d</td>
<td>Binary (development only)</td>
<td>Database schema file index (for .sch)</td>
</tr>
<tr>
<td>.str</td>
<td>Text</td>
<td>Localized strings source file</td>
</tr>
<tr>
<td>.val</td>
<td>Text</td>
<td>Database schema file - form field attributes</td>
</tr>
<tr>
<td>.att</td>
<td>Text</td>
<td>Database schema file - video attributes</td>
</tr>
<tr>
<td>.42r</td>
<td>Binary</td>
<td>Program file</td>
</tr>
<tr>
<td>.42x</td>
<td>Binary</td>
<td>P-code library</td>
</tr>
<tr>
<td>.msg</td>
<td>Text</td>
<td>Message definition source file</td>
</tr>
<tr>
<td>.iem</td>
<td>Binary</td>
<td>Compiled message definition file</td>
</tr>
</tbody>
</table>
## Genero BDL errors

System error messages sorted by error number.

### Table 716: Genero system error messages

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -201   | A syntax error has occurred.  
This general SQL error message indicates mistakes in the syntax of an SQL statement.  
Look for missing or extra punctuation; keywords misspelled, misused, or out of sequence, or a reserved word used as an identifier. |
| -204   | An illegal floating point number has been found in the statement.  
A numeric constant that is punctuated like a floating-point number (with a decimal point and/or an exponent starting with e) is unacceptable. Possibly the exponent is larger than can be processed. |
| -206   | The specified table table-name is not in the database.  
The database server cannot find a table or view specified in the statement. The table or view might have been renamed or dropped from the database. |
| -213   | Statement interrupted by user.  
The database server received an interrupt signal from the user. The statement ended early. The program is expected to roll back the current transaction and terminate gracefully. |
| -217   | Column column-name not found in any table in the query.  
The column specified does not exist in the database tables used in this SQL statement. |
| -235   | Character column size is too big.  
The SQL statement specifies a width for a character data type that is greater than 65,534 bytes.  
If you need a column of this size, use the TEXT data type, which allows unlimited lengths. Otherwise, inspect the statement for typographical errors. |
| -236   | Number of columns in INSERT does not match number of VALUES.  
Each column that is named or implied in an INSERT statement must have a separate value expression. If the statement does not list specific columns, review the definition of the table for the number of columns and their data types. Also check that the list of expressions in the VALUES clause has no extra or missing comma that might result in an incorrect number of values. Be especially careful of long character strings and expressions with parentheses. |
| -239   | Could not insert new row - duplicate value in a UNIQUE INDEX column.  
The row that is being inserted (or being updated to have a new primary key) contains a duplicate value of some row that already exists, in a column or columns that are constrained to have unique values. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -244   | Could not do a physical-order read to fetch next row.  
The database server cannot read the data block for this SQL client program. The database server returns this error when a record is locked by another process, and the lock timeout defined by the current program has expired.  
Consider using the SET LOCK MODE TO WAIT instruction to define a lock timeout. By default, with most databases, this timeout is zero and error -244 is returned immediately when a lock conflict occurs. If all programs do short transactions (holding locks for a short period of time), it is usually safe to define a lock timeout of 5 to 10 seconds to avoid this SQL error. |
| -250   | Cannot read record from file for update.  
The database server cannot get a row of a table prior to update. |
| -251   | ORDER BY or GROUP BY column number is too big.  
The ORDER BY or GROUP BY clause uses column-sequence numbers, and at least one of them is larger than the count of columns in the select list. |
| -253   | Cannot read record from file for update.  
The database server cannot get a row of a table prior to update. |
| -254   | Too many or too few host variables given.  
The number of host variables that you named in the INTO clause of this statement does not match the number of columns that you referenced in the SQL statement. |
| -255   | Not in transaction.  
The database server cannot execute this COMMIT WORK or ROLLBACK WORK statement because no BEGIN WORK was executed to start a transaction. Because no transaction was started, you cannot end one. |
| -256   | Transaction not available.  
The database server does not support transactions. |
| -257   | System limit on maximum number of statements exceeded, maximum is count.  
The database server can handle only a fixed number of prepared SQL statements for each user. This limit includes statements that were prepared with the PREPARE statement and cursors that were declared with the DECLARE statement. |
| -259   | Cursor not open.  
The current statement refers to a cursor that has not been opened. Review the logic of the program to see how it failed to execute the OPEN statement before it reached this point. |
| -263   | Could not lock row for UPDATE.  
This statement, probably a FETCH statement that names a cursor declared FOR UPDATE, failed to get a lock on the row it tried to fetch. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -266   | There is no current row for UPDATE/DELETE cursor.  
The current statement uses the WHERE CURRENT OF cursor-name clause, but that cursor has not yet been associated with a current row. Either no FETCH statement has been executed since it was opened, or the most recent fetch resulted in an error so that no row was returned. Revise the logic of the program so that it always successfully fetches a row before it executes this statement. |
| -268   | Unique constraint constraint-name violated.  
The current statement uses the WHERE CURRENT OF cursor-name clause, but that cursor has not yet been associated with a current row. Either no FETCH statement has been executed since it was opened, or the most recent fetch resulted in an error so that no row was returned. Revise the logic of the program so that it always successfully fetches a row before it executes this statement. |
| -272   | No SELECT permission for table/column.  
The person who created this table has not granted SELECT privilege to your account name or to the public for the table or the column. The owner of the table or the DBA must grant this privilege before you can select data from the table or column. |
| -273   | No UPDATE permission for table/column.  
The person who created this table has not granted UPDATE privilege to your account name or to the public for the table or the column. The owner of the table or the DBA must grant this privilege before you can update a row in this table or update the column. |
| -274   | No DELETE permission for table.  
The person who created this table has not granted DELETE privilege to your account name or to the public. The owner of the table or the DBA must grant this privilege before you can delete a row in this table. |
| -275   | The Insert privilege is required for this operation.  
The Insert access privilege on this table or column is not currently held by your account name, nor by the PUBLIC group, nor by your current role. The owner of the table or the DBA must grant the Insert privilege before you can insert a row into this table. |
| -280   | A quoted string exceeds 256 bytes.  
A character literal in this statement exceeds the maximum length. Check the punctuation and length of all quoted strings in the statement. Possibly two missing quotes make a long string out of two short ones. You must revise the statement to use a shorter character string. |
| -282   | Found a quote for which there is no matching quote.  
Inspect the current statement, examining the punctuation of all quoted strings. |
A subquery has returned not exactly one row.

A subquery that is used in an expression in the place of a literal value must return only a single row and a single column. In this statement, a subquery has returned more than one row, and the database server cannot choose which returned value to use in the expression. You can ensure that a subquery will always return a single row. Use a WHERE clause that tests for equality on a column that has a unique index. Or select only an aggregate function. Review the subqueries, and check that they can return only a single row.

This error can also occur when you use a singleton SELECT statement to retrieve multiple rows. You must use the DECLARE/OPEN/FETCH series of statements or the EXECUTE INTO statement to retrieve multiple rows.

Invalid cursor received by sqlexec.

The cursor that this statement uses has not been properly declared or prepared, or the FREE statement has released it, or an automatic re-prepare has been attempted while opening the cursor but that operation failed, leaving the cursor unavailable. Review the program logic to ensure that the cursor has been declared. If it has, and if the DECLARE statement refers to a statement identifier, check that the referenced statement has been prepared.

Cursor not declared with FOR UPDATE clause.

This statement attempts to update with a cursor that was not declared for update. To use a cursor with the UPDATE or DELETE statements, you must declare it FOR UPDATE. Review the program logic to make sure that this statement uses the intended cursor.

The column column-name must be in the GROUP BY list.

In a grouping SELECT, you must list every nonaggregate column in the GROUP BY clause to ensure that a well-defined value exists for each selected column in each grouped row. A column contains either a single aggregate value or a value unique to that group. If a selected column were neither an aggregate nor in the list, two or more values for that column might possibly exist in some group, and the database server is unable to choose which value to display. Revise the query to include either the column name or its positional number in the clause.

Illegal subscript.

The substring values (two numbers in square brackets) of a character variable are incorrect. The first is less than zero or greater than the length of the column, or the second is less than the first. Review all uses of square brackets in the statement to find the error. Possibly the size of a column has been altered and makes a substring fail that used to work.

ORDER BY column or expression must be in SELECT list.

An expression or column name is in the ORDER BY clause of this SELECT statement, but the expression or column name is not in the select list (the list of values that follows the word SELECT). This action is not supported when a UNIQUE or DISTINCT operator is being used in a query.

Index index-name already exists in database.

This statement tries to create an index with the name shown, but an index of that name already exists. Only one index of a given name can exist in a single database.
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-324</td>
<td>Ambiguous column column-name. The column name appears in more than one of the tables that are listed in the FROM clause of this query. The database server needs to know which columns to use. Revise the statement so that this name is prefixed by the name of its table (table-name.column) wherever it appears in the query.</td>
</tr>
<tr>
<td>-329</td>
<td>Database not found or no system permission. The database you tried to connect to is not known by the db server. Check database client configuration settings and make sure that there is no spelling error in the name of the database.</td>
</tr>
<tr>
<td>-330</td>
<td>Cannot create or rename the database. Possibly you tried to create a database with the same name as one that already exists or rename a database to a name that already exists; if so, choose a different name.</td>
</tr>
<tr>
<td>-349</td>
<td>Database not selected yet. The SQL statement cannot be executed because no current database exists. You must issue a DATABASE or CONNECT TO instruction before executing other SQL statements.</td>
</tr>
<tr>
<td>-350</td>
<td>Index already exists on the column (or on the set of columns). This CREATE INDEX statement cannot be executed because an index on the same column or combination of columns already exists. For a given collation order, at most two indexes can exist on any combination of columns, one ascending and one descending.</td>
</tr>
<tr>
<td>-354</td>
<td>Incorrect database or cursor name format. This statement contains the name of a database or a cursor in some invalid format. If the statement is part of a program, the name might have been passed in a host variable.</td>
</tr>
<tr>
<td>-360</td>
<td>Cannot modify a table or view that is also used in subquery. The UPDATE, INSERT, or DELETE statement uses data taken from the same table in a subquery. Because of the danger of entering an endless loop, this action is not allowed, except in the case of an uncorrelated subquery in the WHERE clause of the DELETE or UPDATE statement.</td>
</tr>
<tr>
<td>-363</td>
<td>CURSOR not on SELECT statement. The cursor named in this statement (probably an OPEN) has been associated with a prepared statement that is not a SELECT statement. Review the program logic, especially the DECLARE for the cursor, the statement id specified in it, and the PREPARE that set up that statement. If you intended to use a cursor with an INSERT statement, you can only do that when the INSERT statement is written as part of the DECLARE statement. If you intended to execute an SQL statement, do that directly with the EXECUTE statement, not indirectly through a cursor.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| -366   | The scale exceeds the maximum precision specified.  
A problem exists with the precision or scale of a DECIMAL or a MONEY data type usage, for example in a DEFINE statement. It is recommended to declare DECIMAL as DECIMAL(p) or DECIMAL(p,s), where p is the precision (total number of digits) between 1 and 32, and s, is the scale (number of digits to the right of the decimal point) greater or equal to zero and not greater than p. The MONEY type follows the same rules.  
Review the DECIMAL or MONEY type definition, and make sure that the precision is in the range [1,32] and that the scale is in the range [0,precision]. |
| -371   | Cannot create unique index on column with duplicate data.  
This CREATE UNIQUE INDEX statement cannot be completed because the column (or columns) contains one or more duplicate rows. |
| -382   | Same number of columns must be specified for view and select clause.  
In this VIEW statement, you have listed the names of the columns of the view; however, their number is different from the number of columns in the SELECT statement for the view. Check the punctuation of the two lists, and make sure that you have supplied a name for each item in the select list. |
| -387   | No connect permission.  
You cannot access the database that this statement requests because you have not been granted CONNECT privilege to it. Contact a person who has Database Administrator privilege to that database and ask to be granted CONNECT privileges to it. |
| -388   | No resource permission.  
If you issued a CREATE TABLE, CREATE INDEX, or CREATE PROCEDURE statement, you cannot execute this statement because your account has not been granted the RESOURCE privilege for this database. You need the RESOURCE privilege to create permanent tables, indexes on permanent tables, and procedures. |
| -389   | No DBA permission.  
This statement cannot be executed because you have not been granted DBA privilege for this database. Contact a person who has DBA privilege for the database and ask to be granted DBA privilege (or simply ask to have this statement executed for you). |
| -391   | Cannot insert a null into column column-name.  
This statement tries to put a null value in the noted column; however, that column has been defined as NOT NULL. Roll back the current transaction. If this is a program, review the definition of the table, and change the program logic to not use null values for columns that cannot accept them. |
| -400   | Fetch attempted on unopen cursor.  
This FETCH statement names a cursor that has never been opened or has been closed. Review the program logic, and check that it will open the cursor before this point and not accidentally close it. Unless a cursor is declared WITH HOLD, it is automatically closed by a COMMIT WORK or ROLLBACK WORK statement. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -404   | The cursor or statement is not available.  
You used a statement that names a cursor that is was destroyed. Review the program logic and check that the cursor specified is declared and opened, but not freed, prior to reaching this statement. |
| -410   | Prepare statement failed or was not executed.  
This EXECUTE statement refers to a statement id that has not been prepared. Either no PREPARE statement was done, or one was done but returned an error code. Review the program logic to ensure that a statement is prepared and the PREPARE return code is checked. A negative error code from PREPARE usually reflects an error in the statement being prepared. |
| -412   | Command pointer is NULL.  
This statement (probably an EXECUTE or DECLARE) refers to a dynamic SQL statement that has never been prepared or that has been freed. Review the program logic to ensure that the statement has been prepared, the PREPARE did not return an error code, and the FREE statement has not been used to release the statement before this point. |
| -413   | Insert attempted on unopen cursor.  
This INSERT statement names a cursor that has never been opened or that has been closed. Review the program logic, and check that it will open the cursor before this point and not accidentally close it. An insert cursor is automatically closed by a COMMIT WORK or ROLLBACK WORK statement. |
| -422   | Flush attempted on unopen cursor.  
This FLUSH statement names a cursor that has never been opened or has been closed. Review the program logic to ensure that it will open the cursor before this point and not accidentally close it. An insert cursor is automatically closed by a COMMIT WORK or ROLLBACK WORK. |
| -450   | Illegal ESQL locator, or uninitialized blob variable in BDL.  
An SQL statement is using a TEXT or BYTE variable that was not initialized with LOCATE IN FILE or MEMORY. LOCATE the TEXT or BYTE variable before using it in SQL statements. |
| -458   | Long transaction aborted.  
The database server ran out of log space in which to record this transaction. A transaction that is not fully recorded cannot be rolled back. To preserve database integrity, the operating system ended the transaction and rolled it back automatically. All changes made since the start of the transaction have been removed. Terminate the application, and replan it so that it modifies fewer rows per transaction. Alternatively, contact the database server administrator to discuss increasing the number or the size of the logical logs. Be prepared to talk about the number of rows being updated or inserted and the size of each row. |
| -481   | Invalid statement name or statement was not prepared.  
The statement has not been prepared, or the format of the statement name is not valid. A valid statement name does not exceed the maximum length, begins with a letter or underscore, does not contain any blanks or nonalphanumeric characters except underscores. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-482</td>
<td>Invalid operation on a non-SCROLL cursor. You cannot issue a FETCH PRIOR, FETCH FIRST, FETCH LAST, FETCH CURRENT, FETCH RELATIVE n, or FETCH ABSOLUTE n statement with a non-scroll cursor. To do so, you must first declare the cursor as a scroll cursor.</td>
</tr>
<tr>
<td>-507</td>
<td>Cursor cursor-name not found. The cursor that is named in the WHERE CURRENT OF clause in this UPDATE or DELETE statement does not exist. Review the spelling of the name. If it is as you intended, check the DECLARE statement to ensure that it has been executed. Also make sure that the cursor has not been freed with the FREE statement or during a failed automatic re-prepare attempt.</td>
</tr>
<tr>
<td>-513</td>
<td>Statement not available with this database server. The SQL statement used by the program is not valid for the target database server. Review the code, the SQL instruction cannot be used.</td>
</tr>
<tr>
<td>-517</td>
<td>The total size of the index is too large or too many parts in index. All database servers have limits on the number of columns that can be included in an index and on the total number of bytes in a key (the sum of the widths of the columns). This CREATE INDEX statement would exceed that limit for this database server.</td>
</tr>
<tr>
<td>-522</td>
<td>Table table-name not selected in query. The table name used in an expression (for example, in the WHERE clause) has not been listed in the clause defining the tables to be used in the query (typically the FROM clause in SELECT statements).</td>
</tr>
<tr>
<td>-526</td>
<td>Updates are not allowed on a scroll cursor. For a DECLARE statement, the clause FOR UPDATE is not allowed in conjunction with the SCROLL keyword.</td>
</tr>
<tr>
<td>-530</td>
<td>Check constraint constraint-name failed. The check constraint placed on the table column was violated.</td>
</tr>
<tr>
<td>-535</td>
<td>Already in transaction. This BEGIN WORK statement is redundant; a transaction is already in progress. If this is a program, review its logic to make sure it has not accidentally failed to end the previous transaction.</td>
</tr>
<tr>
<td>-551</td>
<td>The constraint contains too many columns. The total number of columns listed in a UNIQUE, PRIMARY KEY, or FOREIGN KEY clause is limited. The limit depends on the database server in use.</td>
</tr>
<tr>
<td>-611</td>
<td>Scroll cursor can't select TEXT or BYTE columns. Because of the bulk of TEXT and BYTE values, this action would require too much resources to be performed. Review the declaration of the cursor to select the desired columns of other types and also the primary key column. Use a secondary SELECT statement to fetch TEXT/BYTE columns for a given row, by using the primary key column value.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| -674   | Routine routine-name cannot be resolved.  
You called a routine that does not exist in the database, you do not have permission to execute the routine, or you called the routine with too few or too many arguments. |
| -681   | Column specified more than once in the INSERT list.  
The error occurs if the user specifies a column name more than once in the INSERT column list. |
| -691   | Missing key in referenced table for referential constraint constraint-name.  
A referential constraint has been violated. This condition usually occurs when you are trying to insert a value into or update the value of a column that is part of a referential constraint. The value you are trying to enter does not exist in the referenced (parent-key) column. |
| -692   | Key value for constraint constraint-name is still being referenced.  
You have violated a referential constraint. This situation usually occurs when you are trying to delete a row in a column (parent key) that another row (child key) is referencing. If you are using cascading deletes, database logging must be on. |
| -743   | Object object_name already exists in database.  
You are trying to define an object that already exists in the database. |
| -768   | Internal error in routine routine-name.  
If this internal error recurs, note all circumstances and contact your technical support. |
| -805   | Cannot open file for load.  
The input file that is specified in this LOAD statement is unable to be opened.  
Check the statement. Possibly a more complete path name is needed, the file does not exist, or your account does not have read permission for the file or a directory in which it resides. |
| -806   | Cannot open file for unload.  
The output file that is specified in this UNLOAD statement is unable to be opened.  
Check the statement. Possibly a more complete path name is needed; the file exists, but your account does not have write permission for it; or the disk is full. |
| -809   | SQL Syntax error has occurred.  
The INSERT statement in this LOAD/UNLOAD statement has invalid syntax.  
Review it for punctuation and use of keywords. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-846</td>
<td>Number of values in load file is not equal to number of columns. The LOAD processor counts the delimiters in the first line of the file to determine the number of values in the load file. One delimiter must exist for each column in the table, or for each column in the list of columns if one is specified. Check that you specified the file that you intended and that it uses the correct delimiter character. An empty line in the text can also cause this error. If the LOAD statement does not specify a delimiter, verify that the default delimiter matches the delimiter that is used in the file. If you are in doubt about the default delimiter, specify the delimiter in the LOAD statement.</td>
</tr>
<tr>
<td>-930</td>
<td>Cannot connect to database server servername. The application is trying to access the database server but failed.</td>
</tr>
<tr>
<td>-942</td>
<td>Transaction commit failed - transaction will be rolled back. This error can occur at transaction-commit time if the database server is unable to commit the transaction.</td>
</tr>
<tr>
<td>-1102</td>
<td>Field name not found in form. A field name listed in an INPUT, INPUT ARRAY, CONSTRUCT, SCROLL or DISPLAY statement does not appear in the form specification of the screen form that is currently displayed. Review the program logic to ensure that the intended window is current, the intended form is displayed in it, and all the field names in the statement are spelled correctly.</td>
</tr>
<tr>
<td>-1107</td>
<td>Field subscript out of bounds. The subscript of a screen array in an INPUT, DISPLAY, or CONSTRUCT statement is either less than 1 or greater than the number of fields in the array. Review the program source in conjunction with the form specification to see where the error lies.</td>
</tr>
<tr>
<td>-1108</td>
<td>Record name not in form. The screen record that is named in an INPUT ARRAY or DISPLAY ARRAY statement does not appear in the screen form that is now displayed. Review the program source in conjunction with the form specification to see if the screen record names match.</td>
</tr>
<tr>
<td>-1109</td>
<td>List and record field counts differ. The number of program variables does not agree with the number of screen fields in a CONSTRUCT, INPUT, INPUT ARRAY, DISPLAY, or DISPLAY ARRAY statement. Review the statement in conjunction with the form specification to see where the error lies. Common problems include a change in the definition of a screen record that is not reflected in every statement that uses the record, and a change in a program record that is not reflected in the form design.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| -1110  | Form file (file-name) not found.  
The form file that is specified in an OPEN FORM or OPEN WINDOW WITH FORM statement was not found.  
Inspect the form name used in the statement. The form file suffix is not required. If the form is not in the current directory, verify that FGLRESOURCEPATH / DBPATH environment variables contain the path to the form file. |
| -1112  | A form is incompatible with the current BDL version. Rebuild your form.  
The form file that is specified in an OPEN FORM statement is not acceptable. Possibly it was corrupted in some way, or it was compiled with a version of the Form Compiler that is not compatible with the version of the BDL compiler that compiled this program.  
Use a current version of the Form Compiler to recompile the form specification. |
| -1114  | No form has been displayed.  
The current statement requires the use of a screen form. For example, DISPLAY...TO or an INPUT statement must use the fields of a form; however, the DISPLAY FORM statement has not been executed since the current window was opened.  
Review the program logic to ensure that it opens and displays a form before it tries to use a form. |
| -1119  | NEXT FIELD name not found in form.  
This statement (INPUT or INPUT ARRAY) contains a NEXT FIELD clause that names a field that is not defined in the form.  
Review the form and program logic. Perhaps the form has been changed, but the program has not. |
| -1129  | Field (field-name) in BEFORE/AFTER clause not found in form.  
This statement includes a BEFORE FIELD clause or an AFTER FIELD clause that names a field that is not defined in the form that is currently displayed.  
Review the program to ensure that the intended form was displayed, and review this statement against the form specification to ensure that existing fields are named. |
| -1133  | The NEXT OPTION name is not in the menu.  
This MENU statement contains a NEXT OPTION clause that names a menu-option that is not defined in the statement.  
The string that follows NEXT OPTION must be identical to one that follows a COMMAND clause in the same MENU statement. Review the statement to ensure that these clauses agree with each other. |
| -1140  | NEXT OPTION is a hidden option.  
The option that is named in this NEXT OPTION statement has previously been hidden with the HIDE OPTION statement. Because it is not visible to the user, it cannot be highlighted as the next choice.  
Use the SHOW OPTION statement to unhide the menu option. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -1141  | Cannot close window with active INPUT, DISPLAY ARRAY, or MENU statement.  
This CLOSE WINDOW statement cannot be executed because an input operation is still active in that window. The CLOSE WINDOW statement must have been contained in, or called from within, the input statement itself.  
Review the program logic, and revise it so that the statement completes before the window is closed. |
| -1143  | Window is already open.  
This OPEN WINDOW statement names a window that is already open.  
Review the program logic to see if it contains a CLOSE WINDOW statement. It may be possible that you simply need to use a CURRENT WINDOW statement to bring the open window to the top. |
| -1146  | PROMPT message is too long to fit in the window.  
Although BDL truncates the output of MESSAGE and COMMENT to fit the window dimensions, it does not do so for PROMPT and the user's response.  
Reduce the length of the prompt string, or make the window larger. Another option is to display most of the prompting text with DISPLAY and then prompt with a single space or colon. |
| -1150  | Window is too small to display this menu.  
The window must be at least two rows tall, and it must be wide enough to display the menu title, the longest option name, two sets of three-dot ellipses, and six spaces. Revise the program to make the window larger or to give the menu a shorter name and shorter options.  
Review the OPEN WINDOW statement for the current window in conjunction with this MENU statement. |
| -1168  | Command does not appear in the menu.  
The SHOW OPTION, HIDE OPTION, or NEXT OPTION statement cannot refer to an option (command) that does not exist.  
Check the spelling of the name of the option. |
| -1170  | The type of your terminal is unknown to the system.  
Check the setting of your TERM environment variable and the setting of your TERMCP or TERMINFO environment variable.  
Check with your system administrator if you need help with this action. |
| -1202  | An attempt was made to divide by zero.  
Zero cannot be a divisor.  
Check that the divisor is not zero. In some cases, this error arises because the divisor is a character value that does not convert properly to numeric. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -1204  | Invalid year in date.  
The year in a DATE value or literal is invalid. For example, the number 0000 is not acceptable as the year.  
Check the value of year. |
| -1205  | Invalid month in date.  
The month in a DATE value or literal must be a one- or two-digit number from 1 to 12.  
Check the value of month. |
| -1206  | Invalid day in date.  
The day number in a DATE value or literal must be a one- or two-digit number from 1 to 28 (or 29 in a leap year), 30, or 31, depending on the month that accompanies it.  
Check the value of day. |
| -1210  | Date could not be converted to month/day/year format.  
The DATE type is compatible with the INTEGER type, but not all integer values are valid dates.  
The range of valid integer values for dates is from -693,594 to +2,958,464. Numbers that are outside this range have no representation as dates.  
Check the value of the number used to assign the date variable. |
| -1212  | Date conversion format must contain a month, day, and year component.  
**Note:** This error is non-trappable.  
When a date value is converted between internal binary format and display or entry format, a pattern directs the conversion. When conversion is done automatically, the pattern comes from the environment variable DBDATE. When it is done with an explicit call to the rfmtdate(), rdefmtdate(), or USING functions, a pattern string is passed as a parameter. In any case, the pattern string (the format of the message) must include letters that show the location of the three parts of the date: 2 or 3 letters d; 2 or 3 letters m; and either 2 or 4 letters y.  
Check the pattern string and the value of DBDATE. |
| -1213  | A character to numeric conversion process failed.  
A character value is being converted to numeric form for storage in a numeric column or variable; however, the character string cannot be interpreted as a number.  
Check the character string. It must not contain characters other than whitespace, digits, a sign, a decimal, or the letter e. Verify the parts are in the right order. If you are using NLS, the decimal character or thousands separator might be wrong for your locale. |
| -1214  | Value too large to fit in a SMALLINT.  
The SMALLINT data type can accept numbers with a value range from -32,767 to +32,767.  
To store numbers that are outside this range, redefine the column or variable to use INTEGER or DECIMAL type. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -1215  | Value too large to fit in an INTEGER.  
The INTEGER data type can accept numbers with a value range from \(-2,147,483,647\) to \(+2,147,483,647\).  
Check the other data types available, such as DECIMAL. |
| -1218  | String to date conversion error.  
The data value does not properly represent a date: either it has non-digits where digits are expected, an unexpected delimiter, or numbers that are too large or are inconsistent.  
Check the value being converted. |
| -1222  | Value will not fit in a SMALLFLOAT.  
A statement tries to assign a value that exceeds the limits of the SMALLFLOAT data type.  
Review the code and consider using a FLOAT or DECIMAL type. |
| -1223  | Value will not fit in a FLOAT.  
A statement tries to assign a value that exceeds the limits of the FLOAT data type.  
Review the code and consider using a DECIMAL type. |
| -1226  | Decimal or money value exceeds maximum precision.  
The data value has more digits to the left of the decimal point than the declaration of the variable allows.  
Revise the program to define the variable with an appropriate precision. |
| -1260  | It is not possible to convert between the specified types.  
**Note:** This error is non-trappable.  
Data conversion does not make sense, or is not supported.  
Possibly you referenced the wrong variable or column. Check that you have specified the data types that you intended and that literal representations of data values are correctly formatted. |
| -1261  | Too many digits in the first field of datetime or interval.  
The first field of a DATETIME literal must contain 1 or 2 digits (if it is not a YEAR) or else 2 or 4 digits (if it is a YEAR). The first field of an INTERVAL literal represents a count of units and can have up to 9 digits, depending on the precision that is specified in its qualifier.  
Review the DATETIME and INTERVAL literals in this statement, and correct them. |
| -1262  | Non-numeric character in datetime or interval.  
A DATETIME or INTERVAL literal can contain only decimal digits and the allowed delimiters: the hyphen between year, month, and day numbers; the space between day and hour; the colon between hour, minute, and second; and the decimal point between second and fraction. Any other characters, or these characters in the wrong order, produce an error.  
Check the value of the literal. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1263</td>
<td>A field in a datetime or interval is out of range. At least one of the fields in a datetime or interval is incorrect. Inspect the DATE, DATETIME, and INTERVAL literals in this statement. In a DATE or DATETIME literal, the year might be zero, the month might be other than 1 to 12, or the day might be other than 1 to 31 or inappropriate for the month. Also in a DATETIME literal, the hour might be other than 0 to 23, the minute or second might be other than 0 to 59, or the fraction might have too many digits for the specified precision.</td>
</tr>
<tr>
<td>-1264</td>
<td>Extra characters at the end of a datetime or interval. Only spaces can follow a DATETIME or INTERVAL literal. Inspect this statement for missing or incorrect punctuation.</td>
</tr>
<tr>
<td>-1265</td>
<td>Overflow occurred on a datetime or interval operation. An arithmetic operation involving a DATETIME and/or INTERVAL produced a result that cannot fit in the target variable. Check if the data type can hold the result of the operation. For example, extend the INTERVAL precision by using YEAR(9) or DAY(9).</td>
</tr>
<tr>
<td>-1266</td>
<td>Intervals or datetimes are incompatible for the operation. An arithmetic operation mixes DATETIME and/or INTERVAL values that do not match. Check the data types of the variable used in the operation.</td>
</tr>
<tr>
<td>-1267</td>
<td>The result of a datetime computation is out of range. In this statement, a DATETIME computation produced a value that cannot be stored. This situation can occur, for example, if a large interval is added to a DATETIME value. This error can also occur if the resultant date does not exist, such as Feb 29, 1999. Review the expressions in the statement and see if you can change the sequence of operations to avoid the overflow.</td>
</tr>
<tr>
<td>-1268</td>
<td>Invalid datetime or interval qualifier. This statement contains a DATETIME or INTERVAL qualifier that is not acceptable. These qualifiers can contain only the words YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, FRACTION, and TO. A number from 1 to 5 in parentheses can follow FRACTION. Inspect the statement for missing punctuation and misspelled words. A common error is adding an s, as in MINUTES.</td>
</tr>
<tr>
<td>-1279</td>
<td>Value exceeds string column length. You attempted to insert into a CHAR, NCHAR, VARCHAR, NVARCHAR or LVARCHAR column using a string host variable, but the string is too long.</td>
</tr>
<tr>
<td>-1284</td>
<td>Value will not fit in a BIGINT or INT8. The BIGINT data type can accept numbers with a value range from -9223372036854775807 to +9223372036854775807. To store numbers that are outside this range, redefine the column or variable to use the DECIMAL type.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| -1301  | This value is not among the valid possibilities.  
A list or range of acceptable values has been established for this column in the form-specification file.  
You must enter a value within the acceptable range. |
| -1302  | The two entries were not the same -- please try again.  
To guard against typographical errors, this field has been designated VERIFY in the form-specification file. You must enter the value in this field twice, identically.  
Carefully reenter the data. Alternatively, you can cancel the form entry with the Interrupt key. |
| -1303  | You cannot use this editing feature because a picture exists.  
This field is defined in the form-specification file with a PICTURE attribute to specify its format.  
You cannot use certain editing keys (for example, CTRL-A, CTRL-D, and CTRL-X) while you are editing such a field. Use only printable characters and backspace to enter the value. |
| -1304  | Error in field.  
You entered a value in this field that cannot be stored in the program variable that is meant to receive it.  
Possibly you entered a decimal number when the application provided only an integer variable, or you entered a character string that is longer than the application expected. |
| -1305  | This field requires an entered value.  
The cursor is in a form field that has been designated REQUIRED.  
You must enter some value before the cursor can move to another field. To enter a null value, type any printable character and then backspace. Alternatively, you can cancel the form entry with the Interrupt key. |
| -1306  | Please type again for verification.  
The cursor is in a form field that has been designated VERIFY. This procedure helps to ensure that no typographical errors occur during data entry.  
You must enter the value twice, identically, before the cursor can move to another field. Alternatively, you can cancel the form entry with the Interrupt key. |
| -1307  | Cannot insert another row -- the input array is full.  
You are entering data into an array of records that is represented in the program by a static array of program variables. That array is now full; no place is available to store another record.  
Press the ACCEPT key to process the records that you have entered. |
| -1308  | Cannot delete row -- it has no data.  
You try to delete a row in an empty row. Nothing was deleted. |
| -1309  | There are no more rows in the direction you are going.  
You are attempting to scroll an array of records farther than it can go, either scrolling up at the top or scrolling down at the bottom of the array. Further attempts will have the same result. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -1312  | FORMS statement error number error-num.  
An error occurred in the form at runtime.  
Edit your source file: go to the specified line, correct the error, and recompile the file. |
| -1313  | SQL statement error number error-num.  
The current SQL statement returned this error code number. |
| -1314  | Program stopped at 'filename', line number line-number.  
At runtime an error occurred in the specified file at the specified line. No .err file is generated.  
Edit your source file, go to the specified line, correct the error, and recompile the file. |
| -1318  | A parameter count mismatch has occurred between the calling function and the called function.  
Either too many or too few parameters were given in the call to the function.  
The call is probably in a different source module from the called functions. Inspect the definition of the function, and check all places where it is called to ensure that they use the number of parameters that it declares. |
| -1320  | A function has not returned the correct number of values expected.  
Note: This error is non-trappable.  
A function that returns several variables has not returned the correct number of parameters.  
Check your source code and recompile. |
| -1321  | A validation error has occurred as a result of the VALIDATE command.  
The VALIDATE LIKE statement tests the current value of variables against rules that are stored in the syscolval table. It has detected a mismatch.  
Ordinarily, the program would use the WHENEVER statement to trap this error and display or correct the erroneous values. Inspect the VALIDATE statement to see which variables were being tested and find out why they were wrong. |
| -1322  | A report output file cannot be opened: description  
The file that the REPORT TO statement specifies cannot be opened. See the description for more details.  
Check that your account has permission to write such a file, that the disk is not full, and that you have not exceeded some limit on the number of open files. |
| -1323  | A report output pipe cannot be opened.  
The pipe that the REPORT TO PIPE statement specifies is unable to be started.  
Check that all programs that are named in it exist and are accessible from your execution path.  
Also look for operating-system messages that might give more specific errors. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -1324  | A report output file cannot be written to.  
The file that the REPORT TO statement specifies was opened, but an error occurred while writing to it.  
Possibly the disk is full. Look for operating-system messages that might give more information. |
| -1326  | An array variable has been referenced outside of its specified dimensions.  
**Note:** This error is non-trappable.  
The subscript expression for an array has produced a number that is either less than one or greater than the number of elements in the array.  
Review the program logic that leads up to this statement to determine how the error was made. |
| -1327  | An insert statement could not be prepared for inserting rows into a temporary table used for a report.  
Within the report function, BDL generated an SQL statement to save rows into a temporary table. The dynamic preparation of the statement (see the reference material on the PREPARE statement) produced an error.  
Probably the database tables are not defined now, at execution time, as they were when the program was compiled. Either the database has been changed, or the program has selected a different database than the one that was current during compilation. Possibly the database administrator has revoked SELECT privilege from you for one or more of the tables that the report uses. Look for other error messages that might give more details. |
| -1328  | A temporary table needed for a report could not be created in the selected database.  
**Note:** This error is non-trappable.  
Within the report definition, BDL generated an SQL statement to save rows into a temporary table, but is unable to create the temporary table.  
You must have permission to create tables in the selected database, and there must be sufficient disk space left in the database. You may already have a table in your current database with the same name as the temporary table that the report definition is attempting to create as a sorting table; the sorting table is named "t_reportname". Another possible cause with some database servers is that you have exceeded an operating-system limit on open files. |
| -1329  | A database index could not be created for a temporary database table needed for a report.  
Within the report definition, BDL generated SQL statements to save rows into a temporary table; however, an index failed to be created on the temporary table.  
Probably an index with the same name already exists in the database. (The sorting index is named "i_reportname"; for example, "i_order_rpt".) Possibly no disk space is available in the file system or dbspace. Another possibility with some database servers is that you have exceeded an operating-system limit on open files. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -1330  | A row could not be inserted into a temporary report table.  
Within the report definition, BDL generated SQL statements that would save rows into a temporary table; however, an error occurred while rows were being inserted.  
Probably no disk space is left in the database. Look for other error messages that might give more details. |
| -1331  | A row could not be fetched from a temporary report table.  
Within the report definition, BDL generated SQL statements to select rows from a temporary table. The table was built successfully but now an error occurred while rows were being retrieved from it.  
Almost the only possible cause is a hardware failure or an error in the database server. Check for operating-system messages that might give more details. |
| -1332  | A character variable has referenced subscripts that are out of range.  
**Note:** This error is non-trappable.  
In the current statement, a variable that is used in taking a substring of a character value contains a number less than one or a number greater than the size of the variable, or the first substring expression is larger than the second.  
Review the program logic that leads up to this statement to find the cause of the error. |
| -1335  | A report is accepting output or being finished before it has been started.  
The program executed an OUTPUT TO REPORT or FINISH REPORT statement before it executed a START REPORT.  
Review the program logic that leads up to this statement to find the cause of the error. |
| -1337  | The variable variable-name has been redefined with a different type or length, definition in module-name-1.4gl, redefinition in module-name-2.4gl.  
**Note:** This error is non-trappable.  
The variable that is shown is defined in the GLOBALS section of two or more modules, but it is defined differently in some modules than in others.  
Possibly modules were compiled at different times, with some change to the common GLOBALS file between. Possibly the variable is declared as a module variable in some module that does not include the GLOBALS file. |
| -1338  | The function 'function-name' has not been defined in any module in the program.  
**Note:** This error is non-trappable.  
The named function is called from at least one module of the program, but it is defined in none.  
Verify that the module containing the function is a part of the program, and that the function name is correctly spelled. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -1340  | The error log has not been started.  
**Note:** This error is **non-trappable**.  
The program called the errorlog() function without first calling the startlog() function.  
Review the program logic to find out the cause of this error. |
| -1349  | Character to numeric conversion error.  
A character value is being converted to numeric form for storage in a numeric column or variable; however, the character string cannot be interpreted as a number. It contains some characters other than whitespace, digits, a sign, a decimal, or the letter e, or else the parts are in the wrong order so that the number cannot be deciphered. |
| -1353  | Use '!' to edit TEXT and BYTE fields.  
This is a normal message text used outside an error context. |
| -1355  | Cannot build temporary file.  
A TEXT or BYTE variable has been located in a temporary file using the LOCATE statement.  
The current statement assigns a value into that variable, so BDL attempted to create the temporary file, but an error occurred.  
Possibly no disk space is available, or your account does not have permission to create a temporary file. Look for operating-system error messages that might give more information. |
| -1359  | Read error on blob file 'file-name'.  
The operating system signaled an error during output to a temporary file in which a TEXT or BYTE variable was being saved.  
Possibly the disk is full, or a hardware failure occurred. For more information, look for operating-system messages. |
| -1360  | No PROGRAM= clause for this field.  
No external program has been designated for this field using the PROGRAM attribute in the form-specification file (For Text User Interface mode on ASCII terminals only) |
| -1373  | The field 'field-name' is not in the list of fields in the CONSTRUCT/INPUT statement.  
The built-in function get fldbuf() or field_touched() has been called with the field name shown; however, input from that field was not requested in this CONSTRUCT or INPUT statement. As a result, the function cannot return any useful value.  
Review all uses of these functions, and compare them to the list of fields at the beginning of the statement. |
| -1374  | SQL character truncation or transaction warning.  
The program set WHENEVER WARNING STOP, and a warning condition arose. If the statement involved is a DATABASE statement, the condition is that the database that was just opened uses a transaction log. On any other statement, the condition is that a character value from the database had to be truncated to fit in its destination. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -1375  | SQL NULL value in aggregate or mode ANSI database warning.  
The program set WHENEVER WARNING STOP, and a warning condition arose. If the statement that is involved is a DATABASE statement, the condition is that the database that was just opened is ANSI compliant. On any other statement, the condition is that a null value has been used in the computation of an aggregate value. |
| -1376  | SQL, database server, or program variable mismatch warning.  
The program set WHENEVER WARNING STOP, and a warning condition arose. If the statement that is involved is a DATABASE or CREATE DATABASE statement, the condition is that the database server opened the database. On any other statement, the condition is that a SELECT statement returned more values than there were program variables to contain them. |
| -1377  | SQL float-to-decimal conversion warning.  
The program set WHENEVER WARNING STOP, and a warning condition arose. The condition is that in the database that was just opened, the database server will use the DECIMAL data type for FLOAT values. |
| -1378  | SQL non-ANSI extension warning.  
A database operation was performed that is not part of ANSI SQL, although the current database is ANSI compliant. This message is informational only. |
| -1396  | A report PRINT FILE source file cannot be opened for reading.  
The file that is named in a PRINT FILE statement cannot be opened.  
Review the file name. If it is not in the current directory, you must specify the full path. If the file exists, make sure your account has permissions to read it. |
| -2017  | The character data value does not convert correctly to the field type.  
You have entered a character value (a quoted string) into a field that has a different data type (for example INTEGER); however, the characters that you entered cannot be converted to the type of the field.  
Re-enter the data. |
| -2021  | An illegal color has been specified. Colors 0 through 7 are white, yellow, magenta, red, cyan, green, blue, and black.  
Use one of the color names allowed by the COLOR attribute syntax. |
| -2024  | There is already a record 'record-name' specified.  
A screen record is automatically defined for each table that is used in the ATTRIBUTES section to define a field. If you define a record with the name of a table, it is seen as a duplicate.  
Check that the record-name of every screen record and screen array is unique in the form specification. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -2028  | The symbol 'symbol-name' does not represent a table prefix used in this form.  
In a SCREEN RECORD statement, each component must be introduced by the name of the table as defined in the TABLES section or by the word FORMONLY.  
Review the spelling of the indicated name against the TABLES section, and check the punctuation of the rest of the statement. |
| -2029  | Screen record array 'record-name' has different component sizes.  
The screen record array name has component sizes which either differ from the specified dimension of the array or differ among themselves. This error message appears when one or more of the columns appear a different number of times.  
The dimension of the screen array is written in square brackets that follow its name. Verify that the dimensions of the screen array match the screen fields. |
| -2039  | The attributes AUTONEXT, DEFAULT, INCLUDE, VERIFY, RIGHT and ZEROFILL are not supported for BLOB fields.  
Columns of the data type specified cannot be used in the ways that these attributes imply.  
Check that the table and column names are as you intended, and verify the current definition of the table in the database that the DATABASE statement names. |
| -2041  | The form 'form-name' cannot be opened.  
The form filename cannot be opened. This is probably because it does not exist, or the user does not have read permission.  
Check the spelling of filename. Check that the form file exists in your current directory. If it is in another directory, check that the correct pathname has been provided. On a UNIX™ system, if these things are correct, verify that your account has read permission on the file. |
| -2045  | The conditional attributes of a field cannot depend on the values of other fields.  
The boolean expression in a WHERE clause of a COLOR attribute can use only the name of that field and constants.  
Revise this attribute, and recompile the form. |
| -2100  | Field 'field-name' has validation string error, String = string.  
One of the formatting or validation strings that is stored in the syscolval or syscolatt tables is improperly coded. The string is shown as is the field to which it applies.  
Update the string in the tables. |
| -2810  | The name 'database-name' is not an existing database name.  
This name, which was found in the DATABASE statement at the start of the form specification, is not a database that can be found.  
Check the spelling of the database name and the database entries in the FGLPROFILE file. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -2820  | The label name between brackets is incorrectly given or the label is missing.  
In the layout section of a form specification, it is recommended that brackets contain a simple name. Instead, they contain spaces or an invalid name.  
Check the layout section of the form for invalid form item labels. |
| -2830  | A left square bracket has been found on this line, with no right square bracket to match it.  
Every left square bracket field delimiter must have a right square bracket delimiter on the same line.  
Review the form definition file to make sure all fields are properly marked. |
| -2840  | The field label 'label-name' was not defined in the form.  
The indicated name appears at the left of this ATTRIBUTES statement, but it does not appear within brackets in the SCREEN section.  
Review the field tags that have been defined to see why this one was omitted. |
| -2843  | The column 'column-name' does not appear in the form specification.  
A name in this ATTRIBUTES statement has not been defined in the form specification.  
Check that all names in the statement are spelled correctly and defined properly. |
| -2846  | The field 'field-name' is not a member of the table 'table-name'.  
Something in this statement suggests that the name shown is part of this table, but that is not true in the current database.  
Review the spelling of the two names. If they are as you intended, check that the correct database is in use and that the table has not been altered. |
| -2859  | The column 'column-name' is a member of more then one table -- you must specify the table name.  
Two or more tables that are named in the TABLES section have columns with the name shown.  
You must make clear which table you mean. To do this, write the table name as a prefix of the column name, as table.column, wherever this name is used in the form specification. |
| -2860  | There is a column/value type mismatch for 'column-name'.  
This statement assigns a value to the field with the DEFAULT clause or uses its value with the INCLUDE clause, but it does so with data that does not agree with the data type of the field.  
Review the data type of the field (which comes from the column with which it is associated), and make sure that only compatible values are assigned. |
| -2862  | The table 'table-name' cannot be found in the database.  
The indicated table does not exist in the database that is named in the form.  
Check the spelling of the table name and database name. If they are as you intended, either you are not using the version of the database that you expected, or the database has been changed. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -2863  | The column 'column-name' does not exist among the specified tables.  
The tables that are specified in the TABLES section of the form exist, but column-name, which is named in the ATTRIBUTES section, does not.  
Check its spelling against the actual table. Possibly the table was altered, or the column was renamed. |
| -2864  | The table 'table-name' is not among the specified tables.  
The indicated table is used in this statement but is not defined in the TABLES section of the form specification.  
Check its spelling; if it is as you intended, add the table in the TABLES section. |
| -2865  | The column 'column-name' does not exist in the table 'table-name'.  
Something in this statement implies that the column shown is part of the indicated table (most likely the statement refers to table-name.column-name); however, it is not defined in that table.  
Check the spelling of both names. If they are as you intended, then make sure that the database schema (.sch) is up to date; possibly the table has been altered or the column renamed, and thus needs a new db schema extraction with the fgldbsch tool. |
| -2892  | The column 'column-name' appears more than once. If you wish a column to be duplicated in a form, use the same display field label.  
The same column name is listed in the ATTRIBUTES section more than once.  
The expected way to display the same column in two or more places is to put two or more fields in the screen layout, each with the same tag-name. Then put a single statement in the ATTRIBUTES section to associate that tag-name with the column name. The current column value will be duplicated in all fields. If you intended to display different columns that happen to have the same column-names, prefix each column with its table-name. |
| -2893  | The display field label 'label-name' appears more than once in this form, but the lengths are different.  
You can put multiple copies of a field in the screen layout (all will display the same column), but all copies must be the same length.  
Review the form definition to make sure that, if you intended to have multiple copies of one field, all copies are the same. |
| -2975  | The display field label 'label-name' has not been used.  
A field tag has been declared in the screen section of the form- specification file but is not defined in the attributes section.  
Check your form-specification file. |
| -2992  | The display label 'label-name' has already been used.  
The forms compiler indicates that name has been defined twice. These names must be defined uniquely in the form specification.  
Review all uses of the name to see if one of them is incorrect. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -2997  | See error number error-num.  
The database server returned an error that is shown.  
Look up the shown error in the database server documentation. |
| -4303  | A blob variable or cursor name expected.  
The argument to the FREE statement must be the name of a cursor or prepared statement or, in BDL, the name of a variable with the BYTE or TEXT data type.  
Check the name used after the FREE keyword. |
| -4307  | The number of variables and/or constants in the display list does not match the number of form fields in the display destination.  
There must be exactly as many items in the list of values to display as there are fields listed following the TO keyword in this statement.  
Review the statement. |
| -4308  | The number of input variables does not match the number of form fields in the screen input list.  
Your INPUT statement must specify the same number of variables as it does fields.  
When checking this, keep in mind that when you refer to a record using an asterisk or THRU, it is the same as listing each record component individually. |
| -4309  | Printing cannot be done within a loop or CASE statement contained in report headers or trailers.  
BDL needs to know how many lines of space will be devoted to page headers and trailers; otherwise, it does not know how many detail rows to allow on a page. Since it cannot predict how many times a loop will be executed, or which branch of a CASE will be executed, it forbids the use of PRINT in these contexts within FIRST PAGE HEADER, PAGE HEADER, and PAGE TRAILER sections.  
Re-arrange the code to place the PRINT statement where it will always be executed. |
| -4319  | The symbol 'symbol-name' has been defined more than once.  
The variable that is shown has appeared in at least one other DEFINE statement before this one.  
Review your code. If this DEFINE is within a function or the MAIN section, the prior one is also. If this DEFINE is outside any function, the prior one is also outside any function; however, it might be within the file included by the GLOBALS statement. |
| -4320  | The symbol 'symbol-name' is not the name of a table in the specified database.  
The named table does not appear in the database.  
Review the statement. The table name may be spelled wrong in the program, or the table might have been dropped or renamed since the last time the program was compiled. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -4322  | The symbol 'symbol-name' is not the name of a column in the specified database.  
      The preceding statement suggests that the named column is part of a certain table in the specified database. The table exists, but the column does not appear in it.  
      Check the spelling of the column name. If it is spelled as you intended, then either the table has been altered, or the column renamed, or you are not accessing the database you expected. |
| -4323  | The variable 'variable-name' is too complex to be used in an assignment statement.  
      The named variable is a complex variable like a record or an array, which cannot be used in a LET statement.  
      You must assign groups of components to groups of components using asterisk notation. |
| -4324  | The variable 'variable-name' is not a character type, and cannot be used to contain the result of concatenation.  
      This statement attempts to concatenate two or more character strings (using the comma as the concatenation operator) and assign the result to the named variable. Unfortunately, it is not a character variable, and automatic conversion from characters cannot be performed in this case.  
      Assign the concatenated string to a character variable; then, if you want to treat the result as numeric, assign the string as a whole to a numeric variable. |
| -4325  | The source and destination records in this record assignment statement are not compatible in types and/or length.  
      This statement uses asterisk notation to assign all components of one record to the corresponding components of another; however, the components do not correspond. Note that BDL matches record components strictly by position, the first to the first, second to second, and so on; it does not match them by name.  
      If the source and destination records do not have the same number and type of components, you will have to write a simple assignment statement for each component. |
| -4328  | The variable 'variable-name' is too complex to be used as the destination of a return from a function.  
      The named variable is too complex to be assigned directly in a RETURNING clause.  
      Individual members of the complex variable must be returned separately. |
| -4330  | RETURN statements can be executed only within functions.  
      This error occurs when the RETURN clause is used in an invalid context such as MAIN, REPORT or DIALOG blocks. |
| -4333  | The function 'function-name' has already been called with a different number of parameters.  
      Earlier in the program, there is a call to this same function or event with a different number of parameters in the parameter list. At least one of these calls must be in error.  
      Examine the FUNCTION statement for the named function to find out the correct number of parameters. Then examine all calls to it, and make sure that they are written correctly. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -4334  | The variable 'variable-name' in its current form is too complex to be used in this statement.  
The variable has too many component parts. Only simple variables (those that have a single component) can be used in this statement.  
If variable-name is an array, you must provide a subscript to select just one element. If it is a record, you must choose just one of its components; however, if this statement permits a list of variables, as in the INITIALIZE statement, you can use asterisk or THRU notation to convert a record name into a list of components. |
| -4335  | The symbol 'field-name' is not an element of the record 'record-name'.  
The field name used in a record.field expression is not identified as a member of the record variable.  
Find the definition of the record (it may be in the GLOBALS file), verify the names of its fields, and correct the spelling of field-name. |
| -4336  | The parameter 'param-name' has not been defined within the function or report.  
The name variable-name appears in the parameter list of the FUNCTION statement for this function; however, it does not appear in a DEFINE statement within the function. All parameters must be defined in their function before use.  
Review your code. Possibly you wrote a DEFINE statement but did not spell variable-name the same way in both places. |
| -4338  | The symbol 'symbol-name' has already been defined once as a parameter.  
The name that is shown appears in the parameter list of the FUNCTION statement and in at least two DEFINE statements within the function body.  
Review your code. Only one appearance in a DEFINE statement is permitted. |
| -4340  | The variable 'variable-name' is too complex a type to be used in an expression.  
In an expression, only simple variables (those that have a single component) can be used.  
If the variable indicated is an array, you must provide a subscript to select just one element. If it is a record or object, you must choose just one of its components. |
| -4341  | Aggregate functions are only allowed in reports and SELECT statements.  
Aggregate functions such as SUM, AVG, and MAX can only appear in SQL statements and within certain statements that you use in the context of a report body. They are not supported in ordinary expressions in program statements.  
Review the code and check that the aggregate functions are in an SQL statement or in the correct blocks of the REPORT routine. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4343</td>
<td>Subscripting cannot be applied to the variable 'variable-name'. You tried to use a [x,y] subscript expression with a variable that is neither a character data type or an array type. Check the variable data type and make sure it can be used with a subscript expression.</td>
</tr>
<tr>
<td>-4347</td>
<td>The variable 'variable-name' is not a record. It cannot reference record elements. In this statement variable-name appears followed by a dot, followed by another name. This is the way you would refer to a component of a record variable; however, variable-name is not defined as a record. Either you have written the name of the wrong variable, or else variable-name is not defined the way you intended.</td>
</tr>
<tr>
<td>-4353</td>
<td>The type of this ORDER BY or GROUP item specified for the report is not valid for sorting. A REPORT routine defines an ORDER BY or GROUP clause using a variable defined with a type such as TEXT and BYTE, that is too complex to be used in comparisons. As result, columns with such types cannot be used to sort or group rows. Review the report and sort or group rows by using items defined with simple data types.</td>
</tr>
<tr>
<td>-4356</td>
<td>A PAGE HEADER has already been specified within this report. Only one PAGE HEADER control block is allowed in a REPORT. Search for other PAGE HEADER sections and combine all statements in a unique control block.</td>
</tr>
<tr>
<td>-4357</td>
<td>A PAGE TRAILER has already been specified within this report. Only one PAGE TRAILER control block is allowed in a REPORT. Search for other PAGE TRAILER sections and combine all statements in a unique control block.</td>
</tr>
<tr>
<td>-4358</td>
<td>A FIRST PAGE HEADER has already been specified within this report. Only one FIRST PAGE TRAILER control block is allowed in a REPORT. Search for other FIRST PAGE TRAILER sections and combine all statements in a unique control block.</td>
</tr>
<tr>
<td>-4359</td>
<td>An ON EVERY ROW clause has already been specified within this report. Only one ON EVERY ROW control block is allowed in a REPORT. Search for other ON EVERY ROW sections and combine all statements in a unique control block.</td>
</tr>
<tr>
<td>-4360</td>
<td>An ON LAST ROW clause has already been specified within this report. Only one ON LAST ROW control block is allowed in a REPORT. Search for other ON LAST ROW sections and combine all statements in a unique control block.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>-4361</td>
<td>Group aggregates can occur only in AFTER GROUP clauses. The aggregate functions that apply to a group of rows (GROUP COUNT/PERCENT/SUM/AVG/MIN/MAX) can only be used at the point in the report when a complete group has been processed, namely, in the AFTER GROUP control block. Make sure that the AFTER GROUP block exists and was recognized. If you need the value of a group aggregate at another time (for instance, in a PAGE TRAILER control block), you can save it in a module variable with a LET statement in the AFTER GROUP block.</td>
</tr>
<tr>
<td>-4363</td>
<td>The report cannot skip lines while in a loop within a header or trailer. BDL needs to know how many lines of space will be devoted to the page header and trailer (otherwise it does not know how many detail rows to allow on the page). It cannot predict how many times a loop will be executed, so it has to forbid the use of SKIP statements in loops in the PAGE HEADER, PAGE TRAILER, and FIRST PAGE HEADER sections. Review the report header or trailer to avoid SKIP in loops.</td>
</tr>
<tr>
<td>-4369</td>
<td>The symbol 'symbol-name' does not represent a defined variable. The name shown appears where a variable would be expected, but it does not match any variable name in a DEFINE statement that applies to this context. Check the spelling of the name. If it is the name you intended, look back and find out why it has not yet been defined. Possibly the GLOBALS statement has been omitted from this source module, or it names an incorrect file. Possibly this code has been copied from another module or another function, but the DEFINE statement was not copied also.</td>
</tr>
<tr>
<td>-4371</td>
<td>Cursors must be uniquely declared within one program module. In the statement DECLARE cursor-name CURSOR, the identifier cursor-name can be used in only one DECLARE statement in the source file. This is true even when the DECLARE statement appears inside a function. Although a program variable made with the DEFINE statement is local to the function, a cursor within a function is still global to the whole module Search for duplicated cursor names and change the name to have unique identifiers.</td>
</tr>
<tr>
<td>-4372</td>
<td>The cursor 'cursor-name' has not yet been declared in this program. The name shown appears where the name of a declared cursor or a prepared statement is expected; however, no cursor (or statement) of that name has been declared (or prepared) up to this point in the program. Check the spelling of the name. If it is the name you intended, look back in the program to see why it has not been declared. Possibly the DECLARE statement appears in a GLOBALS file that was not included.</td>
</tr>
<tr>
<td>-4374</td>
<td>This type of statement can only be used within a MENU statement. This statement only makes sense within the context of a MENU statement. Review the program in this vicinity to see if an END MENU statement has been misplaced. If you intended to set up the appearance of a menu before displaying it, use a BEFORE MENU block within the scope of the MENU.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| -4375  | The page length is too short to cover the specified page header and trailer lengths.  

A REPORT defines page header and trailer sections with a total number of lines that is not sufficiently less than the specified page length in order to print some detail lines.  

Review the [FIRST] PAGE HEADER and PAGE TRAILER blocks to use less lines or increase the page length. |
| -4379  | The input file 'file-name' cannot be opened.  

Either the file does not exist, or, on UNIX™, your account does not have permission to read it.  
Possibly the filename is misspelled, or the directory path leading to the file was specified incorrectly. |
| -4380  | The listing file 'file-name' cannot be created.  

The file cannot be created.  
Check that the directory path leading to the file is specified correctly and, on UNIX™ systems, that your account has permission to create a file in that directory. Look for other, more explicit, error messages from the operating system. Possibly the disk is full, or you have reached a limit on the number of open files. |
| -4382  | Record variables that contain array type elements may not be referenced by the ".*" or THROUGH shorthand, or used as a function parameter.  

The .* and THROUGH/THRU notation is used to expand a record with an array member.  
It is allowed to define a record with an array member, but this element must always be used with its full designation of record.array[n]. The .* or THROUGH/THRU notation only expands simple members of the record. |
| -4383  | The elements 'name-1' and 'name-2' do not belong to the same record.  

The two names shown are used where two components of one record are required; however, they are not components of the same record.  
Check the spelling of both names. If they are spelled as you intended, go back to the definition of the record and see why it does not include both names as component fields. |
| -4402  | In this type of statement, subscripting may be applied only to array.  

The statement contains a name followed by square brackets, but the name is not that of an array variable.  
Check the punctuation of the statement and the spelling of all names. Names that are subscripted must be arrays. If you intended to use a character substring in this statement, you will have to revise the program. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -4403  | The number of dimensions for the variable 'variable-name' does not match the number of subscripts.  

In this statement, the array whose name is shown is subscripted by a different number of dimensions than it was defined to have.  

Check the punctuation of the subscript. If it is as you intended, then review the DEFINE statement where variable-name is defined. |
| -4410  | There is a numeric constant in the previous line that is too large or too small.  

The compiler is unable to process a numeric constant because it is too big or too small to represent a valid SMALLINT, INTEGER, BIGINT or DECIMAL constant.  

Check the number of digits and the punctuation of the numeric constant. Make sure you have not typed a letter for a digit for example. |
| -4414  | The label 'label-name' has been used but has never been defined within the above main program or function.  

A GOTO or WHENEVER statement refers to the label shown, but there is no corresponding LABEL statement in the current function or main program.  

Check the spelling of the label. If it is as you intended it, find and inspect the LABEL statement that defines it. You cannot transfer out of a program block with GOTO; labels must be defined in the same function body where they are used. |
| -4415  | An ORDER BY or GROUP item specified within a report must be one of the report parameters.  

The names used in a ORDER BY, AFTER GROUP OF, or BEFORE GROUP OF statement must also appear in the parameter list of the REPORT statement. It is not possible to order or group based on a global variable or other expression.  

Check the spelling of the names in the statement and compare them to the REPORT statement. |
| -4416  | There is an error in the validation string: 'validation-string'.  

The validation string in the syscolval table is not correct.  

Change the appropriate DEFAULT or INCLUDE value in the syscolval table. |
| -4417  | This type of statement can be used only in a report.  

Statements such as PRINT, SKIP, or NEED are meaningful only within the body of a report function, where there is an implicit report listing to receive output.  

Remove the report specific statement from the code which is not in a report body. |
| -4418  | The variable used in the INPUT ARRAY or DISPLAY ARRAY statement must be an array.  

The name following the words DISPLAY ARRAY or INPUT ARRAY must be that of an array of records.  

Check the spelling of the name. If it is as you intended, find and inspect the DEFINE statement to see why it is not an array. (If you want to display or input a simple variable or a single element of an array, use the DISPLAY or INPUT statement.) |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -4420  | The number of lines printed in the IF part of an IF-THEN-ELSE statement of a header or trailer clause must equal the number of lines printed in the ELSE part.  
   The runtime system needs to know how many lines will be filled in header and trailer sections (otherwise it does not know how many detail rows to put on the page). Because it cannot tell which part of an IF statement will be executed, it requires that both produce the same number of lines of output.
   Use the same number of occurrences of PRINT statements in each block of the IF statement. |
| -4425  | The variable 'variable-name' has not been defined like the table 'table-name'.  
   The named variable has been used in the SET clause of an UPDATE statement or in the VALUES clause of an INSERT statement, but it was not define LIKE the table being modified.  
   As a result, then runtime system cannot associate record components with table columns.
   Make sure the schema file is up to date and check that the variable was defined like the table. You can also rewrite the UPDATE or INSERT statement with a different syntax to show the explicit relationship between column names and record components. |
| -4440  | The field 'field-name-1' precedes 'field-name-2' in the record 'record-name' and must also precede it when used with the THROUGH shorthand.  
   The THROUGH or THRU shorthand requires you to give the starting and ending fields as they appear in physical sequence in the record.
   Check the spelling of the names; if they are as you intended, then refer to the VARIABLE statement where the record was defined to see why they are not in the sequence you expected. |
| -4447  | 'key-name' is not a recognized key value.  
   The key name used in an ON KEY clause is not known by the compiler.
   Search the documentation for possible key names (F1-F255, Control-?). |
| -4448  | Cannot open the file 'file-name' for reading or writing.  
   Note: This error is non-trappable.  
   The file cannot be opened because:
   • You do not have permissions. Verify that your account has permission to read or write to the file.
   • The file cannot be found. Verify that the file name is correctly spelled and that it exist. Also the source of the file path, such as an environment or program variable, that would give you the proper path may not be set properly. Check your environment configuration and review your code. |
| -4452  | The function (or report) 'function-name' has already been defined.  
   Each function (or report, which is similar to a function) must have a unique name within the program.
   Change the function or report name. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -4457  | You may have at most 4 keys in the list.  
          An interactive instruction defines a ON KEY() clause with more that 4 keys.  
          Remove keys from the list. |
| -4458  | One dimension of this array has exceeded the limit of 65535.  
          The program is using a static array with a dimension that exceeds the limit.  
          Use a dimension below the 65535 limit. |
| -4463  | The NEXT FIELD statement can only be used within an INPUT or CONSTRUCT statement.  
          The NEXT FIELD statement is used outside an INPUT, INPUT ARRAY or CONSTRUCT statement.  
          Remove the NEXT FIELD statement from that part of the code. |
| -4464  | The number of columns must match the number of values in the SET clause of an UPDATE statement.  
          In an UPDATE statement, the number of values used does not match the number of columns.  
          Check for the table definition, then either add or remove values or columns from the UPDATE statement. |
| -4476  | Record members may not be used with database column substring.  
          This statement has a reference of the form name1.name2[...]. This is the form in which you would refer to a substring of a column: table.column[...] ; however, the names are not a table and column in the database, so BDL presumes they refer to a field of a record.  
          Inspect the statement and determine what was intended: a reference to a column or to a record.  
          If it is a column reference, verify the names of the table and column in the database. If it is a record reference, verify that the record and component are properly defined. |
| -4477  | The variable 'variable-name' is an array. You must specify one of its elements in this statement.  
          You tried to use an array without element specification in a SQL statement.  
          Use one of the members of the array. |
| -4485  | Only blob variables of type BYTE or TEXT may be used in a LOCATE statement.  
          The LOCATE statement is using a variable defined with a data type different from BYTE or TEXT.  
          Make sure the variables used with LOCATE are defined as BYTE or TEXT. |
| -4488  | The program cannot CONTINUE or EXIT statement-type at this point because it is not immediately within statement-type statement.  
          This CONTINUE or EXIT statement is not appropriate in its context.  
          Review your code. Possibly the statement is misplaced, or the statement type was specified incorrectly. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -4489  | A variable used in the above statement must be a global variable.  
A REPORT routine is defining an OUTPUT REPORT TO using a local function variable or report parameter.  
Review the report clause to use a global or module variable instead. |
| -4490  | You cannot have multiple BEFORE clauses for the same field.  
You cannot specify more than one BEFORE FIELD clause for the same field.  
Review your code to eliminate multiple BEFORE FIELD clauses. |
| -4491  | You cannot have multiple AFTER clauses for the same field.  
You cannot specify more than one AFTER FIELD clause for the same field.  
Review your code to eliminate multiple AFTER FIELD clauses. |
| -4534  | Wordwrap may not be used within report headers or trailers.  
The report routine uses the WORDWRAP clause in the FIRST PAGE HEADER, PAGE HEADER or PAGE TRAILER sections.  
Remove the WORDWRAP clause from the expression. |
| -4631  | Startfield of DATETIME or INTERVAL qualifiers must come earlier in the time-list than its endfield.  
The qualifier for a DATETIME or INTERVAL consists of start TO end, where the start and end are chosen from this list: YEAR MONTH DAY HOUR MINUTE SECOND FRACTION.  
The keyword for the start field must come earlier in the list than, or be the same as, the keyword for the end field.  
Check the order of the startfield and endfield qualifiers. For example, qualifiers of DAY TO FRACTION and MONTH TO MONTH are valid but one of MINUTE TO HOUR is not. |
| -4632  | Parenthetical precision of FRACTION must be between 1 and 5. No precision can be specified for other time units.  
In a DATETIME qualifier only the FRACTION field may have a precision in parentheses, and it must be a single digit from 1 to 5.  
Check the DATETIME qualifiers in the current statement; one of them violates these rules. The first field of an INTERVAL qualifier may also have a parenthesized precision from 1 to 5. |
| -4652  | The function 'function-name' can only be used within an INPUT or CONSTRUCT statement.  
The function shown is being used outside of an INPUT or CONSTRUCT statement; however, it returns a result that is only meaningful in the context of INPUT or CONSTRUCT.  
Review the code to make sure that an END INPUT or END CONSTRUCT statement has not been misplaced. Review the operation and use of the function to make sure you understand it. |
-4653
No more than one BEFORE or AFTER INPUT/CONSTRUCT clause can appear in an INPUT/CONSTRUCT statement.

There may be only one BEFORE block of statements to initialize each of these statement types. Make sure that the scope of all your INPUT, CONSTRUCT and MENU statements is correctly marked with END statements. Then combine all the preparation code into a single BEFORE block for each one.

-4656
CANCEL INSERT can only be used in the BEFORE INSERT clause of an INPUT ARRAY statement.

The CANCEL INSERT statement is being used outside of the BEFORE INSERT clause of an INPUT ARRAY.

Review the code to make sure that CANCEL INSERT has not been used anywhere except in the BEFORE INSERT clause.

-4657
CANCEL DELETE can only be used in the BEFORE DELETE clause of an INPUT ARRAY statement.

The CANCEL DELETE statement is being used outside of BEFORE DELETE clause of an INPUT ARRAY.

Review the code to make sure that CANCEL DELETE has not been used anywhere except in the BEFORE DELETE clause.

-4668
The report output, specified by a START REPORT statement, is not any of file, pipe, screen, printer, pipe in line mode, or pipe in form mode.

The output of a report can be sent only to any of file, pipe (in form or line modes), screen, or printer.

Check the START REPORT instruction and make sure that the OUTPUT clause specifies one of the supported values.

-4900
This syntax is not supported here. Use [screenrecordname.]screenfieldname.

The field name specification in a BEFORE FIELD or AFTER FIELD is not valid.

Check for the field name and use [screenrecordname.]screenfieldname syntax.

-4901
Fatal internal error: description ( line-number ).

This generic error occurs when the fglcomp compiler cannot identify the problem and must stop processing the source.

Check the code near the line displayed in the error message.

-6001
The license manager daemon cannot be started.

Note: This error is non-trappable.

This error occurs when a process creation fails during the start of the license manager.

Increase the maximum number of processes allowed (ulimit)
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -6012  | Cannot get license information. Check your environment and the license (run 'fglWrt -a info').  
**Note:** This error is non-trappable.  
See error -6015. |
| -6013  | Time limited version: time has expired.  
**Note:** This error is non-trappable.  
The license installed is a license with time limit and time has expired. The program can not start.  
Contact your distributor or support center. |
| -6014  | Your serial number is not valid for this version.  
**Note:** This error is non-trappable.  
The license serial number is invalid for this version of the software.  
Contact your distributor or support center. |
| -6015  | Cannot get license information. Check your environment and the license (run 'fglWrt -a info').  
**Note:** This error is non-trappable.  
It is not possible for the application to check the license validity.  
• License manager:  
  • The license may not have been installed  
  • The license controller can not communicate with the license manager. Check that the license manager is started and check that the FGLPROFILE entries flm.server and flm.service contain valid information.  
  • The directory $FLMDIR/lock and all the files below must have read/write permission.  
• License controller:  
  • The license may not have been installed.  
  • The directory $FGLDIR/lock and all the files below must have read/write permission. |
| -6016  | Cannot get information for license (Error error-num). Check your environment and the license (run 'fglWrt -a info').  
**Note:** This error is non-trappable.  
The application is unable to check the license validity.  
See error -6015. |
| -6017  | User limit exceeded. Cannot run this program.  
**Note:** This error is non-trappable.  
The maximum number of users allowed by the license has been reached. The program can not start.  
Contact your distributor or support center. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -6018  | Cannot access internal data file. Cannot continue this program. Please, check your environment variablename.  
**Note:** This error is non-trappable.  
When a client computer starts an application on the server, the application stores data in the $FGLDIR/lock directory. The client must have permission to create and delete files in this directory.  
• Do not remove or modify files contained in the directory $FGLDIR/lock  
• Change the permissions of the $FGLDIR/lock directory, or connect to the server with a user name having the correct permissions. |
| -6019  | This demonstration version allows one user only.  
**Note:** This error is non-trappable.  
The demonstration version is designed to run with only one user. Another user or another graphical daemon is currently active.  
Wait until the user stops the current program, or use the same graphical daemon. |
| -6020  | Installation: Cannot open 'file-name'.  
**Note:** This error is non-trappable.  
A file is missing or the permissions are not set for the current user.  
Check that the file permissions are correct for the user trying to execute the application. If the file is missing, re-install the compiler package. |
| -6021  | You are not allowed to be connected. Check your environment and the license (run 'fglWrt/greWrt -a info').  
**Note:** This error is non-trappable.  
Check the license installation. |
| -6022  | Demonstration time has expired. Please, run this program again.  
**Note:** This error is non-trappable.  
The runtime demonstration version is valid only for a few minutes after you have started a program.  
Restart the program. |
| -6023  | C-code generation is not allowed with the demonstration program.  
**Note:** This error is non-trappable.  
This type of license does not allow to produce C code. |
| -6025  | Modules compiled with Genero Mobile require a Genero Mobile license at runtime.  
**Note:** This error is non-trappable.  
The current license used at runtime must be a Genero Mobile license. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -6025  | Demonstration time has expired. Please, contact your vendor.  
**Note:** This error is non-trappable.  
The demonstration version of the product has a time limit of 30 days.  
Either reinstall a new demonstration version, or call your software vendor to purchase a permanent license. |
| -6026  | Bad link for runner demonstration. Please, retry or rebuild your runner.  
**Note:** This error is non-trappable.  
The runner is corrupted. |
| -6027  | Cannot access license server. Please check the following:  
- the license server entry in your resource file. (service port)  
- the license server host.  
- the license server program.  
**Note:** This error is non-trappable.  
You have not specified a value for the environment variable [fglic|fls|flm].server in the $FGLDIR/etc/fglprofile file.  
Check the FGLPROFILE file for the entry point [fglic|fls|flm].server and specify the name of the computer that runs the License Manager. |
| -6028  | The Genero Mobile pcode size limit has been reached. Please, contact your vendor.  
**Note:** This error is non-trappable.  
The current license does not allow larger programs. |
| -6029  | Unknown parameter 'param-name' for checking.  
**Note:** This error is non-trappable.  
The command line of the fglWrt or flmprg tool contains an unknown parameter.  
Check your command-line parameters and retry the command. |
| -6031  | Temporary license license-number has expired.  
**Note:** This error is non-trappable.  
Your temporary runtime license has expired.  
Call your software vendor to get a new license. |
| -6032  | command-name: illegal option: 'option-name'.  
**Note:** This error is non-trappable.  
You are not using a valid option for the specified command.  
Check your command line syntax and try again. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -6033  | `command-name: 'option-name'` option requires an argument.  
**Note:** This error is **non-trappable**.  
You cannot use this option of the tool without a parameter.  
Check your command line and try again. |
| -6034  | Warning! This is a temporary license, installation number is `installation-number'`.  
**Note:** This error is **non-trappable**.  
You have installed a temporary license of 30 days. You will have to enter an installation key before the end of this period if you want to keep on running the program.  
This is only a warning message. |
| -6035  | Cannot read in directory  
**Note:** This error is **non-trappable**.  
The compiler cannot access the `$FGLDIR/lock` directory. The current user must have read and write permissions in this directory.  
Give the current user read and write permissions to the `$FGLDIR/lock` directory. |
| -6036  | This license requires a valid subscription key. Check your environment (run `grxWrt -a info`)  
**Note:** This error is **non-trappable**.  
Check the Genero Report license. |
| -6041  | Can not retrieve network interface information.  
**Note:** This error is **non-trappable**.  
An error occurred while retrieving network interface information.  
Restart your program. If this does not solve your problem, contact your distributor. |
| -6042  | MAC Address has changed.  
**Note:** This error is **non-trappable**.  
The MAC address of the host has changed since the license was first installed.  
The license must be reinstalled, or restore the old MAC address. |
| -6043  | The testing period is finished. You must install a new license.  
**Note:** This error is **non-trappable**.  
The test time license of has expired.  
Call your software vendor to purchase a new license. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -6044  | IP Address has changed.  
  **Note:** This error is non-trappable.  
  The IP Address of the host has changed.  
  Restore the IP address of the host, or reinstall the license. This is no longer checked by the latest versions of the license controller. |
| -6045  | Host name has changed.  
  **Note:** This error is non-trappable.  
  The host name has changed.  
  Restore the host name or reinstall the license. This is no longer checked by the latest versions of the license controller. |
| -6046  | Could not get file reference number information.  
  **Note:** This error is non-trappable.  
  Information about the license file can not be obtained.  
  Reinstall the license. Contact your distributor. |
| -6047  | The device number of the license file has changed.  
  **Note:** This error is non-trappable.  
  The license file has been touched. The license is no longer valid.  
  Reinstall the license. Contact your distributor. |
| -6048  | The file reference number of the license file has changed.  
  **Note:** This error is non-trappable.  
  The license file has been touched. The license is no longer valid.  
  Reinstall the license. Contact your distributor. |
| -6049  | This product is licensed for runtime only. No compilation is allowed.  
  **Note:** This error is non-trappable.  
  You have a runtime license installed with this package. You cannot compile BDL source code modules with this license.  
  If you want to compile .4gl source code, you must purchase and install a development license. Contact your distributor. |
| -6050  | Temporary license license-number expired. Please contact your vendor.  
  **Note:** This error is non-trappable.  
  A license with a time limit has been installed and the license has expired.  
  Install a new license to activate the product. Contact your distributor. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -6051  | Temporary license license-number expired. Please contact your vendor.  
**Note:** This error is non-trappable.  
A license with a time limit has been installed and the license has expired.  
Install a new license to activate the product. Contact your distributor. |
| -6052  | Temporary license license-number expired. Please contact your vendor.  
**Note:** This error is non-trappable.  
A license with a time limit has been installed and the license has expired.  
Install a new license to activate the product. Contact your distributor. |
| -6053  | Installation path has changed. It must hold the original installation path.  
**Note:** This error is non-trappable.  
The value of FGLDIR or the location of FGLDIR has been changed.  
Ask the person who installed the product for the location of the original installation directory and then set the FGLDIR environment variable. |
| -6054  | Cannot read a license file. Check installation path and your environment. Verify if a license is installed.  
**Note:** This error is non-trappable.  
The file that contains the license is not readable by the current user.  
- License controller: Check that the FGLDIR environment variable is correctly set and that the file $FGLDIR/etc/f4gl.sn is readable by the current user.  
- License manager: Check that the file $FLMDIR/etc/license/lic?????.dat is readable by the current user. |
| -6055  | Cannot update a license file. Check installation path and your environment. Verify if a license is installed.  
**Note:** This error is non-trappable.  
The file that contains the license cannot be overwritten by the current user.  
- License controller: Check that the FGLDIR environment variable is correctly set and that the file $FGLDIR/etc/f4gl.sn is writable by the current user.  
- License manager: Check that the file $FLMDIR/etc/license/lic?????.dat is writable by the current user. |
| -6056  | Cannot write into a license file. Please check your rights.  
**Note:** This error is non-trappable.  
The file that contains the license cannot be overwritten by the current user.  
- License controller: Check that the FGLDIR environment variable is correctly set and that the file $FGLDIR/etc/f4gl.sn is writable by the current user.  
- License manager: Check that the file $FLMDIR/etc/license/lic?????.dat is writable by the current user. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -6057 | Cannot read a license file. Check installation path and your environment. Verify if a license is installed.  
*Note:* This error is non-trappable.  
The file that contains the license cannot be read by the current user.  
Check that the current user can read the file $FGLDIR/etc/f4gl.sn. Also check that the FGLDIR environment variable is set correctly. |
| -6058 | Incorrect license file format. Verify if a license is installed.  
*Note:* This error is non-trappable.  
The file that contains the license has been corrupted.  
Reinstall the license. If you have a backup of the current installation of Genero Business Development Language, restore the files located in the $FGLDIR/etc directory. |
| -6059 | Incorrect license file format. Verify if a license is installed.  
*Note:* This error is non-trappable.  
The file that contains the license has been corrupted.  
Reinstall the license. If you have a backup of the current installation of Genero Business Development Language, restore the files located in the $FGLDIR/etc directory. |
| -6061 | License 'license-number' not installed.  
*Note:* This error is non-trappable.  
The license shown is not installed.  
Reinstall the license. |
| -6062 | No installed license has been found for 'license-number'.  
*Note:* This error is non-trappable.  
The add-user license can not be installed. No main license found to add users.  
Contact your distributor. |
| -6063 | License 'license-number' is already installed.  
*Note:* This error is non-trappable.  
The license shown is already installed.  
No particular action to be taken. |
| -6064 | The resource 'flm.license.number' is required to use the license manager.  
*Note:* This error is non-trappable.  
In order to use a license manager, the FGLPROFILE entry described in the error message must exist and define a license number. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -6065  | The resource 'flm.license.key' is required to use the license manager.  
Note: This error is non-trappable.  
In order to use a license manager, the FGLPROFILE entry described in the error message must exist and define a license key. |
| -6066  | License 'license-number' cannot be installed over 'license-number'.  
Note: This error is non-trappable.  
The add-user license does not match the main license. The add-user license can not be installed. Contact your distributor. |
| -6067  | You need a installed license if you want to add users.  
Note: This error is non-trappable.  
The add-user license must be installed after the main license. Install the main license before the add-user license. If this does not solve your problem, contact your distributor. |
| -6068  | No license installed.  
Note: This error is non-trappable.  
There is no license installed for Genero Business Development Language. Install a license. If a license is already installed, check that the $FGLDIR environment variable is set correctly. |
| -6069  | Cannot uninstall the license.  
Note: This error is non-trappable.  
There was a problem during the uninstall of the Genero Business Development Language license. Check whether the FGLDIR environment variable is correctly set in your environment and the current user has permission to delete files in the $FGLDIR/etc directory. |
| -6070  | The license server entry must be set in your resource file in order to reach the license server.  
Note: This error is non-trappable.  
You are using the remote license process and you have set the value of fgllic.server, in $FGLDIR/etc/fglprofile, to localhost or to the 127.0.0.1 address. You must use the real IP address of the computer even if it is the local computer. |
| -6071  | Cannot use directory 'directory-name'. Check installation path and verify if access rights are 'drwxrwxrwx'.  
Note: This error is non-trappable.  
The compiler needs to operate in the specified directory. Change the permission of this directory. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -6072  | Cannot create file in 'file-name'. Check installation path and verify if access rights are 'drwxrwxrwx'.  
**Note:** This error is non-trappable.  
The compiler needs to operate in the specified directory.  
Change the permission of this directory to 777 mode. |
| -6073  | Cannot change mode of a file in 'file-name'. Verify if access rights are 'drwxrwxrwx'.  
**Note:** This error is non-trappable.  
The compiler needs to operate in the specified directory.  
Change the permission of this directory to 777 mode. |
| -6074  | 'file-name' does not have 'rwxrwxrwx' rights or isn't a directory. Check access rights with 'ls -ld <installation-path>/lock' or execute 'rm -r <installation-path>/lock' if no users are connected.  
**Note:** This error is non-trappable.  
The compiler needs to operate in the specified directory.  
Change the permission of this directory. The $FGLDIR/lock directory contains only data needed at runtime by BDL applications. When the application is finished, you can remove this directory. If you delete this directory while BDL applications are running, the applications will be stopped immediately. |
| -6075  | Cannot read from directory 'directory-name'. Check installation path and verify if access rights are 'drwxrwxrwx'.  
**Note:** This error is non-trappable.  
The compiler needs to operate in the specified directory.  
Change the permission of this directory. |
| -6076  | Bad lock tree. Please check your environment.  
**Note:** This error is non-trappable.  
There is a problem accessing the $FGLDIR/lock directory.  
Check if the current user has sufficient permission to read and write to the $FGLDIR/lock directory. Check also if the FGLDIR environment variable is correctly set. |
| -6077  | Bad lock tree. Please check your environment.  
**Note:** This error is non-trappable.  
There is a problem accessing the $FGLDIR/lock directory.  
Check if the current user has sufficient permission to read and write to the $FGLDIR/lock directory. Check also if the FGLDIR environment variable is correctly set. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -6079  | Cannot get machine name or network IP address. Each graphical client must have an IP address when using a license server. FGLSERVER must hold the IP address or the host name of the client.  
**Note:** This error is non-trappable.  
You are using the remote license process and you have set the value of fgllic.server, in $FGLDIR/etc/fglprofile, to localhost or to the 127.0.0.1 address.  
You must use the real IP address of the computer even if it is the local computer. This is also true for the value used with the FGLSERVER environment variable. |
| -6080  | Cannot get IP address from 'host-name' host. Check the 'flm.server' resource.  
**Note:** This error is non-trappable.  
The system cannot find the IP address of the specified host.  
This is a configuration issue regarding your system. Check if the ping command gets a reply. Correct your system configuration and then try to execute your program. |
| -6081  | Cannot reach host 'host-name' with ping. Check license server entry in your resource file. Check your network configuration or increase 'flm.ping' value.  
**Note:** This error is non-trappable.  
The license server cannot ping the client computer, or it does not get the response in the time limit specified by the fgllic.ping entry in the $FGLDIR/etc/fglprofile file.  
Try to manually ping the specified computer. If this works, try to increase the value of the fgllic.ping entry in the FGLPROFILE file. If the ping does not respond, fix the system configuration problem and then try the program again. |
| -6082  | SYSERROR(error-num) description: Cannot set option TCP_NODELAY on socket. Check the system error message and retry.  
**Note:** This error is non-trappable.  
There is a problem with the socket of the Windows® computer.  
Check that the system is correctly configured and retry the program. |
| -6083  | Genero products are not allowed to run in containers.  
**Note:** This error is non-trappable.  
It is not possible to execute Genero products in virtual OS containers. |
| -6084  | The operation can not be performed. The license is assigned to a docker container.  
**Note:** This error is non-trappable.  
The license denies the current operation because it is assigned to a docker container. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -6085 | SYSERROR(error-num) description: Cannot connect to the license server on host 'host-name'. Check following things: - license server entry. - the license server machine. - the license server TCP port.  
**Note:** This error is non-trappable.  
The application cannot check the license validity. To do so, it tries to communicate with the Genero Business Development Language license service running on the computer where the product is installed.  
Check that the Genero Business Development Language License Server is running on the computer where the product is installed. |
| -6086 | SYSERROR(error-num) description: Cannot send data to the license server. Check the system error message and retry.  
**Note:** This error is non-trappable.  
There is a problem with the socket of the Windows® computer.  
Check that the system is correctly configured and retry the program. |
| -6087 | SYSERROR(error-num) description: Cannot receive data from license server. Check the system error message and retry.  
**Note:** This error is non-trappable.  
There is a problem with the socket of the Windows® computer.  
Check that the system is correctly configured and retry the program. |
| -6088 | You are not allowed to be connected for the following reason: description  
**Note:** This error is non-trappable.  
The program cannot connect to the license server because of the specified reason.  
Try to fix the problem described and rerun your application. |
| -6089 | Each graphical client must have an IP address when using a license server. FGLSERVER must hold the IP address or the host name of the client (localhost or 127.0.0.1 are not allowed).  
**Note:** This error is non-trappable.  
Use the real IP address or hostname of the client. |
| -6090 | SYSERROR(error-num) description: Cannot create a socket to start the license server. Check the system error message and retry.  
**Note:** This error is non-trappable.  
There is a problem with the socket of the Windows® computer.  
Check that the system is correctly configured and rerun the program. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -6091  | `SYSERROR(error-num)` description: Cannot bind socket for the license server. Check the system error message and retry.  
  **Note:** This error is non-trappable.  
  There is a problem with the socket of the Windows® computer.  
  Check that the system is correctly configured and rerun the program. |
| -6092  | `SYSERROR(error-num)` description: Cannot listen socket for the license server.  
  **Note:** This error is non-trappable.  
  There is a problem with the socket of the Windows® computer.  
  Check that the system is correctly configured and rerun the program. |
| -6093  | `SYSERROR(error-num)` description: Cannot create a socket to search an active client.  
  **Note:** This error is non-trappable.  
  There is a problem with the socket of the Windows® computer.  
  Check that the system is correctly configured and rerun the program. |
| -6094  | `SYSERROR(error-num)` description: This is a WSASStartup error.  
  Check the system error message and retry.  
  **Note:** This error is non-trappable.  
  There is a problem with the socket of the Windows® computer.  
  Check that the system is correctly configured and rerun the program. |
| -6095  | License problem: description  
  **Note:** This error is non-trappable.  
  License type incompatible. You are installing an earlier version, which was not designated for use with the current license server.  
  Reinstall and then contact your vendor. |
| -6096  | Connection refused by the license server.  
  **Note:** This error is non-trappable.  
  There is problem connecting the client computer to the Windows® license server.  
  There is a configuration problem with the license server computer. Check the configuration of the computers and of the products. |
| -6100  | Bad format of line sent by the license requester.  
  **Note:** This error is non-trappable.  
  The license request sent by the license controller is not understood by the license manager.  
  Upgrade your license software to the latest version available. If the issue is not solved, contact your support center. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -6101  | License number 'license-number' does not correspond to license key 'license-key'.  
  
  **Note:** This error is non-trappable.  
  Either the license number or the license key is invalid.  
  Check the license number and keys entered and try again. If that does not solve the issue, upgrade your license software to the latest version available. If the issue is not solved, contact your support center. |
| -6102  | Verify if resource 'flm.license.number' and 'flm.license.key' correspond to a valid license.  
  
  **Note:** This error is non-trappable.  
  Either the `flm.license.number` or `flm.license.key` entry in `fglprofile` is incorrectly filled. Ensure these `fglprofile` entries contain valid license numbers. |
| -6103  | License 'license-number' is no longer available from the license server.  
  
  **Note:** This error is non-trappable.  
  The license has been uninstalled from the license server. It may still appear as some sessions are active, but the license cannot be used to start a new session.  
  Reinstall the license, or contact your support center. |
| -6107  | User limit exceeded. Please retry later.  
  
  **Note:** This error is non-trappable.  
  The maximum number of clients that can be run has been reached (due to the license installed).  
  Retry later (when the number of current users has decreased) or install a new license that allows more users. |
| -6108  | Environment is incorrect.  
  
  **Note:** This error is non-trappable.  
  There is no local license, or the environment is not set correctly.  
  Check your environment and your FGLDIR environment variable. |
| -6109  | Cannot add session #session-number.  
  
  **Note:** This error is non-trappable.  
  You do not have the permissions to create the new session (the directory representing the new client).  
  Check the permissions of the dedicated directories. |
| -6110  | Cannot add program 'program-name' (pid=processid).  
  
  **Note:** This error is non-trappable.  
  You do not have the permissions to create the new application (the file representing the new application) for the current user.  
  Check the permissions of the dedicated directories. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -6112  | Compilation is not allowed: This product is licensed for runtime only.  
**Note:** This error is **non-trappable**.  
Buy and install a development license. |
| -6113  | Compilation is not allowed: Invalid license.  
**Note:** This error is **non-trappable**.  
Buy and install a development license. |
| -6114  | Cannot start program 'program-name' or result of process number is 0.  
**Note:** This error is **non-trappable**.  
When fgIWrt -u is executed to find the number of users allowed on this installation, the command "ps" may be launched (only on UNIX™).  
Check the permissions for ps. |
| -6116  | Wrong number of characters.  
**Note:** This error is **non-trappable**.  
The license number, license key, installation number, installation key or maintenance key provided is incomplete.  
Ensure that provided license numbers are correct and try again. |
| -6117  | The entry must be 12 characters long.  
**Note:** This error is **non-trappable**.  
The license number, license key, installation number, installation key or maintenance key provided is incomplete.  
Ensure that provided license numbers are correct and try again. |
| -6118  | Wrong checksum result for this entry.  
**Note:** This error is **non-trappable**.  
When entering license numbers, the checksum is verified if it is provided. This error occurs if the checksum computed does not match the provided checksum. Either the checksum or the license number is wrong.  
Ensure that checksum and license numbers are correct and try again. |
| -6122  | You must specify entry 'flm.server' in the resource file.  
**Note:** This error is **non-trappable**.  
The FGLPROFILE entry flm.server is missing. This entry must contain the host name or IP address of the host running the license manager.  
Add and configure the FGLPROFILE entry for flm.server. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -6123  | SYSERRORE(error Num) description: Cannot open socket. Check the system error message and retry.  
**Note:** This error is non-trappable.  
The license controller can not connect to the license manager.  
Check the error message and fix the issue. Ensure that FGLPROFIE entries flm.server and flm.service are correctly filled. Check your network configuration. |
| -6129  | License uninstalled.  
**Note:** This error is non-trappable.  
This is an information message. |
| -6130  | This license requires a full installation.  
**Note:** This error is non-trappable.  
The installed license has not be activated, but can not be used in temporary installation mode.  
Contact your vendor to obtain the activation key. |
| -6131  | This license number is no more valid. Please, contact your vendor.  
**Note:** This error is non-trappable.  
The license number is no longer accepted.  
Contact your vendor to obtain a new license number. |
| -6132  | Incompatible License Controller (fglWrt/greWrt) version. The minimum version required is min-version.  
**Note:** This error is non-trappable.  
Upgrade your license controller version to the specified version or higher. |
| -6133  | This product requires a BDL license. The license number should start with the letter F.  
**Note:** This error is non-trappable.  
A BDL license is required for this product.  
Call you support center to get a BDL license. |
| -6134  | This product requires a Genero license. The license number should start with the letter T.  
**Note:** This error is non-trappable.  
A Genero license is required for this product.  
Call you support center to get a Genero license. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -6135  | Invalid license key.  
**Note:** This error is non-trappable.  
The license key does not correspond to the license number.  
Call you support center to check the license key. |
| -6136  | The date-limited license has expired.  
**Note:** This error is non-trappable.  
The time limited license has expired, the product is blocked.  
Call you support center to get a new license. |
| -6137  | This product requires a GRW license.  
**Note:** This error is non-trappable.  
A GRW license is required for this product.  
Call you support center to get a GRW license. |
| -6138  | GRW licenses are not accepted by this product.  
**Note:** This error is non-trappable.  
This product does not accept GRW licenses.  
Call you support center to check if the license corresponds to the product. |
| -6140  | Version version-number  
**Note:** This error is non-trappable.  
This is an information message. |
| -6142  | Try and buy demonstration time expired. Please, restart your application.  
**Note:** This error is non-trappable.  
Applications started with a Try and Buy version will stop after few minutes of execution.  
Restart your application. |
| -6143  | This license requires a valid maintenance key. Check your environment (run 'fglwrt/gewrt -a info')  
**Note:** This error is non-trappable.  
Genero 2.20 and higher require a valid maintenance key.  
Update your maintenance key. |
| -6144  | The DVM build date is greater than the maintenance key expiration date. Contact your nearest FourJ's sales representative to update the maintenance key.  
**Note:** This error is non-trappable.  
Update your maintenance key or downgrade your Genero installation to an older version. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -6146  | This product requires a Genero Time-Limited Evaluation license.  
**Note:** This error is non-trappable.  
You have installed a trial version of the Genero product, but the installed license is not a trial license.  
Install a trial license for this product. Contact your support to get a trial license. |
| -6147  | This product requires a GRE Time-Limited Evaluation license.  
**Note:** This error is non-trappable.  
You have installed a trial version of the GRE product, but the installed license is not a trial license.  
Install a trial license for this product. Contact your support to get a trial license. |
| -6148  | Installation path is not known.  
**Note:** This error is non-trappable.  
You are handling licenses but the FGLDIR environment variable is not set.  
Set the FGLDIR environment variable and retry. |
| -6149  | Problem while installing license 'license-number'.  
**Note:** This error is non-trappable.  
A problem occurred while licensing.  
Note the system-specific error number and contact your Technical Support. |
| -6150  | Temporary license not found for this version.  
**Note:** This error is non-trappable.  
While adding a definitive license key, the temporary license has not been found.  
Re-install the license. |
| -6151  | Wrong installation key.  
**Note:** This error is non-trappable.  
While adding a definitive license key, the installation key was not valid.  
Re-install the license. |
| -6152  | Problem during license installation.  
**Note:** This error is non-trappable.  
A problem occurred while installing the license. Unable to write information to the disk (either own files or system files).  
Check the FGLDIR environment variable and the rights of the license files (must be able to change them). |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -6153  | License installation failed.  
        **Note:** This error is non-trappable.  
        Unable to write license information to files.  
        Check the system error message if provided, check the file permissions for the current user. |
| -6154  | License installation successful.  
        **Note:** This error is non-trappable.  
        This is an information message. |
| -6156  | Too many temporary licenses. You must reinstall a license.  
        **Note:** This error is non-trappable.  
        You installed a temporary license too many times.  
        Contact technical support to get a valid license. |
| -6158  | Cannot store temporary information.  
        **Note:** This error is non-trappable.  
        A problem occurred while installing the license. Unable to write information to the disk (either own files or system files).  
        Check the FGLDIR environment variable and the rights of the license files (you must be able to change them). |
| -6159  | This kind of license is not permitted.  
        **Note:** This error is non-trappable.  
        The license numbers can not be installed.  
        Contact your support center. |
| -6160  | You do not have the permissions to be connected.  
        **Note:** This error is non-trappable.  
        The host running the license controller (where the DVM is running) is not allowed to connect to this license manager. There is likely a configuration issue.  
        Check your license manager configuration. |
| -6161  | You do not have the permissions to compile.  
        **Note:** This error is non-trappable.  
        The compilation request is rejected by the license manager.  
        Contact your support center. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-6162</td>
<td>Cannot reach the license server. Please check if 'flm.server' is correctly initialized. ('flmprg -a info up' command should answer 'ok'). The license server is running but no autocheck will be done. <strong>Note:</strong> This error is non-trappable. While this error is no longer used, it can be raised by older versions of Genero. The license controller can not connect to the license manager. Ensure that FGLPROFILE entries flm.server and flm.service are filled correctly. Ensure that the license manager is running on the specified host and port.</td>
</tr>
<tr>
<td>-6168</td>
<td>Cannot get information from directory 'directory-name'. Failed to read directory information. <strong>Note:</strong> This error is non-trappable. Ensure that the user installing a license is the user that installed the product. Ensure that the user installing a license has read/write permissions on the 'etc' directory of the product.</td>
</tr>
<tr>
<td>-6169</td>
<td>SYSEXERROR(error-num) description: Cannot set option O_NONBLOCK on socket. Check the system error message and retry. <strong>Note:</strong> This error is non-trappable. Failed to configure the socket in non-blocking mode. Check the system error message. Contact your support center.</td>
</tr>
<tr>
<td>-6170</td>
<td>Old request format to license server detected. You must install a license program version 2.99 or higher. <strong>Note:</strong> This error is non-trappable. The license controller version is too old for the current license manager version. Requests sent by the license controller are no longer supported by the current license manager. Upgrade the license controller to the latest version available.</td>
</tr>
<tr>
<td>-6171</td>
<td>A license has been installed temporarily. Only the installation key is required. You must run 'fglWrt -k &lt;installation-key&gt;' to install it. <strong>Note:</strong> This error is non-trappable. The installed license is temporarily installed, yet it is missing the installation key. Obtain your installation key and install it.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| -6172 | Bad parameter: 'parameter' hasn't the right format.  
**Note:** This error is non-trappable.  
Two issues can raise this error.  
1. The license manager can raise this error if it receives a request from the license controller with unknown commands.  
   Upgrade the license software version to the latest available. Contact your support center.  
2. The license manager etc/lmprofile configuration is invalid.  
   Check your lmprofile entries flm.license.together and flm.license.allow. |
| -6173 | Invalid license number or invalid license key.  
**Note:** This error is non-trappable.  
During the license installation, the license number / license key couple does not match.  
Ensure that the license numbers are correct. Upgrade the license software to the latest version available and retry. |
| -6174 | This option is only available for a local license. And resource 'flm.server' was found in your configuration.  
**Note:** This error is non-trappable.  
A license server is configured and the user tries to install or uninstall a license using fglWrt.  
Install the license on the license manager (flmprg) Or remove the license manager configuration from FGLPROFILE file and install the license locally. |
| -6175 | License number 'license-number' is invalid.  
**Note:** This error is non-trappable.  
The license number is not valid.  
Ensure that the license number provided is correct. Upgrade the license software to the latest version available. |
| -6176 | In license server, following problem occurs with license number 'license-number': problem-description  
**Note:** This error is non-trappable.  
This is a generic error containing the text of another error.  
Check the error. Contact your support center. |
| -6177 | Following problem occurs with license number 'license-number': description  
**Note:** This error is non-trappable.  
This is a generic error containing the text of another error.  
Check the error. Contact your support center. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -6178  | Your machine is not allowed to be connected on any of your authorized licenses.  
Note: This error is non-trappable.  
The CPU license rejects the connection of a new host. All CPU licenses are consumed.  
Contact your support center. |
| -6179  | License validity time is reached. The users control is reactivated.  
Note: This error is non-trappable.  
The CPU license is time limit is reached. The CPU license is degraded and user control is reactivated.  
Contact your support center. |
| -6180  | CPU limit exceeded. The users control is reactivated.  
Note: This error is non-trappable.  
The CPU license has less CPUs available than the number of CPUs on the connected host, resulting in one or more hosts running in degraded mode. When running in degraded mode, the number of users allowed is the number of available CPUs in the license * 100 users. User control is activated for license request coming from that host.  
For example, if you have a 5 CPU license and are using the license manager.  
• Host A with 2 CPUs connects. It consumes 2 CPUs of the 5 CPU license. 3 CPUs remain free on the license.  
• Host B with 2 CPUs connects. It consumes 2 CPUs of the 5 CPU license. 1 CPU remains free on the license.  
• Host C with 2 CPU connects. It expects to be able to consume 2 CPUs, but only 1 CPU remains on the license. User control for Host C is enabled, with 1 CPU (the remaining free CPU) * 100 users allowed.  
• Host D connects with N CPU, however there are no CPUs free on the license. Host D is rejected.  
In summary, Host A and B can have unlimited sessions, Host C is limited to 100 sessions as the license is degraded by CPU, and any other host is rejected.  
In the case of a single host, user control applies to local licenses as well. If Host A has 6 CPUs, yet has a 5 CPU license, it consumes all of the CPUs for the license and runs in degraded mode. That means 5 * 100 users are allowed.  
Contact your support center. |
| -6181  | Cannot get license extension information. Check your environment, the license (run 'fglWrt -a info') and the fglWrt version ('fglWrt -V' should give version-number or higher).  
Note: This error is non-trappable.  
License information is invalid. This error is not yet used.  
Contact your support center. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -6182  | Your license has 'restriction-name' restriction. You are not allowed to run another mode.  
**Note:** This error is non-trappable.  
The license has restrictions, and the requested use of the license is not compatible with these restrictions. For example, the license may have a text-only restriction, where GUI front-ends are not allowed.  
Contact your support center to obtain a license matching your needs. |
| -6183  | Local license controller (fglWrt) may not be compatible with this runner. Check its version ('fglWrt -V' should give version-number or higher).  
**Note:** This error is non-trappable.  
The license controller is incompatible.  
Update the license controller to the latest version available. |
| -6184  | You are not authorized to run this version of runner.  
**Note:** This error is non-trappable.  
Older licenses do not use the maintenance key. The DVM version that can be used is limited. The DVM version is higher that the allowed DVM version.  
Contact your support center. Re-licensing is required. |
| -6185  | Protection file is not compatible with this version of the runner. You must reinstall your license.  
**Note:** This error is non-trappable.  
This error is no longer used by the current licensing software; however, it may occur with older versions.  
Contact your support center. |
| -6186  | Demo version initialization.  
**Note:** This error is non-trappable.  
This is an information message. |
| -6188  | Your evaluation license period has expired. Contact your support center.  
**Note:** This error is non-trappable.  
The software you are using has been installed with a demo license that has expired.  
Contact your software vendor to extend the evaluation period or purchase a permanent license. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -6196  | **You are not authorized to delete sessions from the license server 'server-name'.**  
|        | **Note:** This error is non-trappable. The command `fglWrt -i` can only be used with local licenses. If a license server is configured, this error is raised. Use the command `flmprg` instead. |
| -6197  | 'extension-name' extension is not allowed with this license type.  
|        | **Note:** This error is non-trappable. Generic error indicating that an extension check is rejected. For example, if you are using a non-Informix® database server, this error will raise if the ODI extension is not set in the license. |
| -6198  | Product identifier does not correspond to the license number.  
|        | **Note:** This error is non-trappable. This error indicates that a wrong license is installed in the product, such as when you attempt to use a Genero Report Writer (GRW) license when installing the Genero Business Development Language (BDL). This error is generally not be raised, as `fglWrt` will reject the installation of a Genero Report Engine (GRE) license when installing Genero BDL, and conversely `greWrt` will not allow the installation of a Genero BDL license. That being said, the installation checks to ensure the license is valid for the product, and raises this error if is not. Ensure the proper license is used with the proper package. |
| -6199  | Cannot create directory 'directory-name'. Check installation path and verify your access rights.  
|        | **Note:** This error is non-trappable. The specified directory can not be created or modified. |
| -6200  | Module 'module-name': The function function-signature-1 will be called as function-signature-2 .  
|        | **Note:** This error is non-trappable. An incorrect number of parameters are used to call a FUNCTION. Check your source code and recompile your application. |
| -6201  | Module 'module-name': Bad version: Recompile your sources.  
<p>|        | You have compiled your program with an old version. The newly compiled version of your program is not supported. Compile all source files and form files again. |</p>
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -6202  | filename 'file-name': Bad magic: Code cannot run with this p code machine.  
You have compiled your program with an old version. The new compiled version of your program is not supported. You might also have a file with the same name as the .42r. You used the fglrun 42r-Name without specifying the extension.  
To resolve this problem, call fglrun with the .42r extension or recompile your application. |
| -6203  | Module 'module-name-1': The function 'function-name' has already been defined in module 'module-name-2'.  
**Note:** This error is non-trappable.  
The specified function is defined for the second time in the application. The second occurrence of the function is in the specified module.  
Eliminate one of the two function definitions from your source code. |
| -6204  | Module 'module-name': Unknown op-code.  
**Note:** This error is non-trappable.  
An unknown instruction was found in the compiled BDL application.  
Check that the version of the Genero Business Development Language package executing the compiled application is the same as the one that compiled the application. It is also possible that the compiled module has been corrupted. If so, you will need to recompile your application. |
| -6205  | INTERNAL ERROR: Alignment.  
**Note:** This error is non-trappable.  
This error is internal, and does not normally occur.  
Contact your Technical Support. |
| -6206  | The 42m module 'module-name' could not be loaded, check FGLLDPATH environment variable.  
**Note:** This error is non-trappable.  
The .42m module is not in the current directory or in one of the directories specified by the FGLLDPATH environment variable.  
Set the environment variable FGLLDPATH with the path to the .42m modules to be loaded. |
| -6207  | The dynamic loaded module 'module-name' does not contain the function 'function-name'.  
**Note:** This error is non-trappable.  
A BDL module has been changed and recompiled, but the different modules of the application have not been linked afterward.  
Link the new modules together before you execute your application. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -6208  | Module 'module-name' already loaded.  
     **Note:** This error is non-trappable.  
A module was loaded twice at runtime. This can occur because one module has been concatenated with another.  
Recompile and re-link your BDL modules. |
| -6210  | INTERNAL ERROR: exception 2 raised before invoking the exception handler for exception 1.  
A module was loaded twice at runtime. This can occur because one module has been concatenated with another.  
Check for function names, recompile and re-link your BDL modules. |
| -6211  | Link has failed.  
A problem occurred while linking the BDL program.  
Check for function names, recompile and re-link your BDL modules. |
| -6212  | Function function-name : local variables size is too large - Allocation failed.  
A local function variable is too large and runtime is unable to allocate memory.  
Review the variable data types in the function. |
| -6213  | Module module-name : Module's variable size is too large - Allocation failed.  
A module variable is too large and runtime is unable to allocate memory.  
Review the variable data types in the module. |
| -6214  | Global variable variable-name size is too large - Allocation failed.  
A global variable is too large and runtime is unable to allocate memory.  
Review the variable data types in the globals. |
| -6215  | Memory allocation failed. Ending program.  
Runtime is unable to allocate memory.  
Check for system resources and verify if the OS user is allowed to allocate as much memory as the program needs (check for `ulimits` on UNIX™ systems). |
| -6216  | The global 'constant-name' has been redefined with a different constant-value.  
**Note:** This error is non-trappable.  
A global constant has been defined twice with a different value.  
A global constant may have only one value. Review your code. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -6217  | The global 'variable-name' has been defined as a constant and a variable.  
**Note:** This error is **non-trappable.**  
The same symbol was used to define a constant and a variable.  
Use a different name for the constant and the variable. Review your code. |
| -6218  | No runtime. You must call fgl_start() before calling fgl_call().  
This error occurs when a C extension has redefined the main() routine, but then does not call fgl_start() to initialize the BDL runtime environment.  
Check the C extension and call fgl_start() before any other operation. |
| -6219  | WHENEVER ERROR CALL: The error-handler recursively calls itself.  
The function specified with the WHENEVER ERROR CALL instruction raises an error that would call itself recursively.  
Review the function called by the WHENEVER ERROR CALL and make sure it does not produce a runtime error. |
| -6220  | Could not load C extension library 'library-name'. Reason: description  
**Note:** This error is **non-trappable.**  
Runtime system is unable to find the shared library for the reason given.  
Check if the C extension library exists in one of the directories defined by FGLLDPATH. If the C extension module depends on other shared libraries, make sure that these libraries can be found by the library loader of the operating system (check the LD_LIBRARY_PATH environment variable on UNIX™ or the PATH environment variable on Windows®). |
| -6221  | C extension initialization failed with status number.  
**Note:** This error is **non-trappable.**  
C extension failed to initialize and returned the status shown in the error message.  
Check the C extension source or documentation. |
| -6222  | class-name class not found.  
**Note:** This error is **non-trappable.**  
The program was compiled with the built-in class class-name but at execution time the class is not found.  
Check you installation, it is possible that you are executing program that was compiled with a younger version as the version used in the execution context, which certainly is missing that class in the runtime library. |
| -6223  | No such symbol: symbol-name.  
**Note:** This error is **non-trappable.**  
The runtime system loads a module dynamically (on demand) and searches for the symbol in a given module. But the symbol is not found, for example, because of an invalid FGLLDPATH, or because the installed module no longer contains the symbol (after a recompilation). |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-6300</td>
<td>Can not connect to GUI: description</td>
</tr>
<tr>
<td></td>
<td>You have run a GUI application but the environment variable FGLSERVER is not set correctly, or the Genero client (graphical front-end) is not running.</td>
</tr>
<tr>
<td></td>
<td>It is recommended that the FGLSERVER environment variable is set to the hostname and port of the graphical front-end used by the runtime system to display the application windows. Check that the network connection is still available, make sure no firewall denies access to the workstation, and make sure the front-end is still running.</td>
</tr>
<tr>
<td>-6301</td>
<td>Can not write to GUI: description</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> This error is non-trappable.</td>
</tr>
<tr>
<td></td>
<td>You are running a GUI application but for an unknown reason the front-end no longer responds and the runtime system is unable to write to the GUI socket.</td>
</tr>
<tr>
<td></td>
<td>Check that the network connection is still available, make sure no firewall denies access to the workstation, and see whether the front-end is still running.</td>
</tr>
<tr>
<td>-6302</td>
<td>Can not read from GUI: description</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> This error is non-trappable.</td>
</tr>
<tr>
<td></td>
<td>You are running a GUI application but for an unknown reason the front-end no longer responds and the runtime system is unable to read from the GUI socket.</td>
</tr>
<tr>
<td></td>
<td>Check that the network connection is still available, make sure no firewall denies access to the workstation, and see whether the front-end is still running.</td>
</tr>
<tr>
<td>-6303</td>
<td>Invalid user interface protocol.</td>
</tr>
<tr>
<td></td>
<td>You are trying to execute a program with a runtime system that uses a different AUI protocol version as the front-end.</td>
</tr>
<tr>
<td></td>
<td>Install either a new front-end or a new runtime environment that matches (2.0x with 2.0x, 1.3x with 1.3x).</td>
</tr>
<tr>
<td>-6304</td>
<td>Invalid abstract user interface definition.</td>
</tr>
<tr>
<td></td>
<td>You are trying to execute a program with a runtime system that uses a different AUI protocol version as the front-end.</td>
</tr>
<tr>
<td></td>
<td>Install either a new front-end or a new runtime environment that matches (2.0x with 2.0x, 1.3x with 1.3x).</td>
</tr>
<tr>
<td>-6305</td>
<td>Can not open char table file. Check your fglprofile.</td>
</tr>
<tr>
<td></td>
<td>This error occurs if the conversion file defined by the gui.chartable entry, in the $FGLDIR/etc/fglprofile file, is not readable by the current user.</td>
</tr>
<tr>
<td></td>
<td>Check if the gui.chartable entry is correctly set and if the specified file is readable by the current user.</td>
</tr>
<tr>
<td>-6306</td>
<td>Can not open server file. Check installation.</td>
</tr>
<tr>
<td></td>
<td>A file on the server side cannot be sent to the graphical interface.</td>
</tr>
<tr>
<td></td>
<td>Check the permissions of the file located in the $FGLDIR/etc directory. These files must have at least read permission for the current user.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
|-6307   | GUI server autostart: can not identify workstation.  
GUI Server autostart configuration is wrong. Either DISPLAY, FGLSERVER or FGLPROFILE settings are invalid.  
Set the required environment variables and check for FGLPROFILE autostart entries. |
|-6308   | GUI server autostart: unknown workstation: check gui.server.autostart entries.  
The computer described by the X11 DISPLAY environment variable is neither the local host, nor is it listed in the FGLPROFILE entries.  
Check if the X11 DISPLAY name is correctly set, or review the FGLPROFILE entries. |
|-6309   | Not connected. Cannot write to GUI.  
For unknown reasons there was an attempt to write on the GUI socket before the connection was initiated.  
Check the program for invalid GUI operations. |
|-6310   | Not connected. Cannot read from GUI.  
For unknown reasons there was an attempt to read on the GUI socket before the connection was initiated.  
Check the program for invalid GUI operations. |
|-6311   | No current window.  
The program tries to issue a MENU instruction with no current window open.  
Review the program logic and make sure a window is open before MENU. |
|-6312   | The type of the user interface (FGLGUI) is invalid.  
**Note:** This error is **non-trappable**.  
While initiating the user interface, the runtime system did not recognize the GUI type and stopped.  
Make sure the FGLGUI environment variable has a correct value. |
|-6313   | The UserInterface has been destroyed.  
**Note:** This error is **non-trappable**.  
The error occurs when the front-end sends a DestroyEvent event, indicating some inconsistency with the starting program. This can happen, for example, when multiple StartMenus are used, or when you try to run an MDI child without a parent container, or when two MDI containers are started with the same name, etc.  
Check for inconsistency and fix it. |
|-6314   | Wrong connection string. Check client version.  
While starting the program, the runtime received a wrong or incorrectly constructed answer from the front-end.  
Make sure you are using a front-end that is compatible with the runtime system. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -6315  | The form is too complex for the console-ui.  
The program tries to display a form with a complex layout that cannot be displayed in text mode.  
Review the form file and use a simple grid with a SCREEN section instead of LAYOUT. |
| -6316  | Error error-num returned from client: description  
**Note:** This error is non-trappable.  
Front-end returned the specified error during GUI connection initialization.  
Check the front-end documentation for more details. |
| -6317  | Invalid or unsupported client protocol feature.  
The GUI protocol feature you are trying to use is not supported by the front-end. For example, you are trying to use protocol compression but the runtime is not able to compress data.  
Make sure that the front-end component is compatible with the runtime system (versions must be close). Check the runtime system version for supported protocol features. If compression is enabled, check that the zlib library is installed on your system. |
| -6318  | Choosing the DIALOG implementation by setting the environment variable FGL_USENDIALOG=0 has been desupported since version 2.20.03.  
You try to use the old dialog implementation by setting FGL_USENDIALOG to zero.  
The old dialog implementation has been removed, you must unset the FGL_USENDIALOG environment variable. |
| -6319  | Internal error in the database library. Set FGLSQLDEBUG to get more details.  
An unexpected internal error occurred in the database driver.  
Set the FGLSQLDEBUG environment variable to level 1, 2, 3 or 4 to get detailed debug information. |
| -6320  | Can't open file 'file-name'.  
The runtime system tried to open a resource file in FGLDIR but access is denied or file no longer exists.  
Check for file permissions and existence in FGLDIR. |
| -6321  | No such interface capability: 'feature-name'.  
The runtime system tried to use a front-end protocol capability, but is not able to use it.  
Check if the front-end is compatible with the runtime system. |
Some resource files of FGLDIR have been identified as too old for the current runtime system.  
Re-install the runtime system environment. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -6323  | Can't load factory profile 'file-name'.  
The default fglprofile file located in FGLDIR/etc is missing or is unreadable.  
Check the permission of the file. If the file is missing, reinstall the software. |
| -6324  | Can't load customer profile 'file-name'.  
The configuration file defined by the FGLPROFILE environment variable is missing or unreadable.  
Check if the FGLPROFILE environment variable is correctly set and if the file is readable by the current user. |
| -6325  | Can't load application resources 'file-name'.  
The directory specified by the fglrun.default entry in FGLDIR/etc/fglprofile is missing or not readable for the current user.  
Check if the entry fglrun.default is correctly set in FGLDIR/etc/fglprofile and if the directory specified is readable by the current user. |
| -6327  | Internal error in the run time library file library-name.  
**Note:** This error is **non-trappable**.  
Something unpredictable has occurred, generating an error.  
Contact your Technical Support. |
| -6328  | Bad format of resource 'entry-name' value 'entry-value': you must use the syntax: entry-name='VARNAME=envvar-value'.  
The FGLPROFILE file contains an invalid environment variable definition format.  
Check the content of the profile file. |
| -6329  | All TABLE columns must be defined with the same height.  
The form layout defines a TABLE with field tags using different heights.  
Review all cells of the table to use the same height in all columns. |
| -6330  | Syntax error in profile 'file-name', line number lineno, near 'token'.  
**Note:** This error is **non-trappable**.  
The FGLPROFILE file shown in the error message contains a syntax error.  
Check the content of the profile file. |
| -6331  | Front-end module could not be loaded.  
A front call failed because the module does not exist.  
The front end is probably not supporting this module. |
| -6332  | Front-end function could not be found.  
A front call failed because the function does not exist.  
The front end is probably not supporting this function. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -6333 | Front-end function call failed. Reason: description  
For some reason, the front call failed and produced this runtime error.  
Check the content of the error message to get more information. |
| -6334 | Front-end function call stack problem.  
A front call failed because the number of parameter or returning values does not match.  
Make sure the number of parameters and return values are correct. |
| -6340 | Can't open file: description  
The channel object failed to open the file specified.  
Make sure the filename is correct and user has permissions to read/write to the file. |
| -6341 | Unsupported mode for 'open file'.  
You try to open a channel with an unsupported mode.  
See channel documentation for supported modes. |
| -6342 | Can't open pipe.  
The channel object failed to open a pipe to execute the command.  
Make sure the command you try to execute is valid. |
| -6343 | Unsupported mode for 'open pipe'.  
You try to open a channel with an unsupported mode.  
See channel documentation for supported modes. |
| -6344 | Can't write to unopened file, pipe or socket.  
You try to write to a channel object which is not open.  
First open the channel, then write. |
| -6345 | Channel write error: description  
An unexpected error occurred while writing to the channel. See the description for more details. |
| -6346 | Cannot read from unopened file, pipe or socket.  
You try to read from a channel object which is not open.  
First open the channel, then read. |
| -6360 | This runner cannot execute any SQL.  
The runtime system is not ready for database connections.  
Check the configuration of BDL. |
| -6361 | Dynamic SQL: type unknown: type-name .  
The database driver does not support this SQL data type.  
You cannot use this SQL data type, review the code. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -6364  | Cannot connect to sql back end.  
The runtime system is unable to initialize the database driver to establish a database connection.  
Make sure the database driver exists. |
| -6365  | Database driver not connected yet.  
There is an attempt to execute an SQL statement, but no database connect is established.  
First connect, then execute SQL statements. |
| -6366  | Could not load database driver driver-name.  
**Note:** This error is non-trappable.  
The runtime system failed to load the specified database driver. The database driver shared object (.so or .DLL) or a dependent library could not be found.  
Make sure that the name of the specified driver is spelled correctly. If the driver name is correct, there is probably an environment problem. Make sure the database client software is installed. Check the UNIX™ LD_LIBRARY_PATH environment variable or the PATH variable on Windows®. These must point to the database client libraries. |
| -6367  | Incompatible database driver interface.  
**Note:** This error is non-trappable.  
The database driver interface does not match the interface expected by the runtime system. This can occur if you copy an old database driver into a younger FGLDIR installation.  
Call the support to get a valid database driver. |
| -6368  | SQL driver initialization function failed.  
**Note:** This error is non-trappable.  
The runtime system failed to initialize the database driver, program must stop because no database connection can be established.  
There is probably an environment problem (for example, INFORMIXDIR or ORACLE_HOME is not set). Check your environment and try to connect with a database vendor tool (dbaccess, sqlplus) to identify the problem. |
| -6369  | Invalid database connection mode.  
You try to mix DATABASE and CONNECT statements, but this is not allowed.  
Use either DATABASE or CONNECT. |
| -6370  | Unsupported SQL feature.  
This SQL command or statement is not supported with the current database driver.  
Review the code and use a standard SQL feature instead. |
| -6371  | SQL statement error number error-num (native-error).  
An SQL error has occurred having the specified error number.  
You can query SQLERRMESSAGE or the SQLCA record to get a description of the error. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-6372</td>
<td>General SQL error, check SQLCA.SQLERRD[2]. A general SQL error has occurred. You can query SQLERRMESSAGE or the SQLCA record to get a description of the error. The native SQL error code is in SQLCA.SQLERRD[2].</td>
</tr>
<tr>
<td>-6373</td>
<td>Invalid database connection string. The database connection string that you have used is not valid. Verify the format of the connection string.</td>
</tr>
<tr>
<td>-6374</td>
<td>Wrong database driver context. You try to EXECUTE, OPEN, FETCH, PUT, FLUSH, CLOSE or FREE a cursor that was declared or prepared in a different connect and driver. Issue a SET CONNECTION before the statement to select the same connection and driver as when the cursor was created.</td>
</tr>
<tr>
<td>-6375</td>
<td>LOAD cannot get describe information for table columns. The LOAD instructions needs column description to allocate the automatic fetch buffers, but the database driver is not able to describe the table columns used in the INSERT statement. If the underlying database client API does not provide result set column description, the LOAD statement cannot be supported.</td>
</tr>
<tr>
<td>-6601</td>
<td>Can not open Database dictionary 'directory-name'. Run database schema extraction tool. The schema file does not exist or cannot be found. If the schema file exists, verify that the filename is spelled correctly, and that the file is in the current directory or the FGDLDBPATH environment variable is set to the correct path. If the file does not exist, run the database schema extraction tool to create a schema file.</td>
</tr>
<tr>
<td>-6602</td>
<td>Can not open globals file 'file-name'. The globals file does not exist or cannot be found. Verify that the globals file exists. Check the spelling of the filename, and verify that the path is set correctly.</td>
</tr>
<tr>
<td>-6603</td>
<td>The file 'file-name' cannot be created for writing. The compiler failed to create the file shown in the error message for writing. Check for user permissions to make sure that the .42m file can be created.</td>
</tr>
<tr>
<td>-6604</td>
<td>The function 'function-name' can only be used within an INPUT [ARRAY], DISPLAY ARRAY or CONSTRUCT statement. The language provides built-in functions that can only be used within specific interactive statements. Review your code and make the necessary corrections. Check that the function is within the interactive statement and that appropriate END statements (END INPUT/ARRAY/DISPLAY ARRAY/CONSTRUCT) have been used.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| -6605  | The module 'module-name' does not contain function 'function-name'.  
The module shown in the error message does not hold the function name as expected.  
The specified function needs to be defined in this module. |
| -6606  | No member function 'function-name' for class 'class-name' defined.  
**Note:** This error is non-trappable.  
The function name is misspelled or is not a method of the class for which it is called.  
Review your code and the documentation for the method you are attempting to use. If the function is an object method, make sure the referenced object in your code is of the correct class. |
| -6608  | Resource error: entry-name :parameter expected  
This is a generic error message for resource file problems. |
| -6609  | A grammatical error has been found at 'seen-token' expecting: expected-token.  
A general syntax error message that indicates the location of the problem code and what code was expected.  
Review your code, particularly for missing END statements such as END FUNCTION or END INPUT, etc., and make the necessary corrections. |
| -6610  | The function 'function-name' has already been called with a different number of parameters.  
Earlier in the program, there is a call to this same function or event with a different number of parameters in the parameter list.  
Check the correct number of parameters for the specified function. Then examine all calls to it, and make sure that they are written correctly. |
| -6611  | Function 'function-name': unexpected number of returned values.  
The function shown returned a different number of values as expected.  
Check the body of the function for RETURN instructions. |
| -6612  | Redeclaration of function 'function-name'.  
The function shown was defined multiple times.  
Change the name of conflicting functions. |
| -6613  | The library function 'function-name' is not declared.  
The function shown was not declared.  
Change the name of the function. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -6614    | The function 'function-name' may return a different number of values.  
The function shown contains multiple RETURN instructions which may return different number of values.  
Review the RETURN instructions to return the same number of values. |
| -6615    | The symbol-kind 'symbol-name' is unused.  
This is a warning indicating that the shown symbol is defined but never used. In this message,  
"symbol-kind" can be "symbol", "parameter", etc.  
Useless definition can be removed. |
| -6616    | The symbol 'symbol-name' does not represent a defined CONSTANT.  
The shown symbol is used as a CONSTANT, but it is not a constant.  
Review your code and check for this name. |
| -6617    | The symbol 'symbol-name' is a VARIABLE.  
The symbol shown is a VARIABLE which cannot be used in the current context.  
Review your code and check for this name. |
| -6618    | The symbol 'symbol-name' is a CONSTANT.  
The symbol shown is a CONSTANT which cannot be used in the current context.  
Review your code and check for this name. |
| -6619    | The symbol 'symbol-name' is not an INTEGER CONSTANT.  
The symbol shown is used as if it was an INTEGER constant, but it is not.  
Review your code and check for this name. |
| -6620    | The symbol 'symbol-name' is not a REPORT.  
The symbol shown is used as a REPORT, but it is not defined as a REPORT.  
Review your code and check for this name. |
| -6621    | The symbol 'symbol-name' is not a FUNCTION.  
The symbol shown is used as a FUNCTION, but it is not defined as FUNCTION.  
Review your code and check for this name. |
| -6622    | The symbol 'symbol-name' does not represent a valid variable type.  
The symbol shown is not known as a valid type to define a program variable.  
Review your code and check for the type name. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-6623</td>
<td>The method 'method-name' cannot be called without an object. The specified method is an object method of its class. Review your code. Ensure that the required object of the class has been instantiated and still exists, and that the method is called specifying the object variable as the prefix, with the period character as a separator.</td>
</tr>
<tr>
<td>-6624</td>
<td>The method 'method-name' cannot be called with an object. The specified method is a class method and cannot be called using an object reference. No object has to be created. Review your code. Ensure that the method is called using the class name as the prefix, with the period character as a separator.</td>
</tr>
<tr>
<td>-6625</td>
<td>The statement is not Informix compatible. The SQL statement is not Informix® compatible. Change the SQL statement by using Informix® SQL syntax.</td>
</tr>
<tr>
<td>-6627</td>
<td>The symbol 'symbol-name' is not a VARIABLE. The symbol shown is use as a variable, but is not defined as a variable. Review your code and check for this name.</td>
</tr>
<tr>
<td>-6628</td>
<td>The GLOBALS file does not contain a GLOBALS section. The filename specified in a GLOBALS statement references a file that does not contain a GLOBALS section. Review your code to make sure that the file specified by the filename is a valid GLOBALS file, containing the required GLOBALS section.</td>
</tr>
<tr>
<td>-6629</td>
<td>The type 'type-name' is too complex to be used within a C-extension. The type of the global variable is too complex to be used in a C extension. This error can occur when the -G option of fglcomp, to generate the C sources to share global variables with C extensions, when a global variable is defined with complex data types without a C equivalent. Review the definition of the global variables and use simple types instead, corresponding to a C data type. The BYTE, TEXT and STRING types are complex types.</td>
</tr>
<tr>
<td>-6630</td>
<td>Memory overflow occurred during p-code generation. Simplify the module. A memory overflow occurred during compilation to p-code because the .4gl source module is too large. This problem can occur with very large source files. You must split the module into multiple sources.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>-6631</td>
<td>Incompatible types, found: source-type, required: target-type. A LET or RETURNING tries to assign a value or an object reference to a variable defined with a data type or class that is not compatible to the value type. This occurs usually when using Java classes because Java is a strongly typed language. For example, assigning a Java string to a Java StringBuffer raises this error. Define the target variable with a type corresponding to the assigned value.</td>
</tr>
<tr>
<td>-6632</td>
<td>Cannot find symbol symbol-name, location: category type-name. The symbol used does not exist. This occurs typically when referencing a Java class with an invalid name in the class path, or when referencing a class member that does not exist. Check the symbol names used in the instruction.</td>
</tr>
<tr>
<td>-6633</td>
<td>Primitive-type cannot be dereferenced. An expression references a method or a field with a primitive Java type, but primitive types are not classes and therefore do not have methods or fields. For example, you try to call a method with a symbol defined as integer or short in Java: DISPLAY java.lang.Short.MAX_VALUE.foo Review the code using the symbol defined with a primitive type.</td>
</tr>
<tr>
<td>-6634</td>
<td>Incompatible or corrupted database dictionary 'database-name'. The .sch database schema 'database-name' contains incompatible type definitions or is corrupted. Re-generate the .sch file with the fgldbsch tool by using the correct command line options to generate compatible types.</td>
</tr>
<tr>
<td>-6636</td>
<td>Invalid usage of NULL in an expression The compiler detected an NULL constant in an expression that will always evaluate to NULL or FALSE. For example, when writing IF var == NULL THEN, the program flow will never enter in the IF block.</td>
</tr>
<tr>
<td>-6637</td>
<td>Can not create index file for database dictionary 'dbname'. The compiler could not create the .42d index file for the database schema. Make sure that an existing .42d file is writable.</td>
</tr>
<tr>
<td>-6774</td>
<td>The license 'license-num' is no more valid. Please contact your vendor. <strong>Note:</strong> This error is non-trappable. The license number is no longer valid. Contact your vendor to obtain a new license number.</td>
</tr>
<tr>
<td>-6780</td>
<td>Invalid license request format. <strong>Note:</strong> This error is non-trappable. The request sent to the license manager was not recognized. Check that the version of the license manager is compatible with the runtime system.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| -6781  | Incompatible License Manager (flmprg) version. The minimum version required is version-num.  
**Note:** This error is non-trappable.  
The license manager is too old and is not compatible with the current runtime system.  
Call the support center to get a new version of the license manager. |
| -6783  | The license number 'license-num' is invalid. Please, contact your vendor.  
**Note:** This error is non-trappable.  
The license server failed to validate the license number.  
Call the support center to get a new license number. |
| -6784  | The license 'license-num' has expired. Please, contact your vendor.  
**Note:** This error is non-trappable.  
The license is time limited and it has expired.  
Call the support center to get a new license number. |
| -6785  | CPU limit exceeded. Please, contact your vendor.  
**Note:** This error is non-trappable.  
The license is CPU limited and the system has more CPUs as allowed.  
Call the support center to get a new license number. |
**Note:** This error is non-trappable.  
Check permissions on the lock/token directory (in FGLDIR or FLMDIR). |
| -6787  | This GRW license requires a DVM license with a valid maintenance date.  
**Note:** This error is non-trappable.  
GRW licenses with the option 'DVM under maintenance' require that the DVM maintenance key expiration date not be expired.  
Update the DVM maintenance key. |
| -6788  | Cannot get GRW report token information.  
**Note:** This error is non-trappable.  
Contact your support center. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -6789  | The installed license is invalid and cannot be used by this product.  
Note: This error is non-trappable. |
|        | The current license is not valid for the product you have installed.  
Contact your support center to get a license corresponding to the current installed product. |
| -6802  | Can not open Database dictionary 'schema-name'. Run schema extraction tool.  
The schema file does not exist or cannot be found.  
If the schema file exists, verify that the filename is spelled correctly, and that the file is in the current directory or the FGLDBPATH environment variable is set to the correct path. If the file does not exist, run the database schema extraction tool to create a schema file. |
| -6803  | A grammatical error has been found at 'line-number', expecting 'token-name'.  
This is a generic message for errors. |
| -6804  | 'form-name' form compilation was successful.  
This is an information message indicating that the form was compiled without problem. |
You have compiled your form with a version of the form compiler that is not compatible with that used for compiling the other source code.  
Compile your form file and related source code files using the same or compatible versions of the compilers. |
| -6807  | The label 'label-name' could not be used as column-title.  
The form file defines an invalid TABLE column title.  
Check for column titles which are not corresponding to column positions. |
| -6808  | The widget 'widget-name' can not be defined as array.  
The form file defines an item which is used as a matrix column.  
Review your form definition. |
| -6809  | The layout tag 'tag-name' is invalid, expecting: 'token-name'.  
The form compiler detected an invalid layout tag specification.  
Review your form definition. |
| -6810  | The attribute 'attribute-name' is invalid for item type 'type-name'.  
The form compiler detected an invalid attribute definition for this item type.  
Review your form definition and check for invalid attributes. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -6811  | Syntax error near 'token-1', expecting token-2.  
A general syntax error message that indicates the location of the problem code and what code was expected.  
Review your code and make the necessary corrections. |
| -6812  | Unterminated char constant.  
The form compiler detected an unterminated character constant.  
Review your form definition and check for missing quotes or double-quotes. |
| -6813  | The element 'element-name' conflicts with group-box 'group-name'.  
You have used the same name for an element and for a group-box.  
Review your form definition and ensure that the names used are unique. |
| -6814  | All members of the SCREEN RECORD 'screen-record-name' must reference the same Table or ScrollGrid.  
The shown screen record references multiple tables or scrollgrids in your form file.  
Review your form definition and use one unique table for a given screen record. |
| -6815  | Invalid indentation in between braces.  
The LAYOUT section of your form defines an invalid indentation.  
Review your form definition and check for corresponding indentations. |
| -6817  | TABLE container defined without a SCREEN RECORD in the INSTRUCTION section.  
The minimum value of the defined attribute must be lower than the maximum value.  
Review your code and make the necessary corrections. |
| -6818  | Min value must be lower that Max value.  
The minimum value of the defined attribute must be lower than the maximum value.  
Review your code and make the necessary corrections. |
| -6819  | Number of elements in the SCREEN RECORD must match the number of columns in TABLE container.  
The elements defined in the screen record differs from the columns used for the TABLE container.  
Review your form definition and add missing table columns to the screen record, order does not matter. |
| -6820  | ScrollGrid and/or Group layout tags cannot be nested.  
The form definition has nested ScrollGrid and/or Group layout tags. These tags cannot be nested.  
Review your form definition and make the necessary corrections. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -6821  | HBOX tags cannot be used for ARRAYS.  
The form definition is using an HBOX tag for an array, which is not permitted.  
Review your form definition and make the necessary corrections. |
| -6822  | Escaped graphical characters are not accepted in GRID sections.  
You try to use Text User Interface graphics in the new GRID container.  
This is not allowed, use GROUPs instead. |
| -6823  | Close tag does not have a matching tag above.  
The form definition has a close tag without a prior matching open tag. Open tags and close tags must match.  
Review your form definition file and make the necessary corrections. |
| -6824  | The table 'table-name' is empty.  
The form layout defines a table layout tag identified by tablename, but nothing was found directly under this table that defines a column or a column title.  
Append columns to the table layout region. |
| -6825  | The tag 'tag-name' overlaps with table 'table-name'.  
In the form layout, tagname overlaps the layout region of tablename and makes it invalid.  
Move or remove tagname, or redefine the layout region of tablename. |
| -6826  | Checked value must be different from unchecked value for field 'field-name'.  
The VALUECHECKED and VALUEUNCHECKED attributes have the same value. This makes no sense because these attributes define the values corresponding to the checked and unchecked states of a checkbox.  
Use different values for these attributes. |
| -6827  | Duplicated item key found for field 'field-name'.  
The ITEMS attribute of field fieldname defines item keys with the same value.  
Check ITEMS attribute and use unique key values. Note that " and NULL are equivalent. |
| -6828  | The attribute attribute-name must belong to a column of a TABLE.  
A form item uses an attribute that references a form field which is not defined or does not belong to the TABLE.  
Check the ATTRIBUTES section for invalid column references. |
| -6829  | The column column-name referenced by the attribute-name attribute must belong to the TABLE.  
A form item uses an attribute that references a form field which is not defined or does not belong to the TABLE.  
Check the ATTRIBUTES section for invalid column references. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -6830  | Not implemented (yet): feature-name  
The feature or syntax you are using is not implemented yet.  
This feature cannot be used in the Genero version you have installed. |
| -6831  | At least one member of the SCREEN RECORD 'screen-record-name'  
must not be a PHANTOM field.  
A screen record is defined with form fields that are all defined as PHANTOM fields.  
At least one screen record field must not be a PHANTOM field. |
| -6832  | Repeated screen tags 'tag-name' are misaligned, must align on X  
or Y.  
The layout defines multiple tags with the same name, but these are not properly aligned in the X  
or Y direction.  
Edit the form file and make sure that repeated tags are correctly aligned. |
| -6833  | Invalid TREE definition: the field 'field-name' must be an EDIT  
or LABEL.  
The form defines a TREE container with the field column defined with a wrong item type.  
Replace the item type by EDIT or LABEL. |
| -6834  | Invalid TREE definition: the field 'field-name' must be defined  
for the SCREEN RECORD.  
The form defines a TREE container with an invalid field set.  
Check that mandatory fields such as node name, parent id and node id fields are defined. |
| -6835  | The fields specified in the THRU option appear in the reverse  
order.  
The form defines a screen record by using the THRU or THROUGH keyword, but the first field  
is defined after the last field in the ATTRIBUTES section.  
Exchange the field names specified in the screen record definition, or review the declaration  
order in ATTRIBUTES. |
| -6836  | Invalid TREE definition: the attribute 'attribute-name'  
conflicts with id or parentid.  
The .per form defines a TREE with invalid configuration. You have probably used the same  
field for the named attribute and for IDCOLUMN or PARENTIDCOLUMN.  
Review the form definition and configure the TREE properly. You must use dedicated columns  
for the attributes mentioned. |
| -6837  | Invalid AGGREGATE definition: must be located below a table  
column.  
The .per form defines an AGGREGATE form item with a field tag that is not aligned under a  
table column field tag.  
Review the table layout and make sure that all aggregate fields are properly aligned and placed  
below column tags. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -6838  | This area is reserved for AGGREGATEs.  
The .per form defines a TABLE with aggregate fields, but not all aggregate fields are declared with the AGGREGATE item type.  
Review the field definitions in the ATTRIBUTES section. |
| -6839  | The screen tag 'tag-name' can not be defined in a TABLE.  
The .per form defines a TABLE with columns using different field tag names, an no AGGREGATE field is defined.  
Review columns of the TABLE, each field tag of a given column must use the same tag name, except if you want to define an AGGREGATE field. |
| -6841  | FORM not contain TOPMENU or TOOLBAR.  
The form layout includes an external form specification file containing a TOPMENU or a TOOLBAR. Remove these sections from the included form file. |
| -6842  | FORM is out of date.  
The form layout includes an external form specification file the was compiled with a older version. |
| -6843  | A resizable SCROLLGRID requires the definition of exactly one template.  
A resizable SCROLLGRID (WANTFIXEDPAGESIZE=NO) must define a single row template. |
| -6844  | INITIALPAGESIZE requires a resizable SCROLLGRID.  
When using the INITIALPAGESIZE attribute, the SCROLLGRID must be defined as resizable with the WANTFIXEDPAGESIZE=NO attribute. |
| -6845  | The display field label 'field-name' has already been defined.  
The field item tag is defined several times in different containers of the LAYOUT section. |
| -6846  | The screen tag 'tagname' can not be defined in a SCROLLGRID  
A SCROLLGRID contains a layout tag such as <TABLE > or <TREE >. This is not allowed. |
| -6847  | TABINDEX has to be unique  
Some elements of the form define the same TABINDEX. Review the ATTRIBUTE section and make sure that all TABINDEX values are unique. |
| -6848  | All TABLE columns must have the same size.  
In a TABLE or TREE container, all columns must be defined with the same width and height in the LAYOUT section. |
| -6849  | STRETCH=value conflicts with WANTFIXEDPAGESIZE.  
The STRETCH attribute values Y and BOTH are not consistent with WANTFIXEDPAGESIZE:  
The WANTFIXEDPAGESIZE attribute is equivalent to STRETCH=X.  
Remove the WANTFIXEDPAGESIZE attribute in favor of the STRETCH attribute. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -8000  | Dom: Node not found.  
The node was not found in the current document.  
Review your code. |
The document passed to the DOM API is not a valid document.  
Review your code. |
| -8002  | Dom: Invalid usage of NULL as parameter.  
NULL cannot be used at this place.  
Review your code. |
| -8003  | Dom: A node is inserted somewhere it doesn't belong.  
**Note:** This error is non-trappable.  
You try to insert a node under a parent node which does not allow this type of nodes.  
Check for the possible nodes and review your code. |
| -8004  | Sax: Invalid hierarchy.  
The SAX handler encountered an invalid hierarchy.  
Make sure parent/child relations are respected. |
| -8005  | Deprecated feature: feature-name  
The feature you are using will be removed in a next version.  
A replacement for the feature is normally available. |
| -8006  | The string resource file 'file-name' cannot be found.  
**Note:** This error is non-trappable.  
The given string file was not found.  
Check if file exists and if path is valid. |
| -8007  | The string resource file 'file-name' cannot be read.  
**Note:** This error is non-trappable.  
The given string file was unable to be read.  
Check if file exists and if user has read permissions. |
| -8008  | There is no string text defined for the 'key-name' string key.  
The runtime system was unable to find a string resource corresponding to the shown key.  
Check if the key is defined in one of the resource files. |
| -8009  | String resource syntax error near 'token-name', expecting token.  
The string file compiler detected a syntax error.  
Check for invalid syntax in the .str file. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -8012  | Duplicate string key 'key-name' (file-name : line) IGNORE LINE.  
The string file compiler detected duplicated string keys.  
Review the .str file and remove duplicated keys. |
| -8013  | The string file 'file-name' can not be opened for writing.  
The string file compiler was unable to write to the specified string file.  
Make sure the user has write permissions and file name is valid. |
| -8014  | The string file 'file-name' can not be read.  
The runtime system was unable to read from the specified string file.  
Make sure the user has read permissions. |
| -8015  | Field (field-name) in ON CHANGE clause not found in form.  
The field used in the ON CHANGE clauses was not found in the form specification file.  
Make sure the field name of the ON CHANGE clause matches a valid form field. |
| -8016  | You cannot have multiple ON CHANGE clauses for the same field.  
It is not possible to specify multiple ON CHANGE clauses using the same field.  
Remove unnecessary ON CHANGE clauses. |
| -8017  | SFMT: Invalid % index used.  
The format string is not valid.  
Check for invalid % positions. |
| -8018  | SFMT: Format error.  
The format string is not valid.  
Check for invalid % positions. |
| -8020  | Multiple ON ACTION clauses with the same action name appear in the statement.  
It is not possible to specify multiple ON ACTION clauses using the same action name.  
Remove unnecessary ON ACTION clauses. |
| -8021  | Multiple ON KEY clauses with the same key name appear in the statement.  
It is not possible to specify multiple ON KEY clauses using the same key.  
Remove unnecessary ON KEY clauses. |
| -8022  | Dom: Cannot open xml-file.  
The given file failed to load.  
Check file name and user permissions. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -8023  | Dom: The attribute 'attribute-name' does not belong to node 'node-type'.  
You try to set an attribute to a node which does not have such attribute.  
This is not allowed, review your code. |
| -8024  | Dom: Character data can not be created here.  
**Note:** This error is non-trappable.  
You try to create a text node under a node which does not allow such nodes.  
This is not allowed, review your code. |
| -8025  | Dom: Cannot set attributes of a character node.  
You try to set attributes in a text node.  
This is not allowed, review your code. |
| -8026  | Dom: The attribute 'attribute-name' can not be removed: the node 'node-type' belongs to the user-interface.  
**Note:** This error is non-trappable.  
You try to remove a mandatory attribute from an AUI node.  
You can only change the value of this attribute, try 'none' or an empty string. |
| -8027  | Sax: can not write.  
The SAX handlers were unable to write to the destination file.  
Make sure the file path is correct and the user has write permissions. |
| -8029  | Multiple inclusion of the source file 'file-name'.  
The preprocessor detected that the specified file was included several times by the same source.  
Remove unnecessary file inclusions. |
| -8030  | The full path to the source file 'file-name' is too long.  
The preprocessor does not support very long file names.  
Rename the file. |
| -8031  | The source file 'file-name' cannot be read.  
The preprocessor failed to read the file specified.  
Make sure the user has read permissions. |
| -8032  | The source file 'file-name' cannot be found.  
The preprocessor failed to find the file specified.  
Make sure the file exists. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-8033</td>
<td>Extra token found after 'directive-name' directive. The preprocessor detected an unexpected token after the shown directive. Review your code and make the necessary corrections.</td>
</tr>
<tr>
<td>-8034</td>
<td>feature-name: This feature is not implemented. This preprocessor feature is not supported. Review your code and make the necessary corrections.</td>
</tr>
<tr>
<td>-8035</td>
<td>The macro 'macro-name' has already been defined. The preprocessor found a duplicated macro definition. Review your code and make the necessary corrections.</td>
</tr>
<tr>
<td>-8036</td>
<td>A &amp;else directive found without corresponding &amp;if, &amp;ifdef or &amp;ifndef directive. The preprocessor detected an unexpected &amp;else directive. Review your code and make the necessary corrections.</td>
</tr>
<tr>
<td>-8037</td>
<td>A &amp;endif directive found without corresponding &amp;if, &amp;ifdef or &amp;ifndef directive. The preprocessor detected an unexpected &amp;endif directive. Review your code and make the necessary corrections.</td>
</tr>
<tr>
<td>-8038</td>
<td>Invalid preprocessor directive &amp; name found. The preprocessor directive shown in the error message does not exist. Review your code and check valid macros.</td>
</tr>
<tr>
<td>-8039</td>
<td>Invalid number of parameters for macro-name. The number of parameters of the preprocessor macro shown in the error message does not match de number of parameters in the definition of this macro. Review your code and check for the number of parameters.</td>
</tr>
<tr>
<td>-8040</td>
<td>Lexical error: Unclosed string. The compiler detected an unclosed string and cannot continue. Review your code and make the necessary corrections.</td>
</tr>
<tr>
<td>-8041</td>
<td>Unterminated condition &amp;if or &amp;else. The preprocessor found an un-terminated conditional directive. Review the definition of this directive.</td>
</tr>
<tr>
<td>-8042</td>
<td>The operator '##' can only be used with identifiers and numbers. token is not allowed. The preprocessor found an invalid usage of the ## string concatenation operator. Review the definition of this macro.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-8043</td>
<td>Could not run FGLPP, command used: command</td>
</tr>
<tr>
<td></td>
<td>The compiler failed to run the preprocessor command shown in the error message.</td>
</tr>
<tr>
<td></td>
<td>Make sure the preprocessor command exists.</td>
</tr>
<tr>
<td>-8044</td>
<td>Lexical error: Unclosed comment.</td>
</tr>
<tr>
<td></td>
<td>The compiler detected an unclosed comment and cannot continue.</td>
</tr>
<tr>
<td></td>
<td>Review your code and make the necessary corrections.</td>
</tr>
<tr>
<td>-8045</td>
<td>This type of statement can only be used within an INPUT, INPUT ARRAY, DISPLAY ARRAY, CONSTRUCT or MENU statement.</td>
</tr>
<tr>
<td></td>
<td>This statement has not been used within a valid interactive statement, which must be terminated appropriately with END INPUT, END INPUT ARRAY, END DISPLAY ARRAY, or END CONSTRUCT.</td>
</tr>
<tr>
<td></td>
<td>Review your code and make the necessary corrections.</td>
</tr>
<tr>
<td>-8046</td>
<td>This type of statement can only be used within an INPUT, INPUT ARRAY, DISPLAY ARRAY or CONSTRUCT statement.</td>
</tr>
<tr>
<td></td>
<td>This statement has not been used within a valid interactive statement, which must be terminated appropriately with END INPUT, END INPUT ARRAY, END DISPLAY ARRAY, or END CONSTRUCT.</td>
</tr>
<tr>
<td></td>
<td>Review your code and make the necessary corrections.</td>
</tr>
<tr>
<td>-8047</td>
<td>Invalid use of 'dialog'. Must be used within an INPUT, INPUT ARRAY, DISPLAY ARRAY or CONSTRUCT statement.</td>
</tr>
<tr>
<td></td>
<td>The predefined keyword DIALOG has not been used within a valid interactive statement, which must be terminated appropriately with END INPUT, END INPUT ARRAY, END DISPLAY ARRAY, or END CONSTRUCT.</td>
</tr>
<tr>
<td></td>
<td>Review your code and make the necessary corrections.</td>
</tr>
<tr>
<td>-8048</td>
<td>An error occurred while preprocessing the file 'file-name'.</td>
</tr>
<tr>
<td></td>
<td>Compilation ends.</td>
</tr>
<tr>
<td></td>
<td>The Genero BDL preprocessor failed to parse the whole source file and stopped compilation.</td>
</tr>
<tr>
<td></td>
<td>Review the source code and check for not well formed &amp; preprocessor macros.</td>
</tr>
<tr>
<td>-8049</td>
<td>The program cannot ACCEPT (INPUT</td>
</tr>
<tr>
<td></td>
<td>ACCEPT XXX has not been used within a valid interactive statement, which must be terminated appropriately with END INPUT, END PROMPT, or END INPUT ARRAY.</td>
</tr>
<tr>
<td></td>
<td>Review your code and make the necessary corrections.</td>
</tr>
<tr>
<td>-8050</td>
<td>Dom: Invalid XML data found in source.</td>
</tr>
<tr>
<td></td>
<td>ACCEPT DISPLAY has not been used within a valid DISPLAY ARRAY statement, which must be terminated with END DISPLAY ARRAY.</td>
</tr>
<tr>
<td></td>
<td>Review your code and make the necessary corrections.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-8051</td>
<td>Sax: Invalid processing instruction name.</td>
</tr>
<tr>
<td></td>
<td>The om.SaxDocumentHandler.processingInstruction() does not allow invalid processing instruction names such as 'xml'.</td>
</tr>
<tr>
<td></td>
<td>&lt;?xml ..?&gt; is not a processing instruction, it is reserved to define the XML file text declaration. You must use another name.</td>
</tr>
<tr>
<td>-8052</td>
<td>Illegal input sequence. Check LANG.</td>
</tr>
<tr>
<td></td>
<td>The compiler encountered an invalid character sequence. The source file uses a character sequence which does not match the locale settings (LANG). Check source file and locale settings.</td>
</tr>
<tr>
<td>-8053</td>
<td>Unknown preprocessor directive 'directive-name'.</td>
</tr>
<tr>
<td></td>
<td>The preprocessor directive shown in the error message is not a known directive.</td>
</tr>
<tr>
<td></td>
<td>Check for typo errors and read the documentation for valid preprocessor directives.</td>
</tr>
<tr>
<td>-8054</td>
<td>Unexpected preprocessor directive.</td>
</tr>
<tr>
<td></td>
<td>The preprocessor encountered an unexpected directive.</td>
</tr>
<tr>
<td></td>
<td>Remove the directive.</td>
</tr>
<tr>
<td>-8055</td>
<td>The resource file 'file-name' contains unexpected data.</td>
</tr>
<tr>
<td></td>
<td>The XML resource file shown in the error message does not contain the expected nodes. For example, you try to load a ToolBar with ui.Interface.loadActionDefaults().</td>
</tr>
<tr>
<td></td>
<td>Check if the XML file contains the node types expected for this type of resource.</td>
</tr>
<tr>
<td>-8056</td>
<td>XPath: Unclosed quote at position integer.</td>
</tr>
<tr>
<td></td>
<td>The XPath parser found an unexpected quote at the given position.</td>
</tr>
<tr>
<td></td>
<td>Review the XPath expression.</td>
</tr>
<tr>
<td>-8057</td>
<td>XPath: Unexpected character 'char' at position pos.</td>
</tr>
<tr>
<td></td>
<td>The XPath parser found an unexpected character at the given position.</td>
</tr>
<tr>
<td></td>
<td>Review the XPath expression.</td>
</tr>
<tr>
<td>-8058</td>
<td>XPath: Unexpected token/string 'token-name' at position pos.</td>
</tr>
<tr>
<td></td>
<td>The XPath parser found an unexpected token or string at the given position.</td>
</tr>
<tr>
<td></td>
<td>Review the XPath expression.</td>
</tr>
<tr>
<td>-8059</td>
<td>SQL statement or language instruction with vendor proprietary syntax.</td>
</tr>
<tr>
<td></td>
<td>The compiler found an SQL statement which is using a database specific syntax. This statement will probably not run on other database servers as the current.</td>
</tr>
<tr>
<td></td>
<td>Review the SQL statement and use standard/common syntax and features.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| -8060  | Spacer items are not allowed inside a SCREEN sections.  
The form contains spacer items in a SCREEN section, while these are only allowed in LAYOUT.  
Review the form specification file. |
| -8061  | A TABLE row should not be defined on multiple lines.  
All columns of a row in a TABLE container must be in a single line.  
Use a SCROLLGRID if you want to show row cells on multiple lines. |
| -8063  | The client connection timed out.  
The runtime system failed to establish the connection with the front-end after a given time.  
This can for example happen during a file transfer, when the front-end takes too much time to respond to the runtime system.  
Check that your network connection is working properly. |
| -8064  | File transfer interrupted.  
An interruption was caught during a file transfer.  
File was not transferred, you need to redo the operation. |
| -8065  | Network error during file transfer.  
A socket error was caught during a file transfer.  
Check that your network connection is working properly. |
| -8066  | Could not write destination file for file transfer.  
The runtime system failed to write the destination file for a transfer.  
Make sure the file path is correct and check that user has write permissions. |
| -8067  | Could not read source file for file transfer.  
The runtime system was unable to read the source file to transfer.  
Make sure the file path is correct and check that user has read permissions. |
| -8068  | File transfer protocol error (invalid state).  
The runtime system encountered a problem during a file transfer.  
A network failure has probably raised this error. |
| -8069  | File transfer not available.  
File transfer feature is not supported.  
Make sure the front-end supports file transfer. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -8070  | The localized string file 'file-name' is corrupted.  
**Note:** This error is non-trappable.  
The shown string resource file is invalid (probably invalid multibyte characters corrupt the file).  
Check for locale settings (LANG), make sure the .str source uses valid characters and recompile it. |
| -8071  | 'symbol-name' is already defined.  
The form file defines several elements of the same type with the same name.  
Review the form file and use unique identifiers. |
| -8072  | Statement must terminate with ';'.  
An ESQL/C preprocessor directive is not terminated with a semicolon.  
Add a semicolon to the end of the directive. |
| -8073  | Invalid 'include' directive file name.  
An include preprocessor directive is using an invalid file name.  
Check the file name. |
| -8074  | A &elif directive found without corresponding &if, &ifdef or &ifndef directive.  
The preprocessor found an &elif directive with no corresponding &if.  
Add the &if directive before the &elif, or remove the &elif. |
| -8075  | The compiler plugin name could not be loaded.  
gflcomp failed to load the plugin because it was not found.  
Make sure the plugin exists and can be loaded. |
| -8076  | The compiler plugin name does not implement the required interface.  
gflcomp failed to load the plugin because the interface is invalid.  
Check if the plugin corresponds to the version of the compiler. |
| -8077  | The attribute 'attribute-name' has been defined more than once.  
The variable attribute shown in the error message was defined multiple times.  
Review the variable definition and remove duplicated attributes. |
| -8078  | The attribute 'attribute-name' is not allowed.  
The variable attribute shown in the error message is not allowed for this type of variable.  
Review the possible variable attributes. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-8079</td>
<td>An error occurred while parsing the XML file. The runtime system failed to parse an XML file, which is probably not using a valid XML format. Check for XML format typos and if possible, validate the XML file with a DTD.</td>
</tr>
<tr>
<td>-8080</td>
<td>Could not open xml file. The specified XML file cannot be opened. Make sure the file exists and has access permissions for the current user.</td>
</tr>
<tr>
<td>-8081</td>
<td>Invalid multibyte character has been encountered. A compiler found an invalid multibyte character in the source and cannot compile the form or module. Check locale settings (LANG) and verify if there are no invalid characters in your sources.</td>
</tr>
<tr>
<td>-8082</td>
<td>The item 'item-name' is used in an invalid layout context. The form item name is used in a layout part which does not support this type of form item. This error occurs for example when you try to define a BUTTON as a TABLE column. Review your form definition file and use correct item types.</td>
</tr>
<tr>
<td>-8083</td>
<td>NULL pointer exception. The program is using calling a method thru an object variable which is NULL. You must assign an object reference to the variable before calling a method.</td>
</tr>
<tr>
<td>-8084</td>
<td>Can't open socket: description The channel object failed to open a TCP socket. See the description for more details. Make sure the IP address and port are correct.</td>
</tr>
<tr>
<td>-8085</td>
<td>Unsupported mode for 'open socket'. You try to open a channel with an unsupported mode. See channel documentation for supported modes.</td>
</tr>
<tr>
<td>-8086</td>
<td>The socket connection timed out. Socket failed to establish a connection and timeout expired. Check all network layers and try again.</td>
</tr>
<tr>
<td>-8087</td>
<td>File error in BYTE or TEXT readFile or writeFile. File I/O error occurred while reading from or writing to a file. Verify the file name, content and access permissions.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| -8088 | The dialog attribute 'attribute-name' is not supported.  
**Note:** This error is non-trappable.  
A dialog instruction was declared with an ATTRIBUTES clause containing an unsupported option.  
Review the ATTRIBUTES clause and remove unsupported option. |
| -8089 | Action 'action-name' not found in dialog.  
You try to use an action name that does not exist in the current dialog.  
Verify if name of the action is defined by an ON ACTION clause. |
| -8090 | Field 'field-name' already used in this DIALOG.  
The DIALOG instruction binds the same field-name or screen-record multiple times.  
Review all sub-dialog blocks and check the field-names / screen-records. |
| -8091 | The clause 'clause-name' appears more than once.  
You have defined the same dialog control block multiple times. For example, AFTER ROW was defined twice.  
Remove the un-necessary control blocks. |
| -8092 | At least one field for this INPUT ARRAY must be editable.  
**Note:** This error is non-trappable.  
An INPUT ARRAY is executed on fields that are read-only. At least one field must be editable and active.  
Review the form specification file or check that at least one field is active. |
| -8093 | Multi-range selection is not available in this context.  
You try to use multi-range selection but it is not possible in the current dialog type.  
Disable this feature. |
| -8094 | Multi-range selection is not available in this context.  
You try to use multi-range selection but it is not possible in the current dialog type.  
Disable this feature. |
| -8095 | Cannot change selection flag for this range of rows.  
An attempt of selection flag modification with DIALOG.setSelectionRange() failed because the range is out of bounds or because there is no multi-range selection available in this context.  
Make sure you can use multi-range selection, and check the start and end index of the range. |
| -8096 | General SQL Warning, check SQLCA.SQLERRD[2] or SQLSTATE.  
The last SQL statement has generated an SQL warning setting the SQLCA.SQLAWARN flags.  
Program execution can continue; however, it is recommended that you take care and check the native SQL code and the SQL message in SQLERRMESSAGE. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -8097  | Value too large to fit in a TINYINT.  
The TINYINT data type can accept numbers with a value range from -128 to +127.  
To store numbers that are outside this range, redefine the column or variable to use the SMALLINT or INTEGER type. |
| -8098  | ON FILL BUFFER conflicts with DISPLAY ARRAY as a tree.  
The DISPLAY ARRAY instruction is using a treeview as decoration, but it implements also an ON FILL BUFFER trigger to do paged mode. The paged mode is not possible when using a treeview, because all rows of visible nodes are required (i.e. the dialog cannot display a tree only with a part of the dataset).  
To populate dynamically the array for a treeview, use the ON EXPAND to add new nodes and ON COLLAPSE to remove nodes. |
| -8099  | The form 'form-name' is incompatible with the current runtime version. Rebuild you forms.  
The .42f form was probably compiled with an earlier version as the current runtime system.  
Recompile the form with the fglform compiler corresponding to the current fglrune. |
| -8100  | Attempt to access a closed dialog.  
A call to a DIALOG class method is done with a dialog object that has terminated.  
Review the program logic and call the DIALOG methods only for active running dialogs. |
| -8101  | The TABLE column tag 'tag-name' appears multiple times in the row definition.  
A TABLE column can only be used once in the row definition, you have probably repeated the same screen tag by mistake.  
Modify the TABLE row definition in the layout section in order to use each column only once. |
| -8102  | Syntax error in preprocessor directive.  
The source file contains a preprocessor macro with an invalid syntax.  
Check the preprocessor manual page and fix the syntax error. |
| -8103  | The source and destination file name of a file transfer must not be NULL or empty.  
The program is doing an fgl_getfile() or fgl_putfile() and the source or destination file name is NULL or empty.  
Provide a valid file name for both source and destination parameters. |
| -8104  | Cannot read from TUI: system-error.  
A program running in text mode (FGGUI=0) failed to read from console input stream.  
Check the console/terminal settings. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-8105</td>
<td>Not found.</td>
</tr>
<tr>
<td></td>
<td>This message displayed by the runtime system when a record was not found. It can be displayed in different contexts, for example when searching a record in a list with the built-in search feature.</td>
</tr>
<tr>
<td>-8106</td>
<td>Field (field-name) in ON ACTION INFIELD not found in form.</td>
</tr>
<tr>
<td></td>
<td>The field name used in an ON ACTION INFIELD action handle was not found in the form.</td>
</tr>
<tr>
<td></td>
<td>Make sure you are using the correct field name and field prefix (table name or screen record name).</td>
</tr>
<tr>
<td>-8107</td>
<td>FGL_LENGTH_SEMANTICS environment variable is invalid. Valid values are BYTE and CHAR</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> This error is <strong>non-trappable</strong>.</td>
</tr>
<tr>
<td></td>
<td>The value specified in the FGL_LENGTH_SEMANTICS environment variable must be BYTE or CHAR.</td>
</tr>
<tr>
<td>-8108</td>
<td>Subdialog dialog-name: already active</td>
</tr>
<tr>
<td></td>
<td>The sub-dialog is already in use.</td>
</tr>
<tr>
<td>-8109</td>
<td><strong>JSON parse error:</strong> description</td>
</tr>
<tr>
<td></td>
<td>Verify the input string passed to the JSON parsing function. See the description for more details.</td>
</tr>
<tr>
<td>-8110</td>
<td><strong>JSON stringify error:</strong> description</td>
</tr>
<tr>
<td></td>
<td>The JSON serialization failed. See the description for more details.</td>
</tr>
<tr>
<td>-8111</td>
<td>Can not happen: description</td>
</tr>
<tr>
<td></td>
<td>The runtime system encounters an unexpected situation. The message is displayed to the user, but the program flow will continue. This unexpected situation must be fixed by programmers.</td>
</tr>
<tr>
<td>-8112</td>
<td>Illegal argument.</td>
</tr>
<tr>
<td></td>
<td>The runtime system instruction, function or object method does not expect the value passed as argument. This can for example occur when calling the <code>Array.sort()</code> method with an invalid array-record member name.</td>
</tr>
<tr>
<td>-8113</td>
<td>The actions DETAILACTION and DOUBLECLICK must be different.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> This error is <strong>non-trappable</strong>.</td>
</tr>
<tr>
<td></td>
<td>The DETAILACTION and DOUBLECLICK attributes are used in DISPLAY ARRAY to configure a table decoration and behavior. These attributes cannot define the same action.</td>
</tr>
<tr>
<td>-8114</td>
<td>Completer item list too long. The list must not contain more than 50 items.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> This error is <strong>non-trappable</strong>.</td>
</tr>
<tr>
<td></td>
<td>The array passed to the setCompleterItems() dialog method is too long, reduce the list.</td>
</tr>
<tr>
<td>-8115</td>
<td>Character to boolean conversion error.</td>
</tr>
<tr>
<td></td>
<td>The array passed to the setCompleterItems() dialog method is too long, reduce the list.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| -8116  | Illegal context.  
The current instruction is used on a wrong context. |
| -8117  | '##' cannot appear at start of macro expansion.  
The preprocessor operator ## must join two identifiers (a ## b). |
| -8118  | '##' cannot appear at end of macro expansion.  
The preprocessor operator ## must join two identifiers (a ## b). |
| -8119  | '#' is not followed by a macro parameter.  
The preprocessor operator # must be followed by a parameter of the macro. |
| -8120  | File transfer: copy file to file-name failed.  
The runtime system failed to copy the specified file. |
| -8121  | File transfer: remove file file-name failed.  
The runtime system failed to delete the specified file. |
| -8122  | File transfer: touch file file-name failed.  
The runtime system failed to touch the specified file. |
| -8123  | \x used with no following hex digits.  
The \xNN character code is malformed. |
| -8124  | hex escape sequence out of range.  
The \xNN character code contains an invalid hexadecimal value. |
| -8125  | File transfer: create symbolic link file-name failed.  
A symbolic link required by the file transfer was unable to be created. |
| -8126  | Image to font mapping: Font file file-name not found.  
**Note:** This error is non-trappable.  
The font file was not found, check FGLIMAGEPATH environment variable. |
| -8127  | Image to font mapping: Format error in file file-name.  
**Note:** This error is non-trappable.  
The image to font mapping file contains errors. |
| -8128  | Image to font mapping: Cannot open file file-name.  
**Note:** This error is non-trappable.  
The image to font mapping file was not found, check FGLIMAGEPATH environment variable. |
| -8129  | No current row.  
In order to use ui.Dialog methods such as getFieldValue(), you must set the current row with the ui.Dialog.setCurrentRow() method. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-8130</td>
<td>FOCUSONFIELD conflicts with DISPLAY ARRAY as tree.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> This error is <strong>non-trappable</strong>.</td>
</tr>
<tr>
<td></td>
<td>A DISPLAY ARRAY dialog FOCUSONFIELD attribute cannot be used with a TREE container.</td>
</tr>
<tr>
<td>-8131</td>
<td>SQL parameter index out of bounds.</td>
</tr>
<tr>
<td></td>
<td>The index passed as argument to the <code>setParameterValue()</code> method of <code>base.SqlHandle</code> is lower as 1, or is greater as the number of ? placeholders in the SQL statement.</td>
</tr>
<tr>
<td>-8132</td>
<td><code>profile: 'entry-name-1' conflicts with 'entry-name-2'</code>.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> This error is <strong>non-trappable</strong>.</td>
</tr>
<tr>
<td></td>
<td>The FGLPROFILE configuration file(s) define exclusive entries that cannot be used together.</td>
</tr>
<tr>
<td>-8133</td>
<td>Can not initialize form: reason.'.</td>
</tr>
<tr>
<td></td>
<td>The form file could not be initialized. This error occurs for example when a <code>COMBOBOX INITIALIZER</code> function cannot be found. This error cannot be trapped.</td>
</tr>
<tr>
<td>-8134</td>
<td>Call function <code>function-name</code>: wrong number of parameters, expect <code>caller-count</code>, have <code>definition-count</code>.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> This error is <strong>non-trappable</strong>.</td>
</tr>
<tr>
<td></td>
<td>The function was defined with a wrong number of parameters. Check the function signature, and define the same number of parameters as required by the caller.</td>
</tr>
<tr>
<td>-8135</td>
<td>Call function <code>function-name</code>: wrong number of return values, expect <code>caller-count</code>, have <code>definition-count</code>.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> This error is <strong>non-trappable</strong>.</td>
</tr>
<tr>
<td></td>
<td>The function has returned a wrong number of values. Check the function body for <code>RETURN</code> instructions and return the number of values required by the caller.</td>
</tr>
<tr>
<td>-8200</td>
<td><code>apidoc: parameter name 'param-name' is invalid</code>.</td>
</tr>
<tr>
<td></td>
<td>The compiler has detected a comment error while extracting the source documentation: The <code>@param</code> variable name is not in the list of parameters in the next <code>FUNCTION</code> definition.</td>
</tr>
<tr>
<td></td>
<td>Check the function parameter name.</td>
</tr>
<tr>
<td>-8201</td>
<td><code>apidoc: tag missing: @param param-name</code>.</td>
</tr>
<tr>
<td></td>
<td>The compiler has detected a comment error while extracting the source documentation: There is a missing <code>@param</code> tag that describes a parameter of the next <code>FUNCTION</code> definition.</td>
</tr>
<tr>
<td></td>
<td>Check the function parameter name.</td>
</tr>
<tr>
<td>-8202</td>
<td><code>apidoc: invalid tag name @ tag-name</code>.</td>
</tr>
<tr>
<td></td>
<td>The compiler has detected a comment error while extracting the source documentation: The <code>@</code> tag-name tag is not a known tag name.</td>
</tr>
<tr>
<td></td>
<td>Check for typo errors in the tag name.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| -8300  | Cannot load java shared library. Reason: system-error  
  **Note:** This error is **non-trappable**.  
The runtime system failed to load the JVM shared library (or DLL).  
Make sure that a JRE is installed on the machine and check the environment  
(LD_LIBRARY_PATH on UNIX™ or PATH on Windows®). |
| -8301  | Cannot create java VM.  
  **Note:** This error is **non-trappable**.  
The runtime system loaded the JVM shared library (or DLL), but was unable to initialize the  
Java VM with a call to JNI_CreateJavaVM().  
Check that the Java requirements and resources needs to create a Java VM. |
| -8302  | Array element type is not a Java type.  
The fglcomp compiler detected a Java Array definition which is not using a Java type for the  
elements.  
Review the DEFINE statement and use a Java type. |
| -8303  | Java is not supported.  
The platform you are using does not support a recent Java version required by Genero.  
**Note:** This error is **non-trappable**.  
You cannot use the Java interface in this operating system, you must review your source code  
and remove all Java related parts. |
| -8304  | Cannot assign a value to final variable 'variable-name'.  
The program tries to set a Java class variable which is not writable.  
Review the program logic. |
| -8305  | The Java variable 'variable-name' can not be used here.  
The program tries to use a Java class variable in an invalid context. For example, a Java class  
variable is used in an INPUT instruction.  
Review the program logic and use a regular Genero BDL variable. |
| -8306  | Java exception thrown: java-exception-text.  
A Java exception has been thrown while executing Java code.  
Check the exception text and review the code. |
| -8307  | Java object required.  
A Java object reference is expected by the instruction. This error typically occurs in a CAST()  
or INSTANCEOF().  
Check the expression used in the instruction and make sure it references a Java object. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-8400</td>
<td>module.name has private access.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> This error is non-trappable.</td>
</tr>
<tr>
<td></td>
<td>An instruction references a module function or module variable which is</td>
</tr>
<tr>
<td></td>
<td>declared as private.</td>
</tr>
<tr>
<td></td>
<td>Make the function or variable public in the imported module.</td>
</tr>
<tr>
<td>-8401</td>
<td>Reference to name is ambiguous.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> This error is non-trappable.</td>
</tr>
<tr>
<td></td>
<td>A function or variable referenced without the module prefix, but exists in</td>
</tr>
<tr>
<td></td>
<td>several imported modules. This error can also be printed by the compiler</td>
</tr>
<tr>
<td></td>
<td>for Java calls.</td>
</tr>
<tr>
<td></td>
<td>Add the module prefix before the object name to remove the ambiguity.</td>
</tr>
<tr>
<td>-8403</td>
<td>Module name does not exist.</td>
</tr>
<tr>
<td></td>
<td>The module name to be imported was not found.</td>
</tr>
<tr>
<td></td>
<td>Make sure the module name matches the file name.</td>
</tr>
<tr>
<td>-8404</td>
<td>Module name has not been imported.</td>
</tr>
<tr>
<td></td>
<td>A statement is referencing a module name which has not been imported.</td>
</tr>
<tr>
<td></td>
<td>Import the module before usage.</td>
</tr>
<tr>
<td>-8405</td>
<td>category-name qualifier-name.symbol-name has not been defined.</td>
</tr>
<tr>
<td></td>
<td>The symbol identified by qualifier-name.symbol-name cannot be found.</td>
</tr>
<tr>
<td></td>
<td>For example, a START REPORT or SUBDIALOG is referencing a report or sub-</td>
</tr>
<tr>
<td></td>
<td>dialog symbol with module prefix, but the symbol is not found in the</td>
</tr>
<tr>
<td></td>
<td>specified module.</td>
</tr>
<tr>
<td></td>
<td>You must import the module defining the referenced symbol.</td>
</tr>
<tr>
<td>-8406</td>
<td>The function 'function-name' has not been defined.</td>
</tr>
<tr>
<td></td>
<td>The function name is referenced in the compiled module, but none of the</td>
</tr>
<tr>
<td></td>
<td>imported modules define that function.</td>
</tr>
<tr>
<td></td>
<td>You must import the module containing the function.</td>
</tr>
<tr>
<td>-8407</td>
<td>The type of the parameter 'param-name' is not an SQL type:</td>
</tr>
<tr>
<td></td>
<td>cannot be inserted into a temporary table used for this report.</td>
</tr>
<tr>
<td></td>
<td>The REPORT parameter name is defined with a BDL type that has no SQL</td>
</tr>
<tr>
<td></td>
<td>equivalent and thus cannot be used to create the temporary table needed</td>
</tr>
<tr>
<td></td>
<td>to sort rows for a two-pass report.</td>
</tr>
<tr>
<td></td>
<td>Define the parameter with an SQL-compatible type (CHAR, VARCHAR, INTEGER,</td>
</tr>
<tr>
<td></td>
<td>DECIMAL, etc).</td>
</tr>
<tr>
<td>-8408</td>
<td>ON ACTION action-name conflicts with ON action-name.</td>
</tr>
<tr>
<td></td>
<td>The dialog block defines conflicting ON ACTION and ON triggers, defining</td>
</tr>
<tr>
<td></td>
<td>the same actions.</td>
</tr>
<tr>
<td></td>
<td>For example, an ON ACTION delete is defined within a dialog block that is</td>
</tr>
<tr>
<td></td>
<td>also defining an ON DELETE trigger.</td>
</tr>
<tr>
<td></td>
<td>Review the dialog actions, if you want to use ON triggers defining actions.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-8409</td>
<td>The action action-name shadows another action with the same name.</td>
</tr>
<tr>
<td></td>
<td>The dialog defines ON ACTION blocks using the same action name at different levels (dialog, sub-dialog and field level). Use different action names when a conflict occurs.</td>
</tr>
<tr>
<td>-8410</td>
<td>The symbol 'symbol-name' is not a DIALOG.</td>
</tr>
<tr>
<td></td>
<td>The symbol referenced is not defined as a DIALOG subdialog block.</td>
</tr>
<tr>
<td>-8411</td>
<td>Char constant too long.</td>
</tr>
<tr>
<td></td>
<td>The code defines a character string constant that is too long.</td>
</tr>
<tr>
<td>-8412</td>
<td>The statement id 'statement-name' has already been used.</td>
</tr>
<tr>
<td></td>
<td>Several PREPARE instructions use the same statement identifier in the current module. This feature is supported for backward compatibility; however, it is recommended that you review the code to make sure that this is not a mistake.</td>
</tr>
<tr>
<td>-8413</td>
<td>Illegal type attribute value &quot;wrong-value&quot;, expecting correct-value.</td>
</tr>
<tr>
<td></td>
<td>A program variable is defined with an invalid attribute value. Review the ATTRIBUTES() clause of the DEFINE instruction.</td>
</tr>
<tr>
<td>-8415</td>
<td>Invalid number of return values.</td>
</tr>
<tr>
<td></td>
<td>The function uses a RETURN instruction with a number of values that does not correspond to the number of types defined with the RETURNS clause.</td>
</tr>
<tr>
<td>-8416</td>
<td>Missing RETURN.</td>
</tr>
<tr>
<td></td>
<td>The function is defined with a RETURNS clause, but does not contain a RETURN instruction.</td>
</tr>
<tr>
<td>-8417</td>
<td>Only functions can return values.</td>
</tr>
<tr>
<td></td>
<td>It is not allowed to use a RETURN instruction in a MAIN or REPORT block. Use EXIT PROGRAM in MAIN, or EXIT REPORT in REPORT.</td>
</tr>
<tr>
<td>-8418</td>
<td>Cannot call non-FUNCTION type type-name.</td>
</tr>
<tr>
<td></td>
<td>The CALL instruction is used with a program variable which is not a function reference.</td>
</tr>
<tr>
<td>-8419</td>
<td>Return type of FUNCTION function-name is unknown.</td>
</tr>
<tr>
<td></td>
<td>A function referenced with the FUNCTION expression is missing the RETURNS clause in its definition.</td>
</tr>
<tr>
<td>-8420</td>
<td>Illegal argument name 'parameter-name', expecting 'definition-name'.</td>
</tr>
<tr>
<td></td>
<td>A function call is using a parameter name that does not match the name specified in the function definition.</td>
</tr>
<tr>
<td>-8421</td>
<td>RECORD initializer expected.</td>
</tr>
<tr>
<td></td>
<td>The provided initialization value does not use the ( ) notation for RECORD variable definition.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| -8422  | ARRAY initializer expected.  
The provided initialization value does not use the [ ] notation for an ARRAY variable definition. |
| -8423  | Scalar initializer expected.  
The provided initialization value does not use a scalar value for the primitive-typed variable definition. |
| -8424  | Record member 'name' not exists.  
The provided initialization value specifies a member name that does not match the members of the RECORD variable definition. |
| -8425  | Duplicate field in RECORD initializer: 'name'.  
The provided initialization value defines duplicate RECORD member names. |
| -8426  | Imported symbol 'method-name' can not be qualified in a macro.  
Methods for types must be defined in the module where the TYPE is defined. |
| -8427  | >Imported symbol 'symbol-name' can not be qualified in an include file.  
An file included by the preprocessor &include instruction is referencing a symbol defined in an imported module. This practice is not recommended, reduce the dependencies between imported symbols and code produced by the preprocessor code. Move the code from included file into the imported module. |
| -8428  | A method definition requires a RECORD type defined locally in this module.  
A method is defined in the module for a RECORD type that is not defined in this module, or with a type that is not a RECORD. RECORD types and methods for this type must be defined in the same module. |
| -8429  | Method and field names must distinguish. Type type-name has field filed-name.  
A method for a type cannot use the same name as one of the fields of the RECORD type. |
| -8430  | The method method-name has already been defined.  
A method for type is defined twice in the module or in an interface definition. Rename one of the methods if it's not duplicated code, or remove one occurrence of the method listed in the interface. |
| -8431  | Method parameters or return values can not have anonymous types.  
A method defines a parameter with a RECORD, DYNAMIC ARRAY, DICTIONARY, etc, that defines an implicit anonymous type. This is not allowed in methods. First define a user type with the TYPE instruction, then use this type for the method parameter(s). |
| -8432  | An INOUT parameter requires a RECORD type.  
The INOUT keyword can only be used with a type name declared by TYPE definition. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-8433</td>
<td>A method without return type (RETURNS) can not return values. The method body contains a RETURN instruction with value(s), but the declaration does not specify any RETURNS clause.</td>
</tr>
<tr>
<td>-8434</td>
<td>type-name does not implement interface-name: reason. The user-defined type does not match the specified interface for the given reason.</td>
</tr>
<tr>
<td>-8435</td>
<td>Parameter names can not be specified here. The parameter names of the called function are not known. A function call uses named parameters, but the actual function definition is not known by the compiler. Use IMPORT FGL with the module name implementing the function, or do not use named parameters.</td>
</tr>
<tr>
<td>-8436</td>
<td>Can not initialize a list of variables. A DEFINE statement specifies a list of variables followed by an = initializer. This is not allowed, split the code into several DEFINE statements.</td>
</tr>
<tr>
<td>-8437</td>
<td>Anonymous type interface: not supported. A DEFINE statement declares variable(s) with the INTERFACE keyword, that would create an interface without a name. In order to declare a variable as an interface, a named TYPE must be defined as an INTERFACE and the variable definition must use this type name.</td>
</tr>
<tr>
<td>-8438</td>
<td>Recursive type definition. Successive TYPE statements define types that reference each other recursively. For example, typeA references typeB which in turn references typeA. Review the type definitions to avoid the circular reference.</td>
</tr>
<tr>
<td>-8439</td>
<td>Global variables cannot be initialized with an initialization literal. A GLOBALS block contains a DEFINE statement with a value initializer. This is not allowed. Use PUBLIC variables and IMPORT FGL instead.</td>
</tr>
<tr>
<td>-8500</td>
<td>The Genero Mobile pcode size limit has been reached. <strong>Note:</strong> This error is non-trappable. Contact your vendor for details.</td>
</tr>
<tr>
<td>-8501</td>
<td>Modules compiled with Genero require a Genero license at runtime. <strong>Note:</strong> This error is non-trappable. Contact your vendor for details.</td>
</tr>
<tr>
<td>-9000</td>
<td>Value not allowed for this XML attribute. Remove the value for this attribute or see the &quot;Mapping between simple BDL and XML data types&quot; section.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-9001</td>
<td>Value mandatory for this XML attribute.</td>
</tr>
<tr>
<td></td>
<td>Set a value to the XML attribute. See the &quot;Mapping between simple BDL and</td>
</tr>
<tr>
<td></td>
<td>XML data types&quot; section.</td>
</tr>
<tr>
<td>-9002</td>
<td>Cannot set the XML attribute, because only one XSD attribute is allowed per</td>
</tr>
<tr>
<td></td>
<td>definition. Select the unique appropriate XSD data type.</td>
</tr>
<tr>
<td>-9003</td>
<td>XML Attribute only allowed on a BDL TYPE.</td>
</tr>
<tr>
<td></td>
<td>Remove the XML attribute or change your BDL DEFINE instruction into a BDL</td>
</tr>
<tr>
<td></td>
<td>TYPE definition.</td>
</tr>
<tr>
<td>-9004</td>
<td>XML Attribute is not allowed on a type definition.</td>
</tr>
<tr>
<td></td>
<td>Remove the XML attribute or change your BDL TYPE definition into a BDL DEFINE</td>
</tr>
<tr>
<td></td>
<td>instruction.</td>
</tr>
<tr>
<td>-9005</td>
<td>XML Attribute XSTypeNamespace cannot be set without attribute XSTypeName.</td>
</tr>
<tr>
<td></td>
<td>Add a XSTypeName attribute.</td>
</tr>
<tr>
<td>-9006</td>
<td>XML Attribute is only allowed on a simple data type definition.</td>
</tr>
<tr>
<td></td>
<td>Remove the XML attribute or change your RECORD or ARRAY into a simple BDL</td>
</tr>
<tr>
<td></td>
<td>data type.</td>
</tr>
<tr>
<td>-9007</td>
<td>XML Attribute is only allowed on a BDL RECORD definition.</td>
</tr>
<tr>
<td></td>
<td>Change your BDL variable definition into a RECORD.</td>
</tr>
<tr>
<td>-9008</td>
<td>XML Attribute is only allowed on a one dimensionnal array definition.</td>
</tr>
<tr>
<td></td>
<td>Remove the XML attribute or use a one dimensionnal array.</td>
</tr>
<tr>
<td>-9009</td>
<td>Attributes XMLAttribute, XMLElement, XMLAny and XMLBase are exclusives.</td>
</tr>
<tr>
<td></td>
<td>Choose only one of the above available choices.</td>
</tr>
<tr>
<td>-9010</td>
<td>Attributes XMLChoice, XMLAll, XMLSequence, XMLSimpleContent and XSComplexType</td>
</tr>
<tr>
<td></td>
<td>are exclusives.</td>
</tr>
<tr>
<td></td>
<td>Choose only one of the above available choices.</td>
</tr>
<tr>
<td>-9011</td>
<td>Attribute XSTypeName has been defined twice with the same value XML</td>
</tr>
<tr>
<td></td>
<td>attribute and the same XSTypeNamespace value, but not the same definition.</td>
</tr>
<tr>
<td></td>
<td>Define a unique (XSTypeName,XSTypeNamespace) couple for your program.</td>
</tr>
<tr>
<td>-9012</td>
<td>XMLName or XMLNamespace not allowed on nested XMLChoice variable.</td>
</tr>
<tr>
<td></td>
<td>Remove the XMLName and XMLNamespace attributes.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>-9013</td>
<td>XMLName or XMLNamespace not allowed on nested XMLSequence variable. Remove the XMLName and XMLNamespace attributes.</td>
</tr>
<tr>
<td>-9014</td>
<td>Unrecognized XML attribute value. Review the available values for this XML attribute.</td>
</tr>
<tr>
<td>-9015</td>
<td>XML Attribute is only supported on a member of a record. Remove the XML attribute.</td>
</tr>
<tr>
<td>-9016</td>
<td>XML Attribute is only supported on a record's member when XMLChoice is defined. Remove the XML attribute.</td>
</tr>
<tr>
<td>-9017</td>
<td>XML Attribute is only supported on a record's member when XMLSimpleContent is defined. Remove the XML attribute.</td>
</tr>
<tr>
<td>-9018</td>
<td>XML Attribute not supported on this simple type. Remove the XML attribute or change your BDL type definition.</td>
</tr>
<tr>
<td>-9019</td>
<td>Attribute XMLTypeNamespace cannot be set without attribute XMLTypeName. Set XMLTypeName attribute.</td>
</tr>
<tr>
<td>-9020</td>
<td>XMLSimpleContent attribute supports only XMLAttribute and XMLAnyAttribute attributes. Remove the unallowed XML attributes.</td>
</tr>
<tr>
<td>-9021</td>
<td>Attribute XMLBase has been defined more than once in the BDL record. Set only one XMLBase attribute.</td>
</tr>
<tr>
<td>-9022</td>
<td>Attribute XMLSelector has been defined more than once in the BDL record. Set only one XMLSelector attribute.</td>
</tr>
<tr>
<td>-9023</td>
<td>XML Attribute cannot be set with other attributes. Remove all the other XML attributes.</td>
</tr>
<tr>
<td>-9024</td>
<td>Attribute XMLSelector is missing in the BDL record. Set the XMLSelector attribute on one of the record member.</td>
</tr>
<tr>
<td>-9025</td>
<td>Attribute XMLBase is missing in the BDL record. Set the XMLBase attribute on one of the record member.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>-9026</td>
<td>Nested XML attribute cannot be defined on a BDL TYPE. Remove the Nested XML attribute.</td>
</tr>
<tr>
<td>-9027</td>
<td>Nested XML attribute cannot be defined on root variable. Remove the Nested XML attribute.</td>
</tr>
<tr>
<td>-9028</td>
<td>Invalid parameter. See the documentation about the function parameters.</td>
</tr>
<tr>
<td>-9029</td>
<td>Parameters of a published RPC Web Service operation must be a Record or NULL. Review your parameters definition.</td>
</tr>
<tr>
<td>-9030</td>
<td>Parameters of a published DOC Web Service operation must be a Record, an Array or NULL. Review your parameters definition.</td>
</tr>
<tr>
<td>-9031</td>
<td>XML Attribute is not allowed on a BDL record's member. Remove the XML attribute or set it at the appropriate place.</td>
</tr>
<tr>
<td>-9032</td>
<td>XML Attribute can only be set on a ARRAY defined inside a RECORD. Remove the XML attribute or set it at the appropriate place.</td>
</tr>
<tr>
<td>-9033</td>
<td>XML Attribute cannot be defined at first level of a variable. Remove the XML attribute or set it at the appropriate place.</td>
</tr>
<tr>
<td>-9034</td>
<td>Attributes 'XMLAttribute' are not allowed on nested sequence or choice. Remove the XMLAttribute attribute.</td>
</tr>
<tr>
<td>-9035</td>
<td>RPC Web Functions cannot have XMLList set on one of the parameters. Put your BDL ARRAY inside a BDL RECORD.</td>
</tr>
<tr>
<td>-9036</td>
<td>Attribute XMLName is mandatory on BDL variable when used as SOAP Header. Add the XMLName attribute.</td>
</tr>
<tr>
<td>-9037</td>
<td>RPC Web Functions cannot have XMLNamespace set on one of the parameters. Remove the XMLNamespace attribute.</td>
</tr>
<tr>
<td>-9038</td>
<td>XSComplexType attribute allows only attributes with one optional nested list or nested record. Set only one XMLOptional attribute for all nested record members.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>-9039</td>
<td>XMLName or XMLNamespace not allowed on nested XMLAll. Remove XMLName and XMLNamespace.</td>
</tr>
<tr>
<td>-9040</td>
<td>Nested XML Attribute is not allowed on an array. Remove the XML attribute.</td>
</tr>
<tr>
<td>-9041</td>
<td>XMLBase Attribute allows only one additional XSD attribute. Set a unique XSD attribute.</td>
</tr>
<tr>
<td>-9042</td>
<td>XML Attribute value is not allowed on a BDL record's member. Set the appropriate value to the specified XML attribute.</td>
</tr>
<tr>
<td>-9043</td>
<td>Unsupported facet constraint for the BDL type. Check the available facet constraint in &quot;Mapping between simple BDL and XML data types&quot; section.</td>
</tr>
<tr>
<td>-9044</td>
<td>Invalid value for facet constraint 'constraint-name'. Check the available facet constraint value. See XML facet constraint attributes on page 3534.</td>
</tr>
<tr>
<td>-9045</td>
<td>Facet constraint attributes cannot be defined without a XSD simple type attribute. Add the appropriate XSD attribute.</td>
</tr>
<tr>
<td>-9046</td>
<td>Facet XSDLength and XSDMinLength or XSDMaxLength cannot be used together. Select only one of the above attributes.</td>
</tr>
<tr>
<td>-9047</td>
<td>XML Attribute not allowed on BDL objects. Remove the XML attribute.</td>
</tr>
<tr>
<td>-9048</td>
<td>Attribute XMLName cannot be set with XMLAny or XMLAnyAttribute. Remove the XMLName attribute.</td>
</tr>
<tr>
<td>-9049</td>
<td>XML Attribute not allowed on members of xmlchoice='inherited' records. Remove the XML attribute.</td>
</tr>
<tr>
<td>-9050</td>
<td>Parameter with public qualifier not allowed. Remove the PUBLIC instruction.</td>
</tr>
<tr>
<td>-9051</td>
<td>Parameters of published Web Service operations must be variables in global or modular scope. Move your variables to a GLOBALS instruction or to modular scope.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| -9052  | A published Web service header must be a variable in global or modular scope.  
Move your Web service header to a GLOBALS instruction or to modular scope. |
| -9053  | Web service function with private qualifier not allowed.  
Remove the PRIVATE instruction. |
| -9054  | Web service function must be a string literal.  
You cannot use a variable for your web service function name. |
| -9055  | XML Attribute is not allowed on an array definition.  
Remove the XML attribute. |
| -9056  | Attribute XMLAny has been defined more than once per BDL record.  
Use only one XMLAny attribute in a BDL RECORD. |
| -9057  | Attribute XMLAnyAttribute has been defined more than once per BDL record.  
Use only one XMLAnyAttribute attribute in a BDL RECORD. |
| -9058  | Attribute XMLList and XMLAnyAttribute are exclusives.  
Use only one of the above XML attributes. |
| -9059  | Element of BDL array with XMLAnyAttribute must be a BDL record containing three variables for the namespace, name, value of type STRING.  
Example: DEFINE arr DYNAMIC ARRAY OF RECORD ns, name, value STRING END RECORD |
| -9060  | XML Attribute is only allowed on dynamic arrays.  
Change your BDL ARRAY into a DYNAMIC ARRAY. |
| -9061  | XML Attribute cannot be set inside a nested record.  
Remove the XML attribute. |
| -9062  | Attribute XMLAttribute is not allowed after attribute XMLAnyAttribute.  
Move the record member with XMLAnyAttribute attribute to the last position. |
| -9063  | A published Web service fault must be in global or modular scope.  
Move your variables to a GLOBALS instruction or to modular scope. |
| -9064  | Attribute XMLName is mandatory on the BDL variable when used as Fault.  
Set the XMLName attribute. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-9065</td>
<td>Colon not allowed for XML attribute value. Remove the colon.</td>
</tr>
<tr>
<td>-9066</td>
<td>XML Attribute is only allowed on a root variable. Remove the attribute or move it to the root variable.</td>
</tr>
<tr>
<td>-9067</td>
<td>Bad W3CEndPointReference definition. Review your RECORD definition. It needs to match this structure:</td>
</tr>
<tr>
<td></td>
<td>RECORD ATTRIBUTES(W3CEndPointReference)</td>
</tr>
<tr>
<td></td>
<td>address STRING, -- The location of the Web Service (for ex: URL)</td>
</tr>
<tr>
<td></td>
<td>ref RECORD</td>
</tr>
<tr>
<td></td>
<td>... (other members defining the state)</td>
</tr>
<tr>
<td></td>
<td>END RECORD</td>
</tr>
<tr>
<td></td>
<td>END RECORD</td>
</tr>
<tr>
<td>-9068</td>
<td>Invalid state BDL variable, only simple variables or W3CEndpointReference record allowed. Check that &quot;state&quot; parameter TYPE of function com.WebService.CreateStatefulWebService is correct. Its type must be a simple type definition or a W3CEndPointReference RECORD.</td>
</tr>
<tr>
<td>-9069</td>
<td>Registered HTTP variable error. Check that the BDL variable match the definition set in com.WebService.registerInputHTTPVariable or com.WebService.registerOutputHTTPVariable.</td>
</tr>
<tr>
<td>-9070</td>
<td>Attribute name only allowed on XML elements. Check that attribute names are specified for types that are XMLElement (Optional) on page 3541 elements.</td>
</tr>
<tr>
<td>-9100</td>
<td>Rest verb is missing. Function is missing a REST attribute, for example, WSGet, WSPut, WSPost, or WSDelete. Check the ATTRIBUTES() clause of your function.</td>
</tr>
<tr>
<td>-9101</td>
<td>Only one rest verb allowed. Function must only have one REST attribute of this type, for example, WSGet, WSPut, WSPost, or WSDelete, etc. Check the ATTRIBUTES() clause of your function.</td>
</tr>
<tr>
<td>-9102</td>
<td>Rest operation cannot be private. Check that the REST function is declared as public.</td>
</tr>
<tr>
<td>-9103</td>
<td>Attribute name only allowed as parameter of a rest function. Check that WSParam, WQuery, WHeader, WSCookie, or WOptional Rest attributes are set on the input parameter of a REST function.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| -9104  | Invalid WSPath template value : value.  
Template value in WSPath is incorrect. Check the punctuation, for instance, the ending curly bracket is missing. |
| -9105  | Some WSPath template values are missing : value.  
Add parameters with WSParam attribute to match the missing template values. |
| -9106  | name operation requires input parameter body.  
If you use WSPost, WSPut, or WSPatch as REST resource operation, the FUNCTION must have at least one REST input body. |
| -9107  | Rest parameter name is not exclusive.  
Check that only one of the following is set: WSParam, WSQuery, WSHandler, WSCookie on a Rest function parameter |
| -9108  | Attribute name not allowed with WSParam.  
WSOOptional and WSName cannot be set when WSParam is set. |
| -9109  | Attribute name only allowed on Rest input or output function parameter.  
Check that WSParam, WSQuery, WSHandler, WSCookie, or WSOOptional Rest attributes are set on the output parameter of a REST function. |
| -9110  | WSPath must start with a slash : value  
Add a slash to the WSPath attribute value. |
| -9111  | WSParam 'name' does not match any of the WSPath template value.  
Add the template value to WSPath for the missing WSParam attribute. |
| -9112  | Attribute 'name' is not allowed as element of an array.  
WSParam, WSQuery, WSHandler, WSCookie, or WSOOptional cannot be set on an array |
| -9113  | Attribute 'name' only allowed on function definition.  
WSGet, WSPost, WSPut, WSDelete, WSDescription, WSThrows, and WSRetCode can only be set as function attributes. |
| -9114  | WSMedia value 'value' only allowed on simple datatype definition.  
One of the WSMedia value cannot be set on an array |
| -9115  | WSMedia value 'value' not allowed on this datatype definition.  
One of the WSMedia value is not valid for that variable. For instance, you cannot set 'text/plain' on a DATETIME. |
| -9116  | Malformed WSThrows value : value  
The WSThrows attribute has invalid syntax.  
Look for missing or extra punctuation; missing quotes, missing commas, or spaces after the colon (:) , or a reference to a modular variable that does not exist. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-9117</td>
<td>Invalid HTTP code in WSThrows value : code</td>
</tr>
<tr>
<td></td>
<td>This HTTP code is not allowed in WSThrows.</td>
</tr>
<tr>
<td></td>
<td>Check to make sure you are using HTTP error codes between 400 to 599 in your WSThrows attribute.</td>
</tr>
<tr>
<td>-9118</td>
<td>Invalid error variable in WSThrows value : variable</td>
</tr>
<tr>
<td></td>
<td>WSThrows attribute is referencing a variable incorrectly.</td>
</tr>
<tr>
<td></td>
<td>Check that WSThrows is referencing a public modular variable.</td>
</tr>
<tr>
<td>-9119</td>
<td>Invalid description in WSThrows value : description</td>
</tr>
<tr>
<td></td>
<td>Check that there is no blank in the description of a WSThrows value.</td>
</tr>
<tr>
<td>-9120</td>
<td>Malformed WSRetCode value : value</td>
</tr>
<tr>
<td></td>
<td>Check the WSRetCode attribute syntax and correct the punctuation.</td>
</tr>
<tr>
<td>-9121</td>
<td>Invalid HTTP code in WSRetCode : code</td>
</tr>
<tr>
<td></td>
<td>Check that the HTTP code is in the range 200 to 399</td>
</tr>
<tr>
<td>-9122</td>
<td>WSScope only allowed with WSInfo.</td>
</tr>
<tr>
<td></td>
<td>Check that the WSScope attribute is set on the WSInfo record</td>
</tr>
<tr>
<td>-9123</td>
<td>Only one 'name' allowed per module.</td>
</tr>
<tr>
<td></td>
<td>You can only set one WSContext or WSInfo per module.</td>
</tr>
<tr>
<td>-9124</td>
<td>Attribute 'name' is only allowed on a dictionary definition.</td>
</tr>
<tr>
<td></td>
<td>WSContext can only be set on a DICTIONARY</td>
</tr>
<tr>
<td>-9125</td>
<td>Attribute 'name' must be defined on private variable.</td>
</tr>
<tr>
<td></td>
<td>Attribute WSContext must be defined on a private modular dictionary</td>
</tr>
<tr>
<td>-9126</td>
<td>Attribute 'name' must be defined on public variable.</td>
</tr>
<tr>
<td></td>
<td>Attribute WSInfo must be defined on a public modular variable</td>
</tr>
<tr>
<td>-9127</td>
<td>WSPath value 'value' already encountered for same HTTP verb.</td>
</tr>
<tr>
<td></td>
<td>The same HTTP verb cannot be used more than once with the same resource path. Look for another FUNCTION using the same verb (identified by the attribute with WS* prefix) with the same WSPath. Make sure the paths have different values.</td>
</tr>
<tr>
<td>-9130</td>
<td>Attribute 'name' is not allowed on a function definition.</td>
</tr>
<tr>
<td></td>
<td>The FUNCTION definition contains and WS* attribute that cannot be used in this context. Remove the attribute from the function definition.</td>
</tr>
<tr>
<td>-10098</td>
<td>Incorrectly formed hexadecimal value.</td>
</tr>
<tr>
<td></td>
<td>You try to load data with LOAD or locate a BYTE variable with a file containing malformed hexadecimal values.</td>
</tr>
<tr>
<td></td>
<td>Check the file content and fix the typos before loading again.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-10099</td>
<td>Invalid delimiter. Do not use '' or hex digits (0-9, A-F, a-f). You try to LOAD or UNLOAD data with an invalid field delimiter. Change the field delimiter to a valid character such as</td>
</tr>
<tr>
<td>-15500</td>
<td>Internal runtime error occurred in WS server program. Contact your support center.</td>
</tr>
<tr>
<td>-15501</td>
<td>Cannot create WS operation because the given function is not defined. Verify that the name of the FUNCTION of fgl_ws_server_publishFunction() is correct.</td>
</tr>
<tr>
<td>-15502</td>
<td>Invalid WS-function declaration, no parameters allowed. Verify that the FUNCTION has no input and no output parameters.</td>
</tr>
<tr>
<td>-15503</td>
<td>Operation name is already used in the current web service. You must change the name of the Web-Function operation in the function fgl_ws_server_publishFunction().</td>
</tr>
<tr>
<td>-15504</td>
<td>WS server port already used by another application. You must change the port number in the function Fgl_ws_server_start().</td>
</tr>
<tr>
<td>-15505</td>
<td>Some BDL data types are not supported by XML. Verify that all exposed functions don't contain one of the following data types: • DATETIME beginning with MINUTE • DATETIME beginning with SECOND • INTERVAL beginning with YEAR and/or MONTH</td>
</tr>
<tr>
<td>-15511</td>
<td>Invalid fgl_ws_set/getOption() parameter. Verify that the option flag of the fgl_ws_setOption() or fgl_ws_getOption() function exists.</td>
</tr>
<tr>
<td>-15512</td>
<td>WS input record not defined. Verify that the name of the input record on the fgl_ws_server_publishFunction() exists.</td>
</tr>
<tr>
<td>-15513</td>
<td>WS output record not defined. Verify that the name of the output record on the fgl_ws_server_publishFunction() exists.</td>
</tr>
<tr>
<td>-15514</td>
<td>The port value from the FGLAPPSERVER environment variable or from the parameter of the fgl_ws_server_start() function is not a numeric one. Verify that the port value contains only digits. See fgl_ws_server_start()</td>
</tr>
<tr>
<td>-15515</td>
<td>No application server has been started at specified host. Verify that FGLAPPSERVER contains the right host and port where the application server is listening.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| -15516 | No more licenses available.  
Contact your support center. |
| -15517 | Current runner version not compatible with the Web Services Extension.  
Install the right version of the Genero BDL. |
| -15518 | The input namespace of your Web function is missing.  
Add a valid input namespace in `fgl_ws_server_publishFunction()` |
| -15519 | The output namespace >namespace of your Web function is missing.  
Add a valid output namespace in `fgl_ws_server_publishFunction()` |
| -15520 | Cannot load a certificate or private key file.  
Verify that each `ws.idws.security` FGLPROFILE entry contains a valid security identifier. |
| -15521 | Cannot find a certificate in the Windows key store.  
Verify that each `ws.idws.security` FGLPROFILE entry contains a valid Windows® security identifier. |
| -15522 | Cannot load the Certificate Authorities file.  
Verify that the `security.global.ca` FGLPROFILE entry contains the correct Certificate Authorities file name. |
| -15523 | Cannot create the Certificate Authorities from the Windows key store.  
Verify that you have enough rights to access the Windows® key store. |
| -15524 | Cannot set the cipher list.  
Verify that all ciphers in the list are valid ones and supported by openssl. |
| -15525 | Unable to reach the HTTP proxy.  
Verify that the `proxy.http.location` FGLPROFILE entry contains the correct HTTP proxy address. |
| -15526 | Unable to reach the HTTPS proxy.  
Verify that the `proxy.https.location` FGLPROFILE entry contains the correct HTTPS proxy address. |
| -15527 | Unknown HTTP proxy authenticate identifier.  
Verify that the `proxy.http.authenticate` FGLPROFILE entry contains a valid HTTP authenticate identifier. |
| -15528 | Unknown HTTPS proxy authenticate identifier.  
Verify that the `proxy.https.authenticate` FGLPROFILE entry contains a valid HTTP authenticate identifier. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-15529</td>
<td>Cannot create a HTTP authenticate configuration. Verify that all authenticate logins and passwords are correctly set.</td>
</tr>
<tr>
<td>-15530</td>
<td>Cannot create an encrypted HTTP authenticate configuration. Verify that all authenticate logins and encrypted passwords are correctly set.</td>
</tr>
<tr>
<td>-15531</td>
<td>Cannot create a server configuration. Verify that all ws.ident.url FGLPROFILE entries are correctly set.</td>
</tr>
<tr>
<td>-15532</td>
<td>Unknown server configuration security identifier. Verify that all ws.ident.security FGLPROFILE entries contain a valid Security identifier.</td>
</tr>
<tr>
<td>-15533</td>
<td>Unknown server configuration authenticate identifier. Verify that all ws.ident.authenticate FGLPROFILE entries contain a valid HTTP Authenticate identifier.</td>
</tr>
<tr>
<td>-15534</td>
<td>Invalid self object. Contact your support center.</td>
</tr>
<tr>
<td>-15535</td>
<td>Cannot perform operation due to invalid parameters. Check all parameters against the built-in classes documentation.</td>
</tr>
<tr>
<td>-15536</td>
<td>Service registration failed, see SQLCA.SQLERRM for more details. Check the following:</td>
</tr>
<tr>
<td></td>
<td>• A service of the same name already exists</td>
</tr>
<tr>
<td></td>
<td>• The namespace of the service is missing</td>
</tr>
<tr>
<td></td>
<td>• A header cannot have the same name and namespaces as an operation</td>
</tr>
<tr>
<td>-15537</td>
<td>Cannot create web service, see SQLCA.SQLERRM for more details.</td>
</tr>
<tr>
<td></td>
<td>Check that the service has a valid name and namespace.</td>
</tr>
<tr>
<td>-15538</td>
<td>Cannot create Web operation, see SQLCA.SQLERRM for more details.</td>
</tr>
<tr>
<td></td>
<td>Check that operation name and namespace are valid based on the style (Document or RPC).</td>
</tr>
<tr>
<td>-15539</td>
<td>Cannot publish Web operation, see SQLCA.SQLERRM for more details.</td>
</tr>
<tr>
<td></td>
<td>Check that input or output headers have previously been created.</td>
</tr>
<tr>
<td>-15540</td>
<td>Published BDL function not found, see SQLCA.SQLERRM for more details.</td>
</tr>
<tr>
<td></td>
<td>Check that the FUNCTION to be published exists.</td>
</tr>
<tr>
<td>-15541</td>
<td>Published BDL function not correctly defined, see SQLCA.SQLERRM for more details.</td>
</tr>
<tr>
<td></td>
<td>Check that the FUNCTION has no input or output parameters.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-15542</td>
<td>Input parameter of published operation error. See SQLCA.SQLERRM for more details. Contact your support center.</td>
</tr>
<tr>
<td>-15543</td>
<td>Output parameter of published operation error. See SQLCA.SQLERRM for more details. Contact your support center.</td>
</tr>
<tr>
<td>-15544</td>
<td>Web Service header configuration error, see SQLCA.SQLERRM for more details. Verify that a one-way function do not have an output header.</td>
</tr>
<tr>
<td>-15545</td>
<td>Service is already registered. You cannot modify a service after it has been registered. Check that you do not call a service modifier method on a service after registration.</td>
</tr>
<tr>
<td>-15546</td>
<td>Invalid option. Check the option name according to documentation.</td>
</tr>
<tr>
<td>-15547</td>
<td>Unsupported web service operation. Verify if a Document style operation does not perform SOAP Section5 encoding.</td>
</tr>
<tr>
<td>-15548</td>
<td>Bad URI. Check that URI passed to a HttpRequest or TcpRequest is valid.</td>
</tr>
<tr>
<td>-15549</td>
<td>HTTP runtime exception, see SQLCA.SQLERRM for more details. Contact your support center.</td>
</tr>
<tr>
<td>-15550</td>
<td>XML runtime exception, see SQLCA.SQLERRM for more details. Contact your support center.</td>
</tr>
<tr>
<td>-15551</td>
<td>WSDL generation failed. Contact your support center.</td>
</tr>
<tr>
<td>-15552</td>
<td>Charset conversion exception, see SQLCA.SQLERRM for more details. Change server charset response via a HTTP accept header or change you application locale.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| -15553 | TCP runtime exception, see SQLCA.SQLERRM for more details. If detailed message is 'The TCP connection has been interrupted', then check that your network was working properly and that the INT_FLAG was not set to TRUE. When working with a Web Service application, this can be the result of a COM error. Check in FGLWSDEBUG to see whether it was shut down on the client or server side. For example:  
    WS-DEBUG (IO ERROR)  
    Class: TCPConection::atomicReceive()  
    Msg: TCP input stream shut down.  
    Code: 104  
    WS-DEBUG END=  
    You can find the 104 code in /usr/include/asm-i386/errno.h (depending on your system). In this example it corresponds to: #define ECONNRESET 104 /* Connection reset by peer */  
    • Review the WSDL and see if what we send to the server is correct  
    • Review the server log and see why it has ended the connection |
<p>| -15554 | Index is out of bound. Check your index maximum value. |
| -15555 | Unsupported request-response feature. Check the streaming operations order or for invalid usage. For example, in function readTextRequest(), the incoming request can be read only once, so processing the incoming message while sending the response is not allowed. |
| -15556 | No request was sent. Check that you called one of the doRequest(), doXmlRequest() or doTextRequest() method before to call getResponse() or getAsyncResponse(). |
| -15557 | Request was already sent. Check that you do not call twice one of the doRequest(), doXmlRequest() or doTextRequest() methods. |
| -15558 | Waiting for a response. Check that you do not perform a new request before reading the response of previous one. |
| -15559 | No stream available. Check that you do not call a method to read on a stream that has not yet been created. |
| -15560 | Streaming is over. Check that you do not read a streaming response that was closed. |
| -15561 | Streaming in progress. Check that you do not call twice beginXmlResponse() without a call to endXmlResponse(). |</p>
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
</table>
| -15562 | Streaming not yet started.  
Check that you do not call endXmlRequest() or endXmlResponse() without a beginXmlRequest() or beginXmlResponse(). |
| -15563 | Streaming already started.  
Check that you do not call twice beginXmlRequest() or beginXmlResponse(). |
| -15564 | Unexpected peer stream was shutdown.  
The peer closed connection during reading operation. |
| -15565 | Cannot return incoming request, see SQLCA.SQLERRM for more details.  
Check detailed message. |
| -15566 | Operation failed, see SQLCA.SQLERRM for more details.  
Check the parameter for invalid data. |
| -15567 | Parameter cannot be NULL.  
Check that the parameter is not NULL. |
| -15568 | BDL callback function not found, see SQLCA.SQLERRM for more details.  
Check that BDL callback function exists. |
| -15569 | BDL callback function requires one input and one output parameter, see SQLCA.SQLERRM for more details.  
Check BDL callback parameters according to documentation. |
| -15570 | Web Service fault error. See SQLCA.SQLERRM for more detail.  
A method of the COM class has failed, for example, the variable passed to com.WebServiceEngine.SetRestError() is not the expected one. Contact your support center if the error detail does not provide the information needed to fix the error. |
| -15571 | Stateful Service error. See SQLCA.SQLERRM for more detail.  
Contact your support center. |
| -15572 | Access denied lock error.  
Either the file is already locked, or the application does not have the write access right to the given path. |
| -15573 | HTTP Multipart error : description.  
One of the methods of the COM multipart API has failed. See the description for more details. Contact your support center if the error detail does not provide the information needed to fix the error. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-15574</td>
<td>Cannot load Certificate Authorities from path : path.</td>
</tr>
<tr>
<td></td>
<td>The certificate defined by the current FGLPROFILE configuration was not found.</td>
</tr>
<tr>
<td></td>
<td>Check the certificate authority settings as described in: HTTPS configuration on page 3305.</td>
</tr>
<tr>
<td>-15575</td>
<td>Incoming request has been closed : reason</td>
</tr>
<tr>
<td></td>
<td>The GAS has disconnected the web service server, for example while calling the com.WebServiceEngine.GetHTTPServiceRequest or com.WebServiceEngine.HandleRequest methods.</td>
</tr>
<tr>
<td></td>
<td>Use a TRY/CATCH block to trap this error, as described in com.WebServiceEngine.GetHTTPServiceRequest on page 2832.</td>
</tr>
<tr>
<td>-15576</td>
<td>Invalid TCP IP version.</td>
</tr>
<tr>
<td></td>
<td>The FGLPROFILE configuration parameter ip.global.version defines a value different from valid possible values (4 and 6).</td>
</tr>
<tr>
<td>-15577</td>
<td>Unknown network interface name : name.</td>
</tr>
<tr>
<td></td>
<td>The FGLPROFILE configuration parameter ip.global.v6.interface.name defines a network interface that does not exist.</td>
</tr>
<tr>
<td>-15578</td>
<td>Request canceled by user.</td>
</tr>
<tr>
<td></td>
<td>The HTTP request initiated by a com.HTTPRequest.getResponse() method has been canceled by the user.</td>
</tr>
<tr>
<td>-15579</td>
<td>Proxy unreachable</td>
</tr>
<tr>
<td></td>
<td>The proxy URL is unreachable.</td>
</tr>
<tr>
<td></td>
<td>Check to verify that the host and port are set correctly, and that the proxy is online and available.</td>
</tr>
<tr>
<td>-15598</td>
<td>XML deserialization error.</td>
</tr>
<tr>
<td></td>
<td>The WSDL contract does not match the BDL variable definition.</td>
</tr>
<tr>
<td></td>
<td>Check that BDL variables are correctly generated according to the WSDL.</td>
</tr>
<tr>
<td>-15599</td>
<td>Internal error, should not happen.</td>
</tr>
<tr>
<td></td>
<td>Contact your support center.</td>
</tr>
<tr>
<td>-15600</td>
<td>Operation failed.</td>
</tr>
<tr>
<td></td>
<td>Check method for invalid parameters according to documentation.</td>
</tr>
<tr>
<td>-15601</td>
<td>Name cannot be NULL.</td>
</tr>
<tr>
<td></td>
<td>Check that name parameter is not NULL.</td>
</tr>
<tr>
<td>-15602</td>
<td>Namespace cannot be NULL.</td>
</tr>
<tr>
<td></td>
<td>Check that namespace parameter is not NULL.</td>
</tr>
<tr>
<td>-15603</td>
<td>Prefix cannot be NULL.</td>
</tr>
<tr>
<td></td>
<td>Check that prefix parameter is not NULL.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-15604</td>
<td>Value cannot be NULL. Check that parameter is not NULL according to documentation.</td>
</tr>
<tr>
<td>-15605</td>
<td>Node cannot be NULL. Check that node parameter is not NULL.</td>
</tr>
<tr>
<td>-15606</td>
<td>Text cannot be NULL. Check that text parameter is not NULL.</td>
</tr>
<tr>
<td>-15607</td>
<td>Target of a processing instruction cannot be NULL. Check that target parameter is not NULL.</td>
</tr>
<tr>
<td>-15608</td>
<td>Name of an entity reference cannot be NULL. Check that entity name parameter is not NULL.</td>
</tr>
<tr>
<td>-15609</td>
<td>XPath expression cannot be NULL. Check that xpath parameter is not NULL.</td>
</tr>
<tr>
<td>-15610</td>
<td>Filename cannot be NULL. Check that filename parameter is not NULL.</td>
</tr>
<tr>
<td>-15611</td>
<td>Document cannot be NULL. Check that document parameter is not NULL.</td>
</tr>
<tr>
<td>-15612</td>
<td>DTD string cannot be NULL. Check that dtd parameter is not NULL.</td>
</tr>
<tr>
<td>-15613</td>
<td>Stax cannot be NULL. Check that stax parameter is not NULL.</td>
</tr>
<tr>
<td>-15614</td>
<td>Malformed XML name. Check that xml name is well-formed.</td>
</tr>
<tr>
<td>-15615</td>
<td>Malformed XML string. Check that xml string is well-formed.</td>
</tr>
<tr>
<td>-15616</td>
<td>Malformed XML prefix. Check that xml prefix is well-formed.</td>
</tr>
<tr>
<td>-15617</td>
<td>Malformed XML namespace. Check that xml namespace is well-formed.</td>
</tr>
<tr>
<td>-15618</td>
<td>Bad validation type. Check validation type parameter.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| -15619 | No XML schema found.  
Check that a valid XML schema is used for validation. |
| -15620 | No DTD schema found.  
Check that a DTD schema is present in XML document. |
| -15621 | Feature or option cannot be NULL.  
Check that parameters are not NULL. |
| -15622 | Feature or option is unsupported.  
Check option or feature name according to documentation. |
| -15623 | Feature or option value is invalid.  
Check option or feature validity according to documentation. |
| -15624 | Node is not part of the document.  
Check that node belong to the same XML document. |
| -15625 | Node does not have the correct parent node.  
Check that node to remove belongs to the right parent node. |
| -15626 | Node is already linked to another node.  
Check that node is not already attached to another node. |
| -15627 | Cannot add a node to itself.  
Check that node to add is not itself. |
| -15628 | Index is out of bounds.  
Check index maximum value. |
| -15629 | StaxWriter runtime exception: reason  
See SQLCA.SQLERRM for more details and check the reason for the error. |
| -15630 | StaxReader runtime exception: reason  
See SQLCA.SQLERRM for more details and check the reason for the error. |
| -15631 | Serializer runtime exception: reason  
See SQLCA.SQLERRM for more details and check the reason for the error. |
| -15632 | Document loading runtime exception, check  
xml.DomDocument.getErrorDescription() for more details.  
Check detailed message of dom document. |
| -15633 | Document saving runtime exception, check  
xml.DomDocument.getErrorDescription() for more details.  
Check detailed message of dom document. |
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-15634</td>
<td>Invalid encoding.</td>
</tr>
<tr>
<td></td>
<td>Check encoding value.</td>
</tr>
<tr>
<td>-15635</td>
<td>PublicID of a DTD cannot be set with a SystemID.</td>
</tr>
<tr>
<td></td>
<td>Check DTD node creation</td>
</tr>
<tr>
<td>-15636</td>
<td>Undefined namespace prefix in the XPath expression.</td>
</tr>
<tr>
<td></td>
<td>Check an undeclared prefix used in XPath expression.</td>
</tr>
<tr>
<td>-15637</td>
<td>XPath expression error.</td>
</tr>
<tr>
<td></td>
<td>Check XPath expression.</td>
</tr>
<tr>
<td>-15638</td>
<td>A namespace in the XPath namespace list is missing.</td>
</tr>
<tr>
<td></td>
<td>Check for an undeclared namespace used in XPath expression</td>
</tr>
<tr>
<td>-15639</td>
<td>XPath function has two mandatory parameters.</td>
</tr>
<tr>
<td></td>
<td>Check parameters according to documentation.</td>
</tr>
<tr>
<td>-15640</td>
<td>Internal XPath error.</td>
</tr>
<tr>
<td></td>
<td>Contact your support center.</td>
</tr>
<tr>
<td>-15641</td>
<td>Invalid XPath namespace.</td>
</tr>
<tr>
<td></td>
<td>Check namespace value passed to XPath method.</td>
</tr>
<tr>
<td>-15642</td>
<td>Unable to load schema.</td>
</tr>
<tr>
<td></td>
<td>Check XML schema parameters in DomDocument.setFeature().</td>
</tr>
<tr>
<td>-15643</td>
<td>Schemas are malformed or inconsistent.</td>
</tr>
<tr>
<td></td>
<td>Check XML schema validity in DomDocument.setFeature().</td>
</tr>
<tr>
<td>-15644</td>
<td>URI is malformed.</td>
</tr>
<tr>
<td></td>
<td>Check that URI is well-formed according to documentation.</td>
</tr>
<tr>
<td>-15645</td>
<td>Protocol layer needs a new try to complete operation.</td>
</tr>
<tr>
<td></td>
<td>Sax writer close operation requires a new request to complete previous one.</td>
</tr>
<tr>
<td>-15646</td>
<td>Charset conversion error.</td>
</tr>
<tr>
<td></td>
<td>Check fglrun LANG and system locale.</td>
</tr>
<tr>
<td>-15647</td>
<td>Unable to load xml security library.</td>
</tr>
<tr>
<td></td>
<td>Contact your support center.</td>
</tr>
<tr>
<td>-15648</td>
<td>Xml security operation failed. See SQLCA.SQLERRM for more detail.</td>
</tr>
<tr>
<td></td>
<td>Check detailed message.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-15649</td>
<td>URL cannot be null.</td>
</tr>
<tr>
<td></td>
<td>Check if XML-Security URL is NULL.</td>
</tr>
<tr>
<td>-15650</td>
<td>CryptoX509 cannot be null.</td>
</tr>
<tr>
<td></td>
<td>Verify that CryptoX509 object has been correctly instantiated.</td>
</tr>
<tr>
<td>-15651</td>
<td>CryptoKey cannot be null.</td>
</tr>
<tr>
<td></td>
<td>Verify that CryptoKey object has been correctly instantiated.</td>
</tr>
<tr>
<td>-15652</td>
<td>Bad signature transformation.</td>
</tr>
<tr>
<td></td>
<td>Check transformation URL validity passed to appendReferenceTransformation()</td>
</tr>
<tr>
<td>-15653</td>
<td>Bad signature digest.</td>
</tr>
<tr>
<td></td>
<td>Check digest URL validity passed to createReference().</td>
</tr>
<tr>
<td>-15654</td>
<td>Bad signature node.</td>
</tr>
<tr>
<td></td>
<td>Check XML-Signature node passed to CreateFromNode().</td>
</tr>
<tr>
<td>-15655</td>
<td>Bad key type.</td>
</tr>
<tr>
<td></td>
<td>Check key identifier URL.</td>
</tr>
<tr>
<td>-15656</td>
<td>Bad key usage.</td>
</tr>
<tr>
<td></td>
<td>Verify usage of CryptoKey object passed to setKeyEncryptionKey() or setKey().</td>
</tr>
<tr>
<td>-15657</td>
<td>Bad XPathFilter2 type, only intersect, subtract or union allowed.</td>
</tr>
<tr>
<td></td>
<td>Verify type used in a XPathFilter2 transformation.</td>
</tr>
<tr>
<td>-15658</td>
<td>Bad derived key URL.</td>
</tr>
<tr>
<td></td>
<td>Check derived key identifier URL.</td>
</tr>
<tr>
<td>-15699</td>
<td>Internal error, should not happen.</td>
</tr>
<tr>
<td></td>
<td>Contact your support center.</td>
</tr>
<tr>
<td>-15700</td>
<td>Called operation failed, see SQLCA.SQLERRM for more details.</td>
</tr>
<tr>
<td></td>
<td>See SQLCA.SQLERRM for details on why the operation failed.</td>
</tr>
<tr>
<td>-15701</td>
<td>Invalid parameter.</td>
</tr>
<tr>
<td></td>
<td>Check that your security library function has the correct parameters.</td>
</tr>
<tr>
<td>-15702</td>
<td>File access denied.</td>
</tr>
<tr>
<td></td>
<td>Check that your security library function has the permissions to access the file.</td>
</tr>
<tr>
<td>-15703</td>
<td>File does not exist.</td>
</tr>
<tr>
<td></td>
<td>Check that the file exist on your system for the security library function to access.</td>
</tr>
<tr>
<td>Number</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>-15704</td>
<td>Algorithm not supported.</td>
</tr>
<tr>
<td></td>
<td>Check that the algorithm is in the supported list for security library function. See security.Digest.CreateDigest on page 3145.</td>
</tr>
<tr>
<td>-15705</td>
<td>Invalid current object.</td>
</tr>
<tr>
<td></td>
<td>Check that the context for security library function is correctly initialized. See security.Digest.CreateDigest on page 3145.</td>
</tr>
<tr>
<td>-15799</td>
<td>Internal security error.</td>
</tr>
<tr>
<td></td>
<td>Contact your support center.</td>
</tr>
</tbody>
</table>

### Web services

Create a Web service client or server with Genero BDL.

The Genero APIs for creating Web services can be found in the Library section of this manual. See The com package on page 2811, The xml package on page 2931, and The util package on page 2726.

### General

These topics provide you with an introduction to Genero Web Services and the information needed to get working with the latest version of the software.

### Introduction to Web services

Web services are a standard way of communicating between applications over an intranet or Internet.

Web services can be invoked via the HyperText Transfer Protocol (HTTP), by requests for services, and information is returned in either JSON documents or XML documents. It does not matter if the platform that runs the Web Service is different to the platform that receives the JSON or XML document.

Genero Web Services supports Web services implemented by the Simple Object Access Protocol (SOAP) and Representational State Transfer (REST) architecture, which are defined standards for communicating with Web services. A Web service defines how to communicate between two entities:

- A server that exposes services
- A client that consumes services

### Server usage example

A server exposes, for example, a "StockQuotation" service that responds to an operation "getQuote". For the "getQuote" operation, the input message is a stock symbol as a string, and the output message is a stock value as a decimal number.

The "getQuote" operation is a function written in Genero BDL, and it is published on the server. This function retrieves the stock value for the stock symbol passed in, and returns it.

### Client usage example

The Web service client application calls the function as if it were a local function. It passes the stock symbol to the function, and stores the returned value in a variable.
SOAP

If the Web service operation is named `WebService_StockQuotation_getQuote` and the local variable is `svalue`, the Web service is called as follows:

```plaintext
LET svalue = WebService_StockQuotation_getQuote( "MyStockSymbol" )
```

REST

The Web service operation is requested using the URI with the required parameter that matches the function defined in the "StockQuotation" Web service server side. Therefore, `quotes` becomes the resource name part of the URI included in our example code. `req` is defined as an object of the `com.HTTPRequest` class.

```plaintext
```

Related concepts

REST Web services on page 3374

Genero Web Services allows you to write Web Services server and client applications for the REST Web standard architecture.

Service Oriented Architecture (SOA) and Web services

Service Oriented Architecture (SOA) is based on a philosophy of how to connect systems and exchange data to solve business problems.

Rather than concentrating on a specific task or transaction, SOA addresses how to use data from various sources, reduce human involvement, and mitigate the effects of change in a business process and its supporting systems.

SOA defines the services to be provided. Web services are the means of implementing those services. Web services provide a platform-neutral technology to connect multiple systems in a flexible manner, where the platform-neutrality helps insulate the SOA from changes to the underlying systems.

An SOA's resilience to change is accomplished by adhering to good Web services design practices:

- Building a Web service that performs a specific task
- Having a rigid structure for the data

Migrating to SOA and Web services

To migrate your application from an existing integration method to a Service Oriented Architecture (SOA) one and move to Web services requires an iterative and evolutionary approach.

It requires work and diligent design. When switching to Web services, it is recommended to initially focus on short-term business benefits, targeting an SOA and Web services project that has tangible goals with measurable benefits.

Once an SOA contains some useful services, these services can be arranged together in a workflow that automates a business process. Web services can be reused to answer new questions, and be implemented as new business services in an SOA.

A well-defined Web service does not contain business logic or business process information. Since each Web service in an SOA can be called individually to perform a specific task, they can be arranged (orchestrated) together to perform many different business functions. As a result, companies with a mature SOA in place can change business processes through configuring the orchestration software as opposed to programming individual links between systems.

Planning a Web service

Creating a Web service application requires planning for the future use and reuse of the service.

When creating a Web service, you not only have to think of the task at hand, but you must also consider growth. You likely want the Web service to be flexible; to be able to handle different types of input. Prepare the Web service for what is probable. Developers should think bigger than the needs of a single application. Consider how you might reuse existing services, and how your services might be reused by others.
Security will likely play a larger role than it did previously with existing in-house application infrastructures using programmed links between systems; you will need to become versed in security issues.

Decide which protocol is right for your requirements, Simple Object Access Protocol (SOAP) or Representational State Transfer (REST).

While both are similar, it is worth researching the advantages of one over the other in terms of the requirements and scope of your project. Keep in mind when designing and coding Web services that you need to have: Flexibility, Reusability, and Interoperability.

**Related concepts**

- **SOAP** on page 3273
  Simple Object Access Protocol (SOAP) is a communication protocol that defines an XML data flow between a server and a client.

- **REST** on page 3274
  Representational State Transfer (REST) is a Web standard architecture that provides a method for communication between a Web service and a client over HTTP.

- **Designing REST Web services** on page 3374
  Planning the resources that the client can interact with is essential to REST Web service design.

**Genero Web Services extension**

Applications providing Web services use special libraries of the Genero Business Development Language.

The Genero Web Services Extension (GWS) is an extension to the Genero Business Development Language. It installs within the Genero Business Development Language directory. The fglgws package includes both Genero Business Development Language and Genero Web Services.

The Genero Application Server is required to manage your Web services in a deployment environment. It is not required for Web services development, unless you are interested in testing deployment issues.

**Important:** When programming a Web service, your applications must include `IMPORT com` at the top of each module. This imports the Genero Web Services Extension library named `com`:

```
IMPORT com
```

**Web services standards**

Web services standards are defined by the World Wide Web and other organizations.

For more information about these standards, refer to the "Web services" section of [http://www.w3.org](http://www.w3.org). The Genero Web Services package supports the WSDL 1.1 specification of March 15, 2002 as well as some previous specifications.

Starting with FGLGWS version 3.20, the GWS supports the OpenAPI Specification (OAS), based on the original Swagger 2.0 specification, as the standard for defining and describing RESTful APIs. For more information see the web site at [http://www.openapis.org/](http://www.openapis.org/).

The standards involved in what is commonly called "Web services" include XML, JSON, XML Schema, SOAP, REST, WSDL, and HTTP.

**XML**

XML (eXtensible Markup Language) defines a machine-independent way of exchanging data.

For example, an XML representation of the following BDL data structure:

```bdl
DEFINE Person
RECORD Attribute (XMLName="Person")
    FirstName VARCHAR(32) Attribute (XMLName="FirstName"),
    LastName VARCHAR(32) Attribute (XMLName="LastName"),
    Age INTEGER Attribute (XMLName="Age")
END RECORD
```
Would be:

```xml
<Person>
  <FirstName>John</FirstName>
  <LastName>Smith</LastName>
  <Age>35</Age>
</Person>
```

The record definition allows you to specify XML attributes for data types. This feature was added with Genero 2.00.

**JSON**

JavaScript Object Notation (JSON) defines a machine-independent way of exchanging data.

For example, a JSON representation of the following BDL data structure:

```bdl
DEFINE Person
RECORD Attribute (XMLName="Person")
  FirstName  VARCHAR(32) Attribute (XMLName="FirstName"),
  LastName   VARCHAR(32) Attribute (XMLName="LastName"),
  Age        INTEGER Attribute (XMLName="Age")
END RECORD
```

Would be:

```
{
  "FirstName": "John",
  "LastName": "Smith",
  "Age": 27
}
```

**XML schema**

XML Schema defines the elements, entities, and content model of an XML document.

For example, for the example document shown in the topic **XML** on page 3272, the schema might indicate that the XML document contains an element "Person", and that each "Person" contains one and only one element "FirstName", "LastName", and "Age". The XML Schema has additional capabilities, such as data type control and content restrictions.

An XML Schema allows an XML document to be validated for correctness.

**SOAP**

Simple Object Access Protocol (SOAP) is a communication protocol that defines an XML data flow between a server and a client.

Web services using the Simple Object Access Protocol (SOAP) protocol provide information about their use that developers can invoke using the **Web services Descriptive Language (WSDL)** to generate a WSDL document.

Each SOAP Web service must have an associated WSDL document, so that developers and applications know what to expect from the Web service, and how to invoke it.

The "StockQuote" service mentioned in the **Introduction to Web services** on page 3270 exchanges messages using the following syntax:

**Response**

```xml
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:getQuoteResponse="http://www.sok.com/">
  <soap:Body>
    <getQuoteResponse>
      <stockValue>999.99</stockValue>
    </getQuoteResponse>
  </soap:Body>
</soap:Envelope>
```
SOAP relies on a lower-level protocol for the transport layer. Genero Web Services use SOAP over HTTP, and can also perform low-level XML and TEXT over HTTP communications on the client side. This allows communication between applications using the core Web technology, taking advantage of the large installed base of tools that can process XML delivered plainly over HTTP, as well as SOAP over HTTP.

REST
Representational State Transfer (REST) is a Web standard architecture that provides a method for communication between a Web service and a client over HTTP.

REST Web service resources are implemented through representations. Representations are the state of the resource transferred between the client and the server. Typically, these representations are sent and received in JSON or XML format.

If, for example, the client representation is to get some records from a database, the server does not send its database, instead it sends some JSON or XML that represents these records.

RESTful Web services are stateless. Each HTTP request–response message between the client and the server must be self-descriptive and include enough information to process the message.

The Web services client application request delivers state via body contents, query parameters, request headers, and the requested URI (the resource name).

The "StockQuote" service mentioned in the Introduction to Web services on page 3270 may receive a request for services, and information is returned in JSON as shown in the following:

**Request**

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <getQuote>
      <stockSymbol>MyCompany</stockSymbol>
    </getQuote>
  </soap:Body>
</soap:Envelope>
```

**Response**

According to the HTTP request headers (Accept and Content-type), the response is delivered as JSON.

```
{
  "stockValue": "999.99"
}
```

**Related concepts**

REST Web services on page 3279
Special attributes in the BDL function provide for REST high-level Web services, while low level REST web services are implemented using standard HTTP classes.

**WSDL**
Web Services Description Language (WSDL) is an XML-based language that describes the services offered by a SOAP server.

A WSDL description or file containing that description is sufficient to provide all the information required to communicate with the server. It contains:

- The description of the operations offered by the server, and each operation's input and output messages.
- The location of the SOAP server.
- Internal connection and protocol details (transport layer, encoding, namespaces, and so on).

Genero Web Services package provides the fglwSDL tool that enables Genero client applications to obtain the WSDL description of a Web service.

**HTTP**
Hypertext Transfer Protocol (HTTP).

HTTP is the set of rules for exchanging files (text, graphic images, sound, video, and other multimedia files) on the World Wide Web.

**Web services style options**
Information on Web services Style options available for SOAP Genero Web services. There is no style concept in REST.

The Web services Style options available for created SOAP Genero Web services are WS-I (Web services Interoperability organization) compliant:

- **RPC Style Service (RPC/Literal)** is generally used to execute a function, such as a service that returns a stock option.
- **Document Style Service (Doc/Literal)** is generally used for more sophisticated operations that exchange complex data structures, such as a service that sends an invoice to an application, or exchanges a Word document.
- **RPC Style Service (RPC/Encoded)** is the legacy style, now provided for backward compatibility. You most likely will not be using the RPC/Encoded style to create new Web services.

**Related concepts**
Choosing a web services style on page 3358

Genero Web Services contains style options for creating SOAP Web services. Your choice is dependent on the type of service, (Document or RPC), and the encoding mechanism (literal or encoded) required.

**GWS demos and examples**
Demos and examples are provided for both SOAP and RESTful Web services.

**Demos**
Web Services demos are located in $FGLDIR/demo/WebServices. Each demo sits in its own folder, and includes the code for both the client and the server. A readme.txt file provides details about the demo.

For SOAP demos, the description in the readme.txt file indicates a Web service such as: an RPC style Web service demonstration, a Document Style Web service demonstration, RPC encoded Web service demonstration, Stateful Web service demonstration, or Doc Literal Web service demonstration with XML-Security.

For RESTful demos, the description in the readme.txt file indicates a Web service such as a REST service demonstration, or a low-level Web service demonstration.

**Tutorials and Examples**
Examples are provided within this manual.
For a tutorial-style example of a SOAP Web service, start with Writing a Web Services client application on page 3321 and Writing a Web Services server application on page 3342.

For a tutorial-style example of a RESTful Web service, start with Writing a Web services client application on page 3418 and Writing a Web Services server application on page 3430. The full application code can be viewed at The RESTful calculator demo source on page 3436.

Debugging

Turn on the debug mode to log the data sent or received by your Web service application.

Debug information is written to the standard error stream of the console. If needed, it can be redirected to a file.

To turn on the debugging feature, set the FGLWSDEBUG environment variable before starting the application.

The level of debugging depends on the value set for the FGLWSDEBUG variable. See FGLWSDEBUG on page 283.

Note: To debug a Web Service application managed by the Genero Application Server (GAS), you have to modify the value of the FGLWSDEBUG environment variable in the GAS configuration file. To understand how environment variables are set within the GAS configuration file, see the ENVIRONMENT_VARIABLE topic in the Genero Application Server User Guide.

Platform-specific notes

Some platforms have specific requirements.

Web Services on IBM® AIX®

Requirements for IBM® AIX®.

• The "IBM® C++ Runtime Environment Components for AIX®" must be installed in order to use Genero Web Services. See the IBM® support center for more information about downloading the component.

  Note: If not installed, you will get the following error message:

  Could not load C extension library 'com'. Reason: A file or directory in the path name does not exist.

• Due to an IBM® issue on 64-bit platforms, the OpenSSL library is unable to open the system /dev/urandom device to generate a PRNG number.

  If you want to use security APIs in your GWS application (especially if you access a server in HTTPS), install Entropy Gathering Daemon (EGD).

Web Services on GMI (iOS)

Requirements to use Web services on iOS platforms (GMI).

Web services configuration options

GWS configuration FGLPROFILE entries related to SSL/TLS keys (security.*) are not supported (uses iOS native SSL/TLS).

Long running HTTP request

When executing an HTTP request (for example with com.HTTPRequest.doRequest()), if the request takes a long time to complete (for example, several minutes), the app will go into background mode. If the user taps the app icon to return to foreground mode, the program will get the runtime error -15553.

The XSLTTransformer class

The XSLTTransformer class is not available on iOS.
Web Services on GMA (Android™)
Requirements to use Web services on Android™ platforms.

V3 SSL Certificates
The SSL certificates for secured servers must be of type V3: Android™ does not support other types of SSL certificates. When creating your own self-signed certificates (to be installed in the "Install from storage" Keystore of Android), make sure that type V3 is used.

Known issues
There are some known issues when working with Web services.
These issues are specific to SOAP Web Services.

Forcing RPC style convention when no input message
In RPC style, the convention defines names for input messages and output messages, but if there is no input message, its name cannot be redefined.

To workaround this issue, respect RPC style convention in WSDL, or force RPC convention (on client and server side) by using the -fRPC option of the fglwsdl tool.

Variable names conflicts with library names
The fglwsdl tool can generate variable names conflicting with IMPORT library names.

For example:

```plaintext
DEFINE xml xml.DomDocument
```
will conflict with the xml library, if the code defines also the instruction:

```plaintext
IMPORT xml
```

Legal Notices
Legal notices relevant to Web services.

This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/).

This product includes cryptographic software written by Eric Young (eay@cryptsoft.com).

This product includes software developed by CollabNet (http://www.Collab.Net/).

This product includes software developed by the University of California, Berkeley and its contributors.

This product includes software developed or owned by Caldera International, Inc

Concepts
These topics cover various Genero Web Services concepts.

Genero Web Services APIs
This section provides overview information on the Web services Application Program Interfaces (APIs).

Web services work by responding to requests for data and returning JSON or XML documents. As JSON and XML are simple text formats, Web services are invoked via APIs over the Hypertext Transfer Protocol (HTTP).

Genero Web Services support SOAP and REST Web services.
SOAP Web services APIs
Special APIs are provided for use in creating SOAP Web services.

High-level and low-level SOAP APIs
High-level APIs use specially-designed classes to implement your Web service's function with minimal coding, while low-level APIs use standard HTTP classes that require you to code everything by hand.

SOAP high-level APIs
- The high-level APIs include the following:
  - The com.WebServices class is used to implement a Web service on the server. Instances of this class are used to create a Web service that can be stateless or stateful. Object methods for registering HTTP headers for request and response handling, and generating the WSDL are provided by this class.
  - The com.WebOperation class provides for the interface to create and manage the operations of a Web service. Instances of this class are used to create different style options (Document or RPC) and the encoding mechanism (literal or encoded) for the service. Object methods for this class are used to set the headers, action, and encoding. A stateful Web service can also be initiated by this class.
  - The com.WebServiceEngine class provides an interface to manage the Web Services engine. Instances of this class are used to start the service and handle HTTP requests. This class also handles request faults, timeout, and cookie options for both client and server.

If you have generated code via the fglwsdl tool (client and server side), these high-level APIs are automatically used.

SOAP low level APIs
Web services can be implemented with the HttpServletRequest, HttpServletResponse and HttpSessionRequest classes, but you have to write all the HTTP code of your services by hand.

Related concepts
Writing a Web Services server application on page 3342
Create a Genero Web Services server using the SOAP protocol.

Writing a Web Services client application on page 3321
Create, configure and deploy a Genero Web Services client using the SOAP protocol.

When to use high-level SOAP APIs
The recommendation is to use the high-level APIs wherever possible to implement SOAP Web services.

High-level APIs require less code in your application. They provide functionality that would otherwise need to be handled with many low-level API calls. They are therefore easier to use in comparison with low-level APIs.

You can perform a defined set of simple functions with one call to a high-level API. The global endpoint user-defined type (tGlobalEndpointType) generated by the WSDL stub, is used by the high-level API. For example, you can set variables for the interface, binding, etc., in the endpoint type record at runtime. For an example of this use see the Compression and a Web services client on page 3295.
Related concepts

Writing a Web Services server application on page 3342
Create a Genero Web Services server using the SOAP protocol.

Writing a Web Services client application on page 3321
Create, configure and deploy a Genero Web Services client using the SOAP protocol.

When to use low-level SOAP APIs
The recommendation is to use the high-level APIs wherever possible to implement SOAP Web services.

However, when there is need for more detailed and granular control, you may opt to use the low-level APIs. Low-level APIs may be needed in specific scenarios because a high-level implementation may not yet be available.

For example, a request to send an attached file in separate parts of the request requires a low-level implementation of a multipart style SOAP request. For an example of this implementation see SOAP multipart style requests in GWS on page 3297.

REST Web services
Special attributes in the BDL function provide for REST high-level Web services, while low level REST web services are implemented using standard HTTP classes.

High-level framework and low-level REST APIs
REST Web services implemented with the high-level framework outweigh a low-level solution in terms of ease of creation. Yet make sure you are aware of the differences, and when one or other is preferred, or both are needed.

REST high level

Web services implemented with the high-level REST framework use the Function attributes feature to define details of the Web service. Function attributes are specified in an ATTRIBUTES() clause of the function to define operation type, and path. Attributes also define parameters for path template, query strings, return values, etc., of your RESTful Web service function. These special attributes for REST high-level define the framework for creating REST web services.

Web service function attributes are identified with the WS* prefix. For example, the sample function add, is defined on the server side using the attributes, WSGET, WQUERY, WSPATH, and WSDESCRIPTION.

```
PUBLIC # Query sample
FUNCTION Add(a INTEGER ATTRIBUTES (WQUERY), b INTEGER ATTRIBUTES (WQUERY))
ATTRIBUTES (WSGET, WSPATH='/' add', WSDESCRIPTION='Simple query sample')
RETURNS (INTEGER)
RETURN a + b
END FUNCTION
```
When you need to determine what functions are available on the server, you can use the fglrestful tool (server side), to get all of the information necessary in an OpenAPI specification file (JSON or YAML). The fglrestful tool can also generate code automatically for the client.

REST low-level APIs

The standard HTTP classes HttpRequest, HTTPResponse, and HTTPServiceRequest are available if you need to code individual commands. Be aware that you have to write all the HTTP code of your services by hand.

The client stub generated by the fglrestful tool implements low-level APIs.

Related concepts
RESTful Web services with high-level framework on page 3375
These topics give you the information you need to begin working with RESTful Web services applications using BDL function with support for attributes.

Related reference
High-level RESTful Web service attributes on page 3463

When to use high-level REST
The recommendation is to use the high-level REST framework wherever possible to implement REST Web services.

A high-level REST implementation requires less code in your application. Functionality is provided using the REST attributes in your function parameters and return values. These are easier to use in comparison with low-level APIs.

Related concepts
RESTful Web services with high-level framework on page 3375
These topics give you the information you need to begin working with RESTful Web services applications using BDL function with support for attributes.

When to use low-level REST APIs
The recommendation is to use the high-level APIs wherever possible to implement REST Web services.

However, when there is need for more detailed and granular control, you may opt to use the low-level APIs. Low-level APIs may be needed in specific scenarios because a high-level implementation may not yet be available.

Related concepts
Writing a Web Services server application on page 3430
To create a RESTful Genero Web services server application, there are a minimum of five steps that the server application must handle.

Writing a Web services client application on page 3418
Create, configure and deploy a RESTful Genero Web Services (GWS) client.

SOAP features
The GWS provides support for SOAP versions and for managing SOAP fault.

SOAP 1.1 and 1.2
GWS is able to communicate with Web services provided with SOAP 1.1 or SOAP 1.2.

Since 2.40, Genero Web Services (GWS) supports SOAP 1.2.
Server side

A Genero Web Services server can deliver a service in SOAP 1.1 or SOAP 1.2 using the `com.WebService.setFeature` function.

For example in $FGLDIR/demo/WebServices/calculator/server/calculatorServer.4gl, the calculator server offers the service in SOAP 1.1 and SOAP 1.2.

```plaintext
CALL serv.setFeature("Soap1.1",TRUE)
CALL serv.setFeature("Soap1.2",TRUE)
```

Client side

A GWS client can consume a service in SOAP 1.1 or SOAP 1.2.

For example:

- To create a client that consumes the Calculator service in SOAP 1.1 use command:
  ```bash
  ```
- To create a client that consumes the Calculator service in SOAP 1.2 use command:
  ```bash
  ```

Be aware that when generating different clients for each SOAP version, even if the same operations are provided, the services are using different protocols so the underlying generated stubs are also different.

SOAP Fault

Genero Web Services supports SOAP's built-in error handling.

Since 2.40, Genero Web Services supports SOAP fault.

For backward compatibility, the `fglwsdl` tool provides the `--ignoreFaults` option to disable SOAP fault management.

Server side

A Genero Web Services server can throw a SOAP fault when a processing error is encountered.

To generate a SOAP fault you need to:

- create the fault variable with `com.WebService.createFault()`
- add it to your operation with `com.WebOperation.addFault`
- use it with `com.WebServiceEngine.SetFaultDetail`

For example in $FGLDIR/demo/WebServices/calculator/server/calculatorServer.4gl, the calculator server has a `divide_by_zero` SOAP fault. The SOAP fault is raised when you try to divide a number by zero. To generate a SOAP fault proceed as follows:

Create a SOAP fault

You define the variable to send as a SOAP fault. It can be a simple string like in this example or a complex type. Remember to assign an XMLName to the variable.

```plaintext
DEFINE divide_by_zero STRING ATTRIBUTES (XMLName="DividedByZero")
```

Then you provide the fault variable to the service using the function `com.WebService.createFault()`.

```plaintext
CALL serv.createFault(divide_by_zero,FALSE)
```
Add the SOAP fault to an operation

A SOAP fault can be used by an operation in response to the client when an error has occurred in the service request. An operation can use different SOAP faults but **only one at a time**.

```plaintext
LET op = com.WebOperation.CreateRPCStyle("divide","Divide",divide_in,divide_out)
CALL op.addFault(divide_by_zero,NULL)
```

Here, the SOAP fault is added to the "divide" operation.

**Send the SOAP fault**

Set the values to the fault variable. The fault message is sent to the client at the end of the operation processing.

```plaintext
LET divide_by_zero = "Cannot divide "||divide_in.a||" by zero"
CALL com.WebServiceEngine.SetFaultDetail(divide_by_zero)
```

**Client side**

A Genero Web Services client can receive a SOAP fault number in the operation status and act accordingly.

If a SOAP fault occurs, the operation returns the SOAP fault number in the operation status. The SOAP fault number is defined in the generated stubs as a BDL constant prefixed with the string **FaultID_**.

**Note:** A SOAP fault can occur in case of HTTP error 200 and 500.

For example in `$FGLDIR/demo/WebServices/calculator/client/ws_calculator.inc`, the `Divide` operation has a SOAP fault that informs the client when a number is divided by zero.

```plaintext
# List of Soap fault constants
CONSTANT FaultID_DividedByZero = 1
...
# VARIABLE : DividedByZero
DEFINE DividedByZero STRING ATTRIBUTES(XMLName="DividedByZero",
   XMLNamespace="http://tempuri.org/")
...
# Operation: Divide
# FAULT #1: GLOBALS DividedByZero
```

You can test the operation status code accordingly and display the SOAP fault message.

For example in `$FGLDIR/demo/WebServices/calculator/client/calculatorClient.4gl`, when the divide operation status is 1, `DividedByZero` message is displayed.

```plaintext
ON ACTION divide
   CALL Divide(op1, op2) RETURNING wsstatus, result, remaind
   CASE wsstatus
      WHEN 0
         DISPLAY BY NAME result,remaind
         DISPLAY "OK" TO msg
      WHEN FaultID_DividedByZero
         DISPLAY DividedByZero TO msg
      OTHERWISE
         DISPLAY wsError.description TO msg
   END CASE
```
Stateful SOAP Web services

The GWS provides support for stateful service. Different options for implementing SOAP stateful services are described.

Concept
A stateful service is a service that maintains a context between a Web services client and server. It enables the service to keep track of previous requests from that context, in order to manage different states in the Web service server.

SOAP stateful Web service

Genero Web Services supports two kinds of stateful services:

- Based on the WS-Addressing 1.0 specification to define the XML format used to convey the context from the client to the server.
- Based on an HTTP session cookie to convey the context from the client to the server.

The Genero Web Service engine uses a BDL variable defined at stateful service creation via `createStatefulWebService()` as service context. Use that variable to hold a service state in a database.

It is up to the BDL programmer to create, store and remove the service state in the database.

The SOAP engine is responsible for:

- Deserializing the state variable when getting a new incoming request. The programmer can then read the state variable for any published BDL Web service operation and restore the service state corresponding to that variable.
- Serializing a new instance of the state variable in a Web service response for all BDL Web service operations set as session initiator via `initiateSession()`. The programmer must instantiate a new state by filling the state variable and storing it in a database for future use.

WS-Addressing 1.0 stateful services

WS-Addressing 1.0 uses the WS-Addressing EndpointReference type as a state variable to maintain a stateful service.

The `EndpointReference` type is independent from the transport layers. (See WS-Addressing 1.0 EndpointReferenceType). The session state is conveyed from the client to the server as WS-Addressing 1.0 reference parameters.

Server side

Perform these steps to create a WS-Addressing stateful service.

Step 1: Declare a W3C EndpointReference record to be used as state variable

This record MUST have:

- A mandatory member of type STRING, where you can define a different service end point URL, otherwise the current server URL will be used.
- A sub record to contain one or more BDL variables used as state variables and defined as reference parameter in the WS-Addressing 1.0 specification.

For example:

```bdl
DEFINE EndpointReferenceState RECORD ATTRIBUTES(W3CEndpointReference)
  address STRING, # Mandatory
  ref RECORD # Sub-record Reference parameters containing one # or more state variables
    OpaqueID STRING ATTRIBUTES(XMLName="OpaqueID"), # Unique ID to
    # identify the service state in the database
    Expiration DATE ATTRIBUTES(XMLName="Expiration",
    # Session state expiration date
    XMLNamespace="http://tempuri.org")
END RECORD
END RECORD
```

You can use a unique ID from a database table to manage the Web services sessions in place of `OpaqueID`. 
Related concepts
Step 2: Create a stateful WS-Addressing enabled web service with W3CEndpointReference record as a parameter on page 3284

**Step 2: Create a stateful WS-Addressing enabled web service with W3CEndpointReference record as a parameter**

The Genero Web Service extension provides a new Web service constructor called `createStatefulWebService()` to perform stateful services. This function works as the stateless constructor, but expects a W3CEndpointReference record as parameter.

For example:

```
DEFINE serv com.WebService
  EndpointReferenceState) # Create a stateful service
  # with a W3CEndpointReference state variable
CALL serv.setFeature("WS-Addressing1.0","REQUIRED") # enable
  # support of WS-Addressing 1.0
```

Related concepts
Step 3: Publish a web service operation returning the W3CEndpointReference state variable and set it as session initiator on page 3284

**Step 3: Publish a web service operation returning the W3CEndpointReference state variable and set it as session initiator**

You must define which Web service operation will initiate the session on your service and return the W3CEndpointReference state variable.

All other Web service operations (not defined as session initiator) will return an error if they don't get reference parameters defined in the W3CEndpointReference state variable as WS-Addressing 1.0 headers.

For example:

```
DEFINE op com.WebOperation
LET op = com.WebOperation.CreateDocStyle("GetInstance",
  "GetInstance",NULL,EndpointReferenceState)
CALL op.initiateSession(TRUE)
CALL serv.publishOperation(op,NULL)
```

There is no restriction regarding the input parameter of the Web service initiator function, but the output parameter must be the same W3CEndpointReference record passed to the service creation constructor.

It is not required to have a Web operation which initiates the session in the same service, but then you have to return the same W3CEndpointReference record in another Web service to instantiate the session, such as a Factory service that instantiates all sessions for other stateful services.

Related concepts
Step 4: Create the BDL session initiator function and instantiate a new session on page 3284

**Step 4: Create the BDL session initiator function and instantiate a new session**

In your BDL function declared as session initiator, you have to:

- Handle the creation of the session
- Fill the state variable before returning from the function
- Store the new session in a database based on the state variable (in order to keep the session across consecutive requests from the same client).
For example:

FUNCTION GetInstance()
    LET EndpointReferenceState.address = NULL
    # Use default end point location
    LET EndpointReferenceState.ref.OpaqueID = com.Util.CreateUUIDString()
    # Generate an unique string (can come from a database table id)
    LET EndpointReferenceState.ref.Expiration = CURRENT + INTERVAL HOUR TO HOUR (1)
    # Create expiration date in one hour to discard request after that date
    ... Store OpaqueID into database or use directly a database table entry
    ... to hold the session
END FUNCTION

Related concepts
Step 5: Restore the session in any BDL web operation from the W3CEndpointReference record on page 3285

Step 5: Restore the session in any BDL web operation from the W3CEndpointReference record

In a published BDL Web function, the SOAP engine deserializes the WS-Addressing 1.0 reference parameter headers in the W3CEndpointReference sub-record so that you can retrieve the session from the state variable.

For example:

FUNCTION MyFunction()
    IF EndpointReferenceState.ref.OpaqueID IS NULL THEN
        CALL com.WebServiceEngine.SetFaultString("Invalid session id")
        RETURN
    ELSE
        ... Restore the service session based on the OpaqueID state
        ... variable from the database
    END IF
    ... Process the operation
END FUNCTION

Related concepts
Client side on page 3285
Perform these steps to communicate with a stateful Web service based on WS-Addressing 1.0.

Client side
Perform these steps to communicate with a stateful Web service based on WS-Addressing 1.0.

Step 1: Generate the client stub from your WS-Addressing stateful service

Use the fglwsdl tool as usual. It will detect that the service returns a W3CEndpointReference and generate the appropriate code.

The WSDL imports the WS-Addressing 1.0 schema, so the fglwsdl tool requires access to the W3C server. Use the option -proxy if you need to connect via a proxy server.

For example:

$ fglwsdl -o ws_stub http://localhost:8090/StatefulWSAddressingService?WSDL

The generated .inc file contains a variable of type tWSAGlobalEndpointType to be used to transmit the WS-Addressing 1.0 reference parameters.

Example of a global variable name

DEFINE
    StatefulWSAddressingService_StatefulWSAddressingServicePortTypeEndpoint
**Step 2: Create the MAIN application**

In your main application:

1. Import the XML library. This is due to the support of WS-Addressing 1.0 with `IMPORT XML`.
2. Import the generated `.inc` file with `GLOBALS "ws_stub.inc"`.
3. Manage the WS-Addressing 1.0 reference parameters representing the session state (if your client has to handle several instances of the same service).

For example:

```bdl
IMPORT XML # Import the XML library required for WS-Addressing 1.0
GLOBALS "ws_stub.inc" # Import service global definition

TYPE InstanceType DYNAMIC ARRAY OF xml.DomDocument
# End point WSA reference parameters

DEFINE instance1,instance2,instance3 InstanceType
# Store the different sessions the client will have to manage

MAIN
...
END MAIN
```

**Related concepts**

**Step 3: Instantiate a new session by calling the web service operation set as session initiator**

Call the BDL function generated from the WSDL that is defined as session initiator on the server. This function returns a `W3CEndpointReference` parameter that contains the WS-Addressing 1.0 reference parameters representing the new instance created on server side.

If your application handles several instances, you will have to copy and store those parameters in your application to identify a service instance for further requests.

As the WS-Addressing 1.0 reference parameters are defined as an XML document, they are represented as a dynamic list of `xml.DomDocument` in BDL.

For example:

```bdl
DISPLAY "Creating a new instance ..."
LET wsstatus = GetInstance_g() # call the service session initiator
# web function
IF wsstatus == 0 THEN
   FOR ind=1 TO
      ns1GetInstanceResponse.return.ReferenceParameters._LIST_0.getLength()
      LET instance1[ind]=
         ns1GetInstanceResponse.return.ReferenceParameters._LIST_0[ind].clone()
      # copy the service returned WS-Addressing 1.0 reference parameters
   END FOR
ELSE
   ... handle soap errors
END IF
```
**Important:** When creating a new instance, ensure that the Parameters member of the generated global variable of type `tWSAGlobalEndpointType` has been set to `NULL`, otherwise the server will throw an error.

**Related concepts**

*Step 4: Call any web service operation with previously returned WS-Addressing 1.0 reference parameters* on page 3287

*Step 4: Call any web service operation with previously returned WS-Addressing 1.0 reference parameters*

Before calling any Web service operation, you must set the WS-Addressing 1.0 reference parameters returned by a session initiator function to identify the session to the server.

For example:

```plaintext
LET StatefulWSAddressingService_StatefulWSAddressingServicePortTypeEndpoint.Address.Parameters.* = instance1.*
# assign WS-Addressing 1.0 reference parameters dynamic array by reference
CALL MyFunction("Hello") RETURNING wsstatus,ret
# Call web operation MyFunction of instance 1
```

**Stateful services based on HTTP cookies**

A stateful service based on HTTP cookies uses the HTTP transport protocol and its ability to transmit cookies, used as session context.

**Note:** This works only if the communication path between the client and the server is carried in HTTP, otherwise it is recommended to use WS-Addressing stateful services.

**Server side**

Perform these steps to create an HTTP cookie based stateful service.

*Step 1: Declare any BDL simple variable to be used as state variable*

For example:

```plaintext
DEFINE ServiceState STRING # Unique ID to identify the service state in the database
```

For instance, you can use a unique ID from a database table to manage the Web services sessions.

**Related concepts**

*Step 2: Create a stateful web service with state variable as parameter* on page 3287

*Step 2: Create a stateful web service with state variable as parameter*

The Genero Web Service extension provides a new Web service constructor called `createStatefulWebService()` to perform stateful services. This function works as the stateless constructor, but expects a simple state variable as parameter.

**Example**

```plaintext
DEFINE serv com.WebService
# Create a stateful service with a simple BDL variable as state variable
```

**Related concepts**

*Step 3: Publish a Web service operation defined as session initiator* on page 3287

*Step 3: Publish a Web service operation defined as session initiator*

Define which Web service operation will initiate the session on your service and instantiate a new session. All other Web service operations (not defined as session initiator) will return an error if they don't get an HTTP cookie called `GSESSIONID`. 
For example:

```plaintext
DEFINE op com.WebOperation
LET op =
CALL op.initiateSession(true)
CALL serv.publishOperation(op,NULL)
```

There is no restriction on the Web service session initiator function regarding the input and output parameters.

**Related concepts**

*Step 4: Create the BDL session initiator function and instantiate a new session* on page 3288

**Step 4: Create the BDL session initiator function and instantiate a new session**

In your BDL function declared as session initiator, you must:

- Handle the creation of the session.
- Fill the state variable before returning from the function.
- Store the state variable in a database based on the state variable (in order to keep the session across consecutive requests from the same client).

For example:

```plaintext
FUNCTION GetInstance()
    # Generate an unique string (can come from a database table id)
    LET ServiceState = com.Util.CreateUUIDString()
    ... Store ServiceState value into database or use directly a database table entry to hold the session
END FUNCTION
```

**Related concepts**

*Step 5: Restore the session in any BDL web operation from the state variable* on page 3288

**Step 5: Restore the session in any BDL web operation from the state variable**

In a published BDL web function, the SOAP engine deserializes the HTTP Cookie called `GSESSIONID` from the HTTP layer into the state variable. You can then retrieve the session in BDL via that state variable.

For example:

```plaintext
FUNCTION MyFunction()
    IF ServiceState IS NULL THEN
        CALL com.WebServiceEngine.SetFaultString("Invalid session id")
        RETURN
    ELSE
        ... Restore the service session based on the ServiceState variable from the database
    END IF
    ... Process the operation
END FUNCTION
```

**Related concepts**

*Step 6: Deployment recommendation* on page 3288

**Step 6: Deployment recommendation**

When deploying stateful Web services based on HTTP cookies, the complete server path will be added to the cookie when first instantiated, so you must pay attention to that URL. In other words, you MUST always call the service via the complete URL containing the service name.
For instance if your service is named **MyService** and if your GAS configuration file is called "Server.xcf", the stateful service is accessible at URL: http://localhost:6394/ws/r/group/Server/MyService.

**Client side**
Perform the following steps to communicate with a stateful Web service based on HTTP cookies.

*Step 1: Generate the client stub from your stateful service*

Use the fglwsdl tool as usual.

For example:

```bash
$ fglwsdl -o ws_stub http://localhost:8090/StatefulCookieService?WSDL
```

The generated `.inc` file contains a variable of type `tGlobalEndpointType` to be used to transmit the HTTP Cookie.

**Example of a global variable name**

```plaintext
DEFINE StatefulCookieService_StatefulCookieServicePortTypeEndpoint tGlobalEndpointType
```

**Related concepts**

*Step 2: Create the MAIN application* on page 3289

**Step 2: Create the MAIN application**

In your main application:

- Import the generated `.inc` file with `GLOBALS "ws_stub.inc"`.
- Manage the HTTP cookies representing the session state (if your client has to handle several instances of the same service).

For example:

```plaintext
GLOBALS "ws_stub.inc" # Import service global definition

# Store the different sessions the client will have to manage
# in a string
DEFINE instance1,instance2,instance3 String

MAIN
    ...
END MAIN
```

**Related concepts**

*Step 3: Instantiate a new session by calling the web service operation set as session initiator* on page 3289

**Step 3: Instantiate a new session by calling the web service operation set as session initiator**

Call the BDL function generated from the WSDL that was defined as session initiator on the server. This function returns a new HTTP Cookie saved in the `Binding.Cookie` member of the global service variable of type `tGlobalEndpointType`. If your application handles several instances, you will have to copy and store that cookie in your application to identify a service instance for further requests.

For example:

```plaintext
DISPLAY "Creating a new instance ..."
LET wsstatus = GetInstance_g() # call the service session
    # initiator web function
IF wsstatus == 0 THEN
    # copy the service returned HTTP cookie
    LET instance1 =
```
When creating a new instance, ensure that the Binding.Cookie member of the generated global variable of type tGlobalEndpointType has been set to NULL, otherwise the server will throw an error.

**Related concepts**

Step 4: Call any web service operation with previously returned HTTP cookie on page 3290

*Step 4: Call any web service operation with previously returned HTTP cookie*

Before calling any Web service operation, set the HTTP cookie returned by a session initiator function to identify the session to the server.

For example:

```plaintext
# use instance1
LET
  StatefulCookieService_StatefulCookieServicePortTypeEndpoint.Binding.Cookie = instance1
# Call web operation MyFunction of instance 1
CALL MyFunction("Hello") RETURNING wsstatus,ret
```

**Related concepts**

Step 5: Troubleshooting on page 3290

*Step 5: Troubleshooting*

If your Genero application doesn't set the HTTP cookie when accessing a stateful service via the GAS, it may be because you did not use the complete URL when accessing the service.

For instance if your service is named MyService and if your GAS configuration file is called Server.xcf, the stateful service is accessible at URL: http://localhost:6394/ws/r/group/Server/MyService.

**Encryption, BASE64 and password agent with fglpass tool**

Genero Web Services supports password encryption with fglpass as password agent.

For security reasons, it is recommended that you avoid storing clear passwords in a file. The Genero Web Services enables the password encryption of a HTTP Authenticate entry in the FGLPROFILE file. The encrypted password is decrypted by the Genero Web Services engine when required.

**The fglpass tool**

With the fglpass tool you can encrypt and decrypt passwords.

The Genero Web Services package provides a command line tool called fglpass, which can be used to encrypt a password from an X.509 certificate or a RSA private key. The encrypted password is displayed on the console in Base64 form, composed only of alphanumerical characters, and therefore easily usable in any text file.

See [fglpass](#) for more details.

**Encrypt a HTTP authenticate password for FGLPROFILE**

Use the fglpass tool to encrypt a password to store in the FGLPROFILE file.

1. Find the HTTP Authenticate entry with the password you want to encrypt:

```plaintext
authenticate.myentry.login = "mylogin"
authenticate.myentry.password = "mypassword"
```
2. Add the certificate and its private key in the FGLPROFILE file as follows:

```plaintext
security.mykey.certificate = "MyCertificate.crt"
security.mykey.privatekey  = "MyPrivateKey.pem"
```

3. Encrypt the password with fglpass:

```
$ fglpass -e -c MyCertificate.crt
Enter password :mypassword
```

The fglpass output looks like the following:

```
BASE64 BEGIN
dBy3E5JCxuoxsR+aOBVfp1j0SwQPt+hdjpMKriWvO2xM5rFnFEwv+sPPd4w
/onWV1GOM5mqubBeS7QUt/2KOD1aO9/R5RVa5wy1Qu//6vxfyd8NG/
SFJm1VH63kuyXf1Vfq6bHo5+n1Q2pVjSHF2msET3S9HTp2Ut4NbiP4=BASE64 END
```

**Note:** The encrypted password is enclosed between BASE64 BEGIN and BASE64 END. In the above example, the cyphertext is wrapped for display purposes only.

4. Replace the clear password with the encrypted one, and specify the key used to encrypt it (mykey in our example):

```
authenticate.myentry.login = "mylogin"
authenticate.myentry.password.mykey = "dBy3E5JCxuoxsR+aOBVfp1j0SwQPt+hdjpMKriWvO2xM5rFnFEwv+sPPd4w
/onWV1GOM5mqubBeS7QUt/2KOD1aO9/R5RVa5wy1Qu//6vxfyd8NG/
SFJm1VH63kuyXf1Vfq6bHo5+n1Q2pVjSHF2msET3S9HTp2Ut4NbiP4=
```

**Note:** Do not forget to put quotes around the base64 form; otherwise the '=' character is interpreted during the loading of FGLPROFILE. The long line of text is wrapped for display purposes only.

**Encrypt a HTTP authenticate password using a certificate in the Windows key store**

Use the fglpass tool to encrypt a password to store in the Windows® key store.

1. Find the HTTP authenticate entry with the password you want to encrypt:

```
authenticate.myentry.login    = "mylogin"
authenticate.myentry.password = "mypassword"
```

2. Add the subject of the certificate registered in the Windows® key store:

```
security.mykey.subject = "Georges"
```

3. Encrypt the password with fglpass:

```
$ fglpass -c Georges
Enter password :mypassword
```

The fglpass output looks like this:

```
BASE64 BEGIN
dBy3E5JCxuoxsR+aOBVfp1j0SwQPt+hdjpMKriWvO2xM5rFnFEwv+sPPd4w
/onWV1GOM5mqubBeS7QUt/2KOD1aO9/R5RVa5wy1Qu//6vxfyd8NG/
SFJm1VH63kuyXf1Vfq6bHo5+n1Q2pVjSHF2msET3S9HTp2Ut4NbiP4=BASE64 END
```

**Note:** The encrypted password corresponds to the string of alphanumeric characters between BASE64 BEGIN and BASE64 END. The long line of text is wrapped for display purposes only.
4. Replace the clear password with the encrypted one, and specify the key used to encrypt it (mykey in our example):

```plaintext
authenticate.myentry.login = "mylogin"
authenticate.myentry.password.mykey = "dBy3E5JCxuosxR+aOBVfpj0SWQPT+hdjphKriWv02zXnd5rFPEwv+spPd4w/onWvIgOM5mqubBeS7Q1Ut/ZK0DiAo9/R5RvA5wy1Qu/6vxyd8NG/SFJmltVGH63kuyXf1Fqq6bHo5+niQZpVJSfF2msEiT3S9HTpZUt4NbiP4="
```

**Note:** Do not forget to put quotes around the base64 form; otherwise the '=' character is interpreted. The long line of text is wrapped for display purposes only.

**Use the password agent**

Start the fglpass tool as the password agent.

The `fglpass` tool can be started as an agent with the `-agent:tcp-port` option, to allow BDL applications requiring passwords, to grant access to private keys without providing the passwords.

Passwords are provided once for each private key at the `fglpass` agent startup.

The `fglpass` agent has the following `fglrun` authentication methods:

- **By default,** the agent allows only `fglrun` to access the passwords, if the OS user executing `fglpass` and `fglrun` are the same.
- **On UNIX® platforms,** the `fglpass` agent can be started with the `-gid` option, to allow all users belonging to the OS group of users executing the `fglpass` program:

```plaintext
fglpass -gid -agent:4242 myprivate1.pem myprivate2.pem ...
```

For `fglrun` programs requiring the group-based agent authentication method, you need to set the following `FGLPROFILE` entry:

```plaintext
security.global.agent.gid=true
```

Authentication and data encryption are performed between the BDL application and the agent to guarantee passwords confidentiality, and the passwords are also stored in encrypted form in the agent memory.

1. To start the password agent at port number 4242 and to serve the BDL applications with the passwords of the private key RSAKey1.pem and DSAKey2.der, specify the option `-agent`, followed by a colon, followed by the port number where it will be reachable, followed by the list of private keys the agent will handle for all BDL applications.

```plaintext
fglpass -agent:4242 RSAKey1.pem DSAKey2.der
```

2. The agent prompts you to silently enter the password of the different keys (the passwords are not displayed to the console while being typed). In this example, you have:

```plaintext
Enter pass phrase for RSAKey1.pem:
```

Followed by:

```plaintext
Enter pass phrase for DSAKey2.der:
```

3. Once all keys have been processed, the following message is displayed to notify that the agent is ready.

```plaintext
Agent started
```

4. To enable one BDL application to use the password agent capability, set the entry called `security.global.agent` in the `FGLPROFILE` file with the port number of the agent.
In our example, with value 4242:

```
security.global.agent  = "4242"
```

**Encrypt a password from RSA key**

Use the `fglpass` tool to encrypt a password from a RSA key.

The `fglpass` tool can encrypt a password using an RSA key or certificate, and then encode it in BASE64 form. This allows you to easily add a protected password in the FGLPROFILE file for future use by any BDL application.

1. To encrypt a password from a RSA key and encoded in BASE64, enter:

   ```
   fglpass -e -k RSAPub.pem
   ```

2. You are prompted to enter the password you want to encrypt.

   ```
   Enter password : hello
   ```

   The `fglpass` tool outputs the BASE64 form of the encrypted password on the console.

   ```
   BASE64 BEGIN
   Pzk/fNRhetdJDZz5kjNg7P0XET4XsW6bys/fi0DvuqxrPh9d/s41oAws65JY0EPb2zytQjxzZ/dwaaRzJPYoQmA==
   BASE64 END
   ```

   **Note:** The BASE64 encrypted password is the string between the `BASE64 BEGIN` and `BASE64 END`.

**Related concepts**

- [Decrypt a password from BASE64](#)
  - The `fglpass` tool can decrypt a BASE64 encoded and encrypted password using a RSA private key.

**Decrypt a password from BASE64**

The `fglpass` tool can decrypt a BASE64 encoded and encrypted password using a RSA private key.

The `fglpass` tool uses the RSA private key that was used to encrypt it or that is associated to a certificate containing the public part of that private key.

1. To decrypt a BASE64 encoded and encrypted password from a RSA private key, enter:

   ```
   fglpass -d -k RSAPriv.pem
   ```

2. If the RSA key is protected with a password, you are prompted to enter that password (the password is not displayed to the console while being typed).

   ```
   Enter pass phrase for RSAPriv.pem:
   ```

3. You are prompted to enter the BASE64 encoded and encrypted password you want to decrypt.

   ```
   Enter password : Pzk/fNRhetdJDZz5kjNg7P0XET4XsW6bys/fi0DvuqxrPh9d/s41oAws65JY0EPb2zytQjxzZ/dwaaRzJPYoQmA==
   ```

   The `fglpass` tool outputs the password in clear text on the console.

   ```
   hello
   ```

**Related concepts**

- [Encrypt a password from RSA key](#)
Use the fglpass tool to encrypt a password from a RSA key.

**Encode a file in BASE64 form**
The fglpass tool can encode a file in BASE64 form.

1. To encode the file *MyFile* in BASE64, enter:

   ```
   fglpass -enc64 MyFile
   ```

   The fglpass tool outputs the BASE64 form of the file to the console.

   ```
   BASE64 BEGIN
c2VjdXJpdHkuZ2xvYmFsLmFnZW50ICAgICAgICAgICAgPSAiNDI0MiINCmNyeXB0by5pZDEua2V5ICAgICAgICAgICAgPSAiMjM5IzXkIGICAgICAgICAgICAgPSAiY2Vydml2ZSINCmNyeXB0by5pZDEua2V5ICAgICAgICAgICAgPSAiMTQWPSIyMzY0MmFyZSINCmNyeXB0by5pZDEua2V5ICAgICAgICAgICAgPSAiMjM5IzXkIGICAgICAgICAgICAgPSAiY2Vydml2ZSINCmNyeXB0by5pZDEua2V5ICAgICAgICAgICAgPSAiMjM5IzXkIGICAgICAgICAgICAgPSAiY2Vydml2ZSI=`

   Note:
   - The BASE64 encoded file is the string between BASE64 BEGIN and BASE64 END.
   - You can redirect the output of fglpass tool to a file. For example:

     ```
     fglpass -enc64 MyFile > Base64filename
     ```

   **Related concepts**
   - *Decode a file encoded in BASE64 form* on page 3294

**Decode a file encoded in BASE64 form**
The fglpass tool can decode a BASE64 encoded file.

1. To decode a file encoded in BASE64 form, enter:

   ```
   fglpass -dec64 Base64filename
   ```

   The fglpass tool outputs the file in clear form on the console.

   ```
   security.global.agent = "4242"
crypto.id1.key = "RSA1024Key.pem"
crypto.id2.key = "RSA2048Key.pem"
crypto.id3.key = "DSA1024Key.pem"
crypto.id4.key = "RSA512Protected.pem"
crypto.id5.key = "DSA512ReallyProtected.pem"
```

Note:
- You don't have to remove the BASE64 BEGIN and BASE64 END tags, if they are present in the file, because the fglpass tool detects and removes them automatically.
- You can redirect the output of the fglpass tool to a file. For example:

   ```
   fglpass -dec64 Base64MyFile > MyFile2
   ```

**Related concepts**
- *Encode a file in BASE64 form* on page 3294
The fglpass tool can encode a file in BASE64 form.

**HTTP compression in SOAP**

HTTP compression is a capability that can be built into web servers and web clients to make better use of available bandwidth, and provide greater transmission speeds between both.

There are a variety of places where you can set up HTTP compression.

- You can set up the Web services client to send and receive compressed requests. See Compression and a Web services client on page 3295.
- You can enable compression for the Web server. Refer to your Web server documentation for details.
- You can enable compression in the Genero Application Server. Compression is enabled by default in $FGLASDIR/etc/imt.cfg. Refer to the Compression in Genero Application Server page in Genero Application Server User Guide for more information.
- You can set up the Web services server to send and receive compressed requests. See Compression and a Web services server on page 3296.

**Compression and a Web services client**

Send and receive compressed requests from a Web services client.

When you create a low-level Web service and do not have any stubs created by fglwsdl, you need to manage it by setting the HTTP headers.

**Important:** HTTP request compression for POST/PUT is not supported on GMI mobile devices.

**Send a compressed request**

The method used to set up the client for sending a compressed request depends on whether the Genero Web Services client is a high-level or low-level Web services client. A high-level client is a Genero Web Services client that includes the stub files created by the fglwsdl tool. A low-level client is a Genero Web Services client that does not utilize stub files created by the fglwsdl tool.

Regardless of the type of client, the server must be set up to handle such compression, otherwise the request will be rejected.

**Send a compressed request from a high-level client**

Set the variable Binding.CompressRequest to either "gzip" or "deflate".

```plaintext
LET EchoDocStyle_EchoDocStylePortTypeEndpoint.Binding.CompressRequest = "gzip"
```

The Binding.CompressRequest variable is defined in the stub file, specifically the client's global (inc) file.

```plaintext
# Global Endpoint user-defined type definition

# End point

TYPE tGlobalEndPointType RECORD # End point
  Address RECORD # Address
    Uri STRING # URI
  END RECORD,
  Binding RECORD # Binding
    Version STRING, # HTTP Version (1.0 or 1.1)
    Cookie STRING, # Cookie to be set
    ConnectionTimeout INTEGER, # Connection timeout
    ReadWriteTimeout INTEGER, # Read write timeout
    CompressRequest STRING # HTTP request compression mode (gzip or deflate)
  END RECORD
END RECORD
```
Send a compressed request from a low-level client

A low-level client is a Genero Web Services client that does not utilize stub files created by the fglwsdl tool.

Set the Content-Encoding field in the request header to either "gzip" or "deflate".

This example sets the Content-Encoding field to "gzip", where the request is a com.HTTPRequest object.

```
CALL request.setHeader("Content-Encoding","gzip")
```

Accept a compressed response

A Genero Web Services client can accept a compressed request if it sets the Accept-Encoding field in the header to "gzip, deflate". These values represent supported compression schema names (called content-coding tokens) separated by commas.

This example sets the Accept-Encoding field with the setHeader method, where the request is a com.HTTPRequest object.

```
CALL request.setHeader("Accept-Encoding","gzip, deflate")
```

Related concepts

- Global Endpoint user-defined type definition on page 3329
- Bindings defined for the Web service can be changed at runtime.
- WS client stubs and handlers on page 3328
- To access a remote Web Service, you first must get the WSDL information from the service provider.
- Compression and a Web services server on page 3296
- Send and receive compressed requests from a Web services server.
- High-level and low-level SOAP APIs on page 3278
- High-level APIs use specially-designed classes to implement your Web service's function with minimal coding, while low-level APIs use standard HTTP classes that require you to code everything by hand.

Compression and a Web services server

Send and receive compressed requests from a Web services server.

If the Genero Web Services client accepts compression, the Genero Web Services server will reply with a compressed response.

To disable compression, you must disable compression in the Genero Application Server $FGLASDIR/etc/imt.cfg file. See the Compression in Genero Application Server page in Genero Application Server User Guide for more information.

Related concepts

- Compression and a Web services client on page 3295
Send and receive compressed requests from a Web services client.

**SOAP multipart style requests in GWS**

This topic describes multipart support with Genero Web Services

**What is multipart style in SOAP?**

Multipart style SOAP is the ability to send and receive a SOAP request in multiple pieces. The sending of attached files in separate parts of the SOAP request is one example of a multipart style SOAP request.

**Multipart SOAP on the client**

When using a WSDL with multipart style, `fglwsdl` generates a client-side stub handling multipart requests. For more details, see *Multipart in the client stub* on page 3332.

**Multipart SOAP on the server**

Multipart style is not yet supported with the high-level WS API of Genero.

- It is not possible to write a GWS server handling multipart style SOAP requests with the high-level API.
- When generating code from a WSDL using multipart style, the `fglwsdl` will produce a warning message:
  
  **WARNING : Unable to manage MIME Mutlipart binding on message 'name', where name is the name of the message in XML.**

**Implementing multipart using the low-level APIs**

If required, you can implement a WS server handling multipart with the low-level APIs of Genero Web Services. For more details, see *com.HTTPServiceRequest.getRequestMultipartType* on page 2855.

**Related concepts**

*High-level and low-level SOAP APIs* on page 3278

High-level APIs use specially-designed classes to implement your Web service's function with minimal coding, while low-level APIs use standard HTTP classes that require you to code everything by hand.

*`fglwsdl` on page 2081*

The `fglwsdl` tool produces web services stub files for client or server programs (from WSDL / XSD).

**Message Transmission Optimization Mechanism (MTOM)**

Use MTOM to efficiently send binary data to and from SOAP Web services.

**When to use MTOM**

When you have a Web service transmitting large documents, it is recommended to use MTOM. MTOM must be used when you have to optimize the transfer of binary data located in a BLOB. Any BDL BYTE will be transferred as an HTTP part over the wire. Nothing changes from the programmer perspective: the program manipulates a BDL BLOB on the client side as well as on the server side. The programmer may not even know that an HTTP part was used.

**How to implement MTOM with a GWS server application**

If you are creating a server application from scratch, use `setFeature` to enable MTOM. See *Enabling MTOM on the server side* on page 3354.

If you are using the `-s` option of the `fglwsdl` command to generate the server stubs, you can also use the `-mtom` option of the `fglwsdl` command to override the WS-Policy; you normally do not have to override the WS-Policy. See *`fglwsdl` on page 2081*.
How to implement MTOM with a GWS client application

For a GWS client, if the Web service enabled MTOM, then the generated WSDL should include the MTOM policy and it should work transparently. If the MTOM policy is not included in the WSDL and you know that the Web service is using MTOM, you can force the generation of stubs with MTOM support with the -mtom option of the fglwsdl command. The -mtom option is only needed if you want to override the WS-Policy; you normally do not have to override the WS-Policy. See fglwsdl on page 2081.

Optimization Layer 1: Using MTOM

Optimization has layers.

When MTOM is used, GWS manages the large documents using BYTE variables. When MTOM is enabled, a BDL BLOB is transmitted transparently via a HTTP attachment. Your Genero program receives a BLOB variable, filled as before, oblivious to how the content was transmitted. It is “seen” as a BYTE, as if it was inline.

Optimization Layer 2: Using STRING variables

In addition to enabling MTOM, you can further optimize by using STRING variables instead of BYTE variables.

Note: A BYTE variable contains the data value, located in memory. A STRING variable contains the path to a file. The -hexb64AsString option of the fglwsdl command forces the stub generation with STRING instead of BYTE.

For example, you can avoid loading a big file into a BYTE entirely in memory. Instead of doing this:

```plaintext
DEFINE req RECORD
    data BYTE
END RECORD
DEFINE resp RECORD
    data2 BYTE
END RECORD
LOCATE req.data IN MEMORY
LOCATE resp.data2 IN MEMORY
CALL rec.data.readFile("myfile.jpg")
... Do SOAP Operation
CALL resp.data2.writeFile("retfile.jpg")
```

You can do this:

```plaintext
DEFINE req RECORD
    data STRING ATTRIBUTES(XMLOptimizedContent)
END RECORD
DEFINE resp RECORD
    data2 STRING ATTRIBUTES(XMLOptimizedContent)
END RECORD
LET rec.data = "myfile.jpg"
... Do SOAP Operation
handle returned file at resp.data2
```

Related information

https://www.w3.org/TR/soap12-mtom/

swaRef (SOAP with attachments using wsi:swaRef)

swaRef is a specific way for sending and receiving attachments in SOAP. It is used when you have to transfer files as attachment and locate on disk.

swaRef refers to "Soap with attachments using the wsi:swaRef XML type from WS-I". It is a specific case of the SoapWithAttachment specification.
Warning: Java handles swaRef automatically. Dot Net does not.

With swaRef, you can set an attribute with a dedicated value of "swaRef" on a STRING. When set, a Web service automatically sends the file in that string as an HTTP part or receives it as an HTTP part.

In the following example, a sendAttachment operation computes the MD5 of a file. The program sets XMLOptimizedContent="swaRef" on the DataIn string and the SOAP engine carries it over the wire as an HTTP part.

```plaintext
# Request Type
TYPE tsendAttachment RECORD
  ATTRIBUTES(XMLSequence,XSTypeName="sendAttachment",XSTypeNamespace="http://4js.com/services/swa")
    Name STRING ATTRIBUTES(XMLName="Name",XMLOptional),
    DataIn STRING ATTRIBUTES(XMLOptimizedContent="swaRef",XMLAttribute,
    XMLName="DataIn",XMLOptional)
END RECORD

# Response Type
TYPE tsendAttachmentResponse RECORD
  ATTRIBUTES(XMLSequence,XSTypeName="sendAttachmentResponse",XSTypeNamespace="http://4js.com/services/swa")
    return STRING ATTRIBUTES(XMLName="return",XMLOptional)
END RECORD

# VARIABLE : sendAttachment
DEFINE sendAttachment tsendAttachment
  ATTRIBUTES(XMLName="sendAttachment",XMLNamespace="http://4js.com/services/swa")

# VARIABLE : sendAttachmentResponse
DEFINE sendAttachmentResponse tsendAttachmentResponse
  ATTRIBUTES(XMLName="sendAttachmentResponse",XMLNamespace="http://4js.com/services/swa")

... # Publish Operation : sendAttachment
  LET operation = com.WebOperation.CreateDOCStyle("sendAttachment","sendAttachment",sendAttachment,sendAttachmentResponse)
  CALL service.publishOperation(operation,"")
...
```

Related concepts
XMLOptimizedContent on page 3552
Set on STRING or BYTE data type so that such string content represents a file on disk to be transmitted as base64 binary in SOAP via HTTP attachment.

Security
These topics cover security for Genero Web Services.

Encryption and authentication
A scenario involving a person (Georges) and his bank guides you through the concepts of secured communication, certificates, and certificate authorities.

Secured communications
Secured communications are important. If an application wants to send or receive messages from a financial, business, or personnel application on the web, it must be able to authenticate the origin of the message, ensure that
no malicious application has altered the original message, and ensure that no third party application can intercept the message.

Suppose that a person named Georges wants to send a message to his bank to transfer some money on the Internet. In this scenario, he faces the following concerns:

1. **The privacy** of the message, since it includes his account number and the transfer amount.
2. **The integrity** of the message, since someone might try to modify the original message or substitute a different message in order to transfer the money to another account.
3. **The authentication** of the message, since the bank must ensure that the message was sent from the right person.

**Message privacy**

To keep a message private, use a cryptographic algorithm - a technique that transforms a message into an encrypted form unreadable except by those it is intended for. Once it is in this form, the message may only be interpreted through the use of a secret key. There are two kinds of cryptography algorithms: symmetric and asymmetric.

**Symmetric** means the sender and the receiver of a message have to share the same key used to encrypt a clear message into an encrypted form, and then to decrypt it back into the original message. If that key is kept secret, nobody other than the sender and the receiver can read the message. However, the task of choosing a private key before communicating can be problematic.

**Asymmetric** means that there are two different keys working as a key-pair. One key is used to encrypt a message, and the second one is used to decrypt the encrypted message back into its original form. This solves the problem of key sharing in the symmetric cryptography algorithm, and makes it possible to receive secure messages, simply by publishing the key used to encrypt messages (the **public key**), and keeping secret the key used to decrypt messages (the **private key**). Anyone can encrypt a message using the public key, but only the owner of the private key can read it.

**Important:** The use of an asymmetric key-pair (public and private key), allows Georges to send private messages to his bank, simply by using the bank's public key to encrypt a message. Only the owner of the corresponding private key (the bank in this scenario) is able to read it.

**Message integrity**

To guarantee the integrity of a message, send a concise summary of the original message. The receiver of the message can create its own summary and compare it to the sender's summary. If they are similar, the message is considered intact, meaning that no third party has modified the original message.

Such a summary is called a **message digest** and is based on hash algorithms that produce a fixed-length representation of variable-length messages. Message digests are designed to make it very difficult (if not impossible) to determine the original message from a summary.

The message digest must be sent to the receiver in a secure way to assure the message integrity. This is achieved with a digital signature authenticating the sender and containing the sender's message digest.

**Important:** The use of message digests allows Georges' bank to verify that no one has modified the original message he sent.

**Message authentication**

To authenticate a message, add a digital signature to that message.

**A digital signature** is another message, created by encrypting the message digest, along with some other information, with the sender's private key. Anyone with the corresponding public key can decrypt the digital signature. If an application is able to decrypt it, it means the owner of the private key was able to encrypt it, proving that the message comes from this sender and not from someone else.

Once the sender has been authenticated, the receiver can compare the message digest integrated into the digital signature to the one it created from the message it receives, in order to check the message integrity.

**Important:** The use of digital signatures allows Georges' bank to verify that the message really comes from him.
Certificates

An SSL/TLS certificate is a kind of digital identity card that associates the public key with a unique digital thumbprint identifying an individual, a server, or any other entity.

Now that Georges is able to send a secured message to his bank, there is still a problem. How can Georges be sure that the server he is connected to is really the bank's server and not a malicious server?

Georges must be sure that the public key he is using to encrypt his message corresponds to the bank's private key. Similarly, the bank needs to verify that the message signature it receives corresponds to Georges' signature.

To identify a remote peer, use a **certificate** - a kind of digital identity card that associates the public key with a unique digital thumbprint identifying an individual, a server, or any other entity (known as the **subject**). It also includes the identification and signature of the Certificate Authority that issued the certificate, and the period of time during which the certificate is valid. It may have additional information (or extensions) as well as administrative information for the Certificate Authority's use, such as a serial number.

A standard X.509 certificate contains the following standard fields:

- Certificate version
- Serial number of the certificate
- The distinguished name of the certificate issuer
- The distinguished name of the certificate owner
- The validity period of the certificate
- The public key
- The digital signature of the issuer
- Signature algorithm used
- Zero or more certificate extensions

**Note:**

1. An example of a distinguished name is: **CN=Georges,E=georges@mycompany.com,OU=Sales,O=My Company Name,C=FR,S=France**
2. The **CN** (Common Name) of the distinguished name of the certificate owner corresponds to the certificate subject, and identifies the owner of that certificate.

Certificate authorities

When a certificate authority signs a certificate, it is validating that the certificate is valid.

Each time Georges sends a message to his bank, he will present his own certificate to the bank, and will get the bank's certificate back. But as every one can create a certificate in the name of Georges, a higher authority that confirms the validity of a certificate is necessary. The bank must be sure it is Georges' certificate, and that no one else has taken his identity. Similarly, Georges needs an authority that confirms that the certificate coming from the server is really the bank's certificate.

The solution to validating a certificate is to sign it with a trusted certificate called **certificate authority**. This is a certificate in which an application has total confidence concerning the validity of the certificates it has signed. Before signing a certificate, a certificate authority must proceed with a strict identification of the owner of that certificate.

**Note:** The private key associated to a Certificate Authority must be managed with care, as it is the entity in charge of the validity of all other certificates it has signed.

There are several companies (such as *VeriSign*, *GlobalSign* or *RSA Security*) that have established themselves as certificate authorities and provide the following services over the Internet:

- Verifying certificate requests
- Processing certificate requests
- Issuing and managing certificates

**Note:** It is also possible to create your own Certificate Authority, but it is up to you to manage it securely.
Root Certificate Authority

A Certificate Authority signed by itself is called a Root Certificate Authority, meaning that the certificate issuer is the same as the certificate subject. Most of the time, such a certificate belongs to a company established as a Certificate Authority, and is used to sign certificate requests coming from different companies that want their own Certificate Authority. If a client certificate is signed by a Certificate Authority previously signed by a Root Certificate Authority, the client certificate can be validated by the Root Certificate Authority even if the Certificate Authority is not present.

For example, if a company wants to buy a Certificate Authority from VeriSign, VeriSign signs that Certificate Authority with its own Root Certificate Authority. The company can then create certificates with the Certificate Authority provided by VeriSign and connect to secure servers without providing them with their own Certificate Authority. The secure server, of course, has to know the VeriSign Root Certificate Authority.

Certificate Chains

A certificate authority may issue a certificate for another certificate authority. This means that when an application wants to examine the certificate of the issuer, it must check all parent certificates of that issuer until it reaches one in which it has confidence.

The certificate chain corresponds to the number of parent certificate authorities allowed to validate a certificate.

Certificate Authority List

A Certificate Authority List is a list of all certificate authorities considered as trusted by one application, classified by order of importance. Each of these certificates allows the authentication of a certificate presented to that application from a remote peer.

Note: With most applications, the Certificate Authority List is a concatenated file of all certificate authorities.

Certificates and private keys storage

The entire concept of security is based on the publication of the public key, and the privacy of the associated private key. For maximum security, it is critical to restrict the access of the private key to the owner of the certificate and associated private key.

Note: Some companies provide systems to manage certificates and private keys in complete security.

UNIX® systems

As the UNIX™ system is already able to restrict the access of a file to only one person, simply restrict access to the private key to the owner of that key to achieve a good level of security. This provides enough security to allow a Genero Web Services client to perform secured communications in the name of the certificate and private key owner, because access to the private key file is granted only if the correct user has logged in.

Windows® systems

Windows® has an integrated key store system to manage certificates and private keys. It allows the registration and the storage of X.509 certificate authorities, as well as personal X.509 certificates and their associated private keys.
keys accessible only if the correct user has logged in. It is recommended that you store the certificate and associated private key in the Windows® key store instead of in files on the disk.

**Related concepts**

- [Missing certificates](#) on page 3311
- Identifying missing certificates.

**Error: Peer certificate is issued by a company not in our CA list** on page 3310

When a client connects to a server using HTTPS, the client needs to trust the server it is in communication with. So the client needs to add the server's CAs (certificate authorities lists) to its trusted CAs.

### Accessing secured services

Security and authentication are important. Genero Web Services provides various communications options for a client to connect to a Web service.

![Communications options for a client to connect to a Web Service](image)

- **HTTP**
  - Client connects to a Web Server (or a Web Service) using HTTP as the communication protocol. *(No security, No authentication).*

- **HTTP with Basic Authentication**
  - Client connects to a Web Server using HTTP as the communication protocol, but a valid login and password are required from the Web server to grant access to the Web Service. *(No security, Weak Authentication).* The login and password are sent in clear text on the communication layer.

- **HTTP with Digest Authentication**
  - Client connects to the Web Server using HTTP as the communication protocol, but a valid login and password are required from the Web server to grant access to the Web Service. *(No security, Authentication).* The login and password are encoded using a digest algorithm, requiring additional information from the Web server. This means that the first connection will always fail, but it is necessary in order to return Web server additional information back to the client.

- **HTTPS**
  - Client connects to a Web server using HTTPS as the communication protocol. *(Security, No authentication).* The communication channel is encrypted by SSL/TLS.

- **HTTPS with Basic Authentication**
  - Client connects to a Web server using HTTPS as the communication protocol, but a valid login and password are required from the Web server to grant access to the Web Service. *(Security, Weak Authentication).*
The login and password are sent in clear text on the communication layer, but the communication channel is encrypted by SSL/TLS.

**HTTPS with Digest Authentication**

Client connects to the Web Server using HTTPS as the communication protocol, but a valid login and password are required from the Web server to grant access to the Web Service. (Security, Authentication). The login and password are encoded using a digest algorithm, requiring additional information from the Web server. This means that the first connection will always fail, but it is necessary in order to return Web server additional information back to the client. The communication channel is encrypted by SSL/TLS.

To improve communication speed with the cache mechanism, or to restrict internet access to specific clients, Genero Web Services allows a client to connect via proxies. The proxy is in charge of dispatching the client request to the server, and uses the same protocol as that used by the server. So, when a client connects via a proxy to access a HTTP server, the configuration of the HTTP proxy is used, and when the client communicates in HTTPS, the HTTPS proxy configuration is used.

**HTTP proxy**

Client connects via a proxy using HTTP as the communication protocol.

**HTTP proxy with Basic Authentication**

Client connects via a proxy using HTTP as the communication protocol, but a valid login and password are required from the proxy to dispatch the request to the Web Service. The login and password are sent in clear text on the communication layer between client and proxy.

**HTTP proxy with Digest Authentication**

Client connects via a proxy using HTTP as the communication protocol, but a valid login and password are required from the proxy to dispatch the request to the Web Service. The login and password are encoded using a digest algorithm, requiring additional information from the proxy. This means that the first connection will always fail, but it is necessary in order to return proxy additional information back to the client.

**HTTPS proxy**

Client connects via a proxy using HTTPS as the communication protocol. The communication channel is encrypted by SSL/TLS.

**HTTPS proxy with Basic Authentication**

Client connects via a proxy using HTTPS as the communication protocol, but a valid login and password are required from the proxy to dispatch the request to the Web Service. The login and password are sent in clear text on the communication layer between client and proxy, but the communication channel is encrypted by SSL/TLS.

**HTTPS proxy with Digest Authentication**

Client connects via a proxy using HTTPS as the communication protocol, but a valid login and password are required from the proxy to dispatch the request to the Web Service. The login and password are encoded using a digest algorithm, requiring additional information from the proxy. This means that the first connection will always fail, but it is necessary in order to return proxy additional information back to the client. The
Web services

communication channel between client and proxy is encrypted by SSL/TLS.

Related reference
- Basic or digest HTTP authentication on page 3491
- Proxy configuration on page 3492

HTTPS configuration
If no client certificate is provided, Genero Web Services (GWS) does the HTTPS request transparently.

GWS can use an implicit certificate when no HTTPS configuration is provided. For stronger security, you can provide HTTPS configuration with your own certificates and CA list.

The implicit client certificate
For the implicit certificate, no configuration is required. GWS creates a temporary certificate for the HTTPS request. The temporary certificate is valid for the application session.

The explicit client certificate
For the explicit certificate, configure your certificate with entries for your HTTPS and password encryption on page 3489 settings of your FGLPROFILE file.

For access to a specific site, specify security.idsec.certificate and security.idsec.privatekey.

If you use the same certificate across all sites, specify security.global.certificate and security.global.privatekey.

Certificate authorities
Certificate authorities (file extension .crt) are usually provided by the system (the operating system keystore). If they are not provided by the system, the certificate authorities are searched for in the following locations:

1. The $FGLDIR/web_utilities/certs directory
2. The file specified by the FGLPROFILE file entry security.global.ca
3. The keystore specified by the FGLPROFILE file entry security.global.windowsca (Windows® systems) or security.global.systemca (all systems).
4. The directories listed in the FGLPROFILE file entry security.global.ca.lookuppath

Mobile platforms
On mobile platforms, no HTTPS configuration is required, because the Web Service library uses the SSL/TLS certificates installed in the key database of the device (Keystore for Android™ and Keychain® for iOS).

See also Web Services on GMA (Android) on page 3277.

Related concepts
- FGLPROFILE entries for Web Services on page 3489

The FGLPROFILE entries relating to Genero Web Services are divided between five categories: security, basic or digest HTTP authentication, proxy configuration, server configuration, and XML cryptography.

Configure a WS client to access an HTTPS server
Configuration steps to access a server in HTTPS.

To configure access to an HTTPS server, you need a client certificate.

Important: On GMI mobile devices the iOS SSL/TLS layer is used for HTTPS, and the device Keychain® typically holds the server certificate authority list. Therefore, the FGLPROFILE file security settings are ignored with the exception of the following: security.global.ca, security.global.ca.lookuppath, and security.global.systemca.
Before you begin, there are options to consider depending on how you wish to use the client certificate:

- If you do not have the certificate information in your FGLPROFILE file, Genero Web Services creates a certificate for you. This is an implicit or temporary certificate that is valid for a session only. For more information see HTTPS configuration on page 3305.

  **Note:** For the implicit certificate, no configuration is required.

- Alternatively, for stronger security, you generate a client certificate of your own, configure your application to use the client certificate generated, and add the configuration details to the FGLPROFILE file. Follow the steps outlined in this section.

  **Note:** In a production environment, some servers provide a client certificate and you use the certificate as provided, and add the configuration details to the FGLPROFILE file.

The **openssl command line tool** can be used to create your own certificates for the configuration of secured communication. The following steps outline the configuration process:

**Related concepts**
- Configuring the Apache Web server for HTTPS on page 3460
- Configuration steps to secure a Web service for Apache server in HTTPS.
- Certificates in practice on page 3313
- Procedures and tools for creating, importing, and viewing certificates and keys.
- Encryption and authentication on page 3299
- A scenario involving a person (Georges) and his bank guides you through the concepts of secured communication, certificates, and certificate authorities.
- Accessing secured services on page 3303
- Security and authentication are important. Genero Web Services provides various communications options for a client to connect to a Web service.

**Create the client certificate**
Generate a root certificate authority that signs a client certificate.

In this task you create your own root certificate authority and client certificate using the OpenSSL command line tool.

1. Create the root certificate authority.

   a) Create the root certificate authority serial file.
   
   ```bash
   $ echo 01 > MyCompanyCA.srl
   ```

   This command creates a serial file with an initial HEX value 01. OpenSSL uses this file to track the serial numbers of certificates it creates. The serial file is typically given the same name as the CA with the extension .srl.

   b) Create the Certificate Signing Request (CSR) and private key.
   
   ```bash
   $ openssl req -new -out MyCompanyCA.csr -keyout MyCompanyCA.pem
   ```

   Follow the instructions to create the CSR.

   c) Create the Root Certificate Authority for a period of validity of 2 years.

   ```bash
   $ openssl x509 -trustout -in MyCompanyCA.csr -out MyCompanyCA.crt -req -signkey MyCompanyCA.pem -days 730
   ```

   **Note:** The private key file (MyCompanyCA.pem) of a Root Certificate Authority must be handled with care. This file is responsible for the validity of all other certificates it has signed. As a result, it must not be accessible by other users.

2. Create the client's X.509 certificate and private key.

   a) Create the client's Certificate Signing Request and private key.

   ```bash
   $ openssl req -new -out MyClientCA.csr -keyout MyClientCA.pem
   ```

   Follow the instructions to create the CSR.
$ openssl req -new -out MyClient.csr

**Note:** By default, openssl outputs the private key in the `privkey.pem` file. If you want to specify a different file name, or if your openssl version does not output the private key by default, add the `-keyout <myprivkey>.pem` to the command.

b) Remove the password from the RSA private key.

```
$ openssl rsa -in privkey.pem -out MyClient.pem
```

**Note:** The unprotected private key is output in `MyClient.pem`.

c) Create the client's certificate (self-signed X.509 certificate valid for a period of 1 year) trusted by the Root Certificate Authority created in step 1.

```
$ openssl x509 -in MyClient.csr -out MyClient.crt -req
-signtype MyClient.pem -CA MyCompanyCA.crt
-CAkey MyCompanyCA.pem -days 365
```

**Note:** Most servers do not check the identity of the clients. For these servers, the client's certificate does not necessarily need to be trusted; it is only used for data encryption purpose. If, however, the server performs client identification, you must trust a Certificate Authority in which it has total confidence concerning the validity of the client's certificates.

**Note:** The purpose of the client's certificate is to identify the client to any server; therefore the subject of the certificate must correspond to the client's identity as it is known by the servers.

**Note:** To import the certificate in a keystore you can create a pkcs12 certificate. See Import a certificate and its private key into the Windows key store on page 3315.

**What to do next**

Configure your FGLPROFILE file for the client certificate. See Configure for the client certificate on page 3307.

**Related concepts**

Encryption and authentication on page 3299
A scenario involving a person (Georges) and his bank guides you through the concepts of secured communication, certificates, and certificate authorities.

The OpenSSL (openssl) tool on page 3313
The openssl command line tool creates certificates for the configuration of secured communications.

**Related tasks**

View a certificate on page 3316
This procedure allows you to view a certificate using the openssl command.

**Configure for the client certificate**

Configure your application to use the client certificate generated. For production systems, you add the configuration details to your FGLPROFILE file.

You have generated a client certificate as described in Create the client certificate on page 3306.

In this task you add the certificate information to your FGLPROFILE file.

Add the client's security configuration to your FGLPROFILE file.

The client security entry defines the certificate and the associated private key used by the Genero Web Services client during an HTTPS communication. The security entry must be defined with an unique identifier (id1 in this example).

```
security.id1.certificate = "MyClient.crt"
security.id1.privatekey = "MyClient.pem"
```
Note: If the private key is protected with a password, you must remove it or create a script that returns the password on demand. See Create the client certificate.

Your applications are configured to use the client certificate.

What to do next

Create the client's certificate authority list and configure the global certificate authority list (security.global.ca) in your FGLPROFILE file. See Configure for the certificate authority list on page 3308.

Related concepts

FGLPROFILE entries for Web Services on page 3489
The FGLPROFILE entries relating to Genero Web Services are divided between five categories: security, basic or digest HTTP authentication, proxy configuration, server configuration, and XML cryptography.

FGLPROFILE: HTTP(S) Proxy Authentication on page 3496
FGLPROFILE entries can be used to define a connection to an HTTPS server via a proxy, and with HTTP and Proxy Authentication.

Encryption and authentication on page 3299
A scenario involving a person (Georges) and his bank guides you through the concepts of secured communication, certificates, and certificate authorities.

Related tasks

View a certificate on page 3316
This procedure allows you to view a certificate using the openssl command.

Configure for the certificate authority list

The client needs to check to see if the server certificate is trusted. This is done using a certificate authority list.

In this task you create the client's certificate authority list using the OpenSSL command line tool, and set the global certificate authority list in your FGLPROFILE file.

Important: On GMI mobile devices the iOS SSL/TLS layer is used for HTTPS, and the device Keychain® typically holds the server certificate authority list. Therefore, the FGLPROFILE file security settings are ignored with the exception of the following: security.global.ca, security.global.ca.lookuppath, and security.global.systemca.

1. Create the client's certificate authority list.
   a) Save the certificates of the HTTPS server to disk.
      Type the server's URL in your browser. When prompted, save all the certificates from the Certificate Hierarchy. For more information see, Selecting the certificate to add on page 3317 and Missing certificates on page 3311.
   b) Create the client's Certificate Authority List by running the following command for each of the certificates that you saved to disk.

      $ openssl x509 -in ServerCertificate.crt -text >> ClientCAList.pem

      Note: All trusted certificate authorities are listed. These are checked following a chain of child to parent certificates until a certificate is reached which is trusted. All other certificates that were trusted by the Root Certificate Authority will also be considered as trusted by the client. For more information see Certificate authorities on page 3301.

2. Set the global certificate authority list in your FGLPROFILE file.

   The global certificate authority list entry defines the file containing the certificate authority list used by the Genero Web Services client to validate all certificates coming from the different servers it will connect to.

   security.global.ca = "ClientCAList.pem"
**Note:** The file is located based on the current execution directory. If you use Genero Studio, for instance, *fglrun* may not be executed in the same directory as when you use the command line. The recommended practice therefore is to specify an absolute path for the .pem, for example:

```plaintext
security.global.ca = "/opt/usr/certs/ClientCAList.pem"
```

If `security.global.ca` is not defined, Genero Web Services will look to see whether the operating system has a keystore, otherwise `security.global.ca.lookuppath` will be used. For further information, see Certificate authorities in HTTPS configuration on page 3305.

The client application is configured to use the appropriate certificate authority list to validate a server's certificate.

**What to do next**

In your FGLPROFILE file ensure there are configuration entries (ws.*) for the HTTPS server URL and for the HTTP authentication when accessing the HTTPS server. See Add configuration entries in your FGLPROFILE file on page 3309.

**Related concepts**

**Error: Peer certificate is issued by a company not in our CA list** on page 3310

When a client connects to a server using HTTPS, the client needs to trust the server it is in communication with. So the client needs to add the server's CAs (certificate authorities lists) to its trusted CAs.

**Enable OCSP** on page 3310

To enable OCSP (Online Certificate Status Protocol), set the `security.global.ocsp.enable` and `security.global.ocsp.url` entries in FGLPROFILE.

**FGLPROFILE entries for Web Services** on page 3489

The FGLPROFILE entries relating to Genero Web Services are divided between five categories: security, basic or digest HTTP authentication, proxy configuration, server configuration, and XML cryptography.

**FGLPROFILE: HTTP(S) Proxy Authentication** on page 3496

FGLPROFILE entries can be used to define a connection to an HTTPS server via a proxy, and with HTTP and Proxy Authentication.

**Accessing secured services** on page 3303

Security and authentication are important. Genero Web Services provides various communications options for a client to connect to a Web service.

**Add configuration entries in your FGLPROFILE file**

Add a set of configuration entries that specify the security configuration for GWS clients accessing an HTTPS server.

In this task you add configuration entries (ws.*) in your FGLPROFILE file for the HTTPS server URL and for HTTP authentication when accessing the HTTPS server. For an example, see FGLPROFILE: HTTP(S) Proxy Authentication on page 3496.

**Important:** On GMI mobile devices the iOS SSL/TLS layer is used for HTTPS, and the device Keychain typically holds the server certificate authority list. Therefore, the FGLPROFILE file security settings are ignored with the exception of the following: `security.global.ca`, `security.global.ca.lookuppath`, and `security.global.systemca`.

Add configuration entries for the server.

The following entries must be defined with an unique identifier (such as "myserver"):


   (Line breaks added for document readability)

b) `ws.myserver.security = "id1"

   Where the value (id1 in this example) must match the unique identifier defined by the client security entry created in Configure for the client certificate on page 3307.

**Tip:** The unique identifier "myserver" can be used in the BDL client code in place of the actual URL.
Related concepts
FGLPROFILE entries for Web Services on page 3489
The FGLPROFILE entries relating to Genero Web Services are divided between five categories: security, basic or digest HTTP authentication, proxy configuration, server configuration, and XML cryptography.

Accessing secured services on page 3303
Security and authentication are important. Genero Web Services provides various communications options for a client to connect to a Web service.

Troubleshoot certificate issues on page 3310
You may encounter known (and common) issues when completing the Genero Web Services tutorials or when adding Web services of your own. These issues and their solutions are presented in the following topics.

Enable OCSP
To enable OCSP (Online Certificate Status Protocol), set the `security.global.ocsp.enable` and `security.global.ocsp.url` entries in FGLPROFILE.

When these options are set, for each HTTPS connection, once the X509 certificate has been validated, the Web service will check whether all certificates used for that validation are still valid and have not been revoked at the time of the connection.

Related concepts
FGLPROFILE entries for Web Services on page 3489
The FGLPROFILE entries relating to Genero Web Services are divided between five categories: security, basic or digest HTTP authentication, proxy configuration, server configuration, and XML cryptography.

Troubleshoot certificate issues
You may encounter known (and common) issues when completing the Genero Web Services tutorials or when adding Web services of your own. These issues and their solutions are presented in the following topics.

Error: Peer certificate is issued by a company not in our CA list
When a client connects to a server using HTTPS, the client needs to trust the server it is in communication with. So the client needs to add the server's CAs (certificate authorities lists) to its trusted CAs.

This error means the client CA list is missing a certificate authority in its CA list.

To display the client CA list, use the following command:

```
openssl x509 -in ClientCAList.pem -noout -text
```

Solution:
1. Add the missing CA list to the client CA list.

```
openssl x509 -in MyCompanyCA.crt -text >> ClientCAList.pem
```

Theory
Usually certificates work in pairs: a public key and a private key.
Figure 122: Certificates working in pairs: a public key and a private key

The certificate which appears at the top of the list on the client and server side is the Certificate Authority (CA) root. This means that the client has a certificate that can be signed by an authority signed itself by a root authority. Likewise, the server has a certificate that can be signed by an authority signed itself by a root authority. In some instances, a certificate can be signed by itself.

Things to note:

- The server certificate is expected to have its host name set as CN (Common Name). For example, if you want to access the server https://www.mycompany.com the CN is expected to be "www.mycompany.com".
- In the client CA list it is recommended that you have all the CA of the server. In this example you need the server CA (5) and the server CA Root (4). If the server is self-signed then add the server certificate (6) to the client CA list.
- Sometimes, the needed CAs are not listed in the certificates hierarchy. Setting environment variable FGLWSDEBUG=3, will give you information about the missing CA.

Related concepts

Encryption and authentication on page 3299
A scenario involving a person (Georges) and his bank guides you through the concepts of secured communication, certificates, and certificate authorities.

FGLWSDEBUG on page 283
The FGLWSDEBUG environment variable enables web services library debugging.

Related tasks

Configure for the certificate authority list on page 3308
The client needs to check to see if the server certificate is trusted. This is done using a certificate authority list.

Missing certificates
Identifying missing certificates.

Sometimes the CA hierarchy described in the server certificate is incomplete or needs another certificate (default ones used by browsers or private ones).
Figure 123: Certificate Viewer in Firefox Web Browser; Details Tab
When this occurs, you will get this kind of error message when you set FGLWSDEBUG:

```
WS-DEBUG (Security error)
Error with certificate at depth: 3
issuer = /C=US/O=VeriSign, Inc./OU=Class 3 Public Primary Certification Authority
subject = /C=US/O=VeriSign, Inc./OU=Class 3 Public Primary Certification Authority
err 19:self signed certificate in certificate chain
WS-DEBUG END
```

This means OpenSSL is looking for a third ancestor that is not listed in the hierarchy above. In this example, gatewaybeta.fedex.com only has two ancestors, and none are named "Class 3 Public Primary Certification Authority". You need to download the root certificates from VeriSign and add "Class 3 Public Primary Certification Authority" in your CA list.

If the certificate authorities are not found in the operating system keystore, you need to download them:

- If you store the CA certificates in $FGLDIR/web_utilities/certs directory, ensure they are named with the .crt extension.
- If you have configured the FGLPROFILE file using the global certificate authority list entry security.global.ca, you need to create the CA list with the .pem extension.

The Firefox browser allows you to download the CA list as a pem file; a PEM (cert) or PEM (chain). Use the PEM (chain) file in this case. Otherwise, you will need to download the certs individually and create the pem chain file as described in Create a certificate authority list on page 3315.

**Related concepts**

Error: Peer certificate is issued by a company not in our CA list on page 3310

When a client connects to a server using HTTPS, the client needs to trust the server it is in communication with. So the client needs to add the server's CAs (certificate authorities lists) to its trusted CAs.

**The OpenSSL (openssl) tool** on page 3313

The openssl command line tool creates certificates for the configuration of secured communications.

Certificates in practice

Procedures and tools for creating, importing, and viewing certificates and keys.

**The OpenSSL (openssl) tool**

The openssl command line tool creates certificates for the configuration of secured communications.

OpenSSL is provided in the FGLGWS package. To use it, you need to add its directory, $FGLDIR/web_utilities/tools, to your PATH environment variable.

The openssl tool has a default configuration file, openssl.cnf. It looks for the openssl.cnf file in the directory where it is executed; it stops if the file is not present. To use the openssl tool from any directory, set the OPENSSL_CONF environment variable to specify the location of the configuration file.

For information on how the openssl tool works, refer to the openssl documentation.

Create a root certificate authority

Generate a root certificate authority that signs a client certificate.

In this task you use the openssl command tool to create the root certificate authority.

1. Create the root certificate authority serial file:

```
$ echo 01 > MyRootCA.srl
```

This command creates a serial file with an initial HEX value 01. OpenSSL uses this file to track the serial numbers of certificates it creates. The serial file is typically given the same name as the root CA with the extension .srl.
2. Create a Certificate Signing Request (CSR):

   $ openssl req -new -out MyRootCA.csr

   Follow the instructions to create the CSR. Two files are created, the MyRootCA.csr and a file called privkey.pem, which contains the RSA private key of the CSR certificate protected by a password.

3. Remove the password of the private key (Optional):

   $ openssl rsa -in privkey.pem -out MyRootCA.pem

   Note: Removing the password of a certificate authority's private key is not recommended.

4. Create a self-signed certificate from the CSR and the unprotected private key for a validity period of 365 days:

   $ openssl x509 -trustout -in MyRootCA.csr -out MyRootCA.crt -req -signkey MyRootCA.pem -days 365

   Note: About the CSR and its private key:
   • If you want an official Certificate Authority, you must send the CSR file to one of the self-established Certificate Authority companies on the Internet (instead of creating it with openssl. See Encryption and authentication on page 3299).
   • The CSR file is also used to encrypt messages that only its corresponding private key can decrypt.

Related tasks
Create a certificate authority on page 3314
This procedure allows you to create a certificate authority.

Configure a WS client to access an HTTPS server on page 3305
Configuration steps to access a server in HTTPS.

Create a certificate authority
This procedure allows you to create a certificate authority.

In this task you use the openssl command tool to create the certificate authority.

1. Create the certificate authority serial file:

   $ echo 01 > MyCA.srl

   This command creates a serial file with an initial HEX value 01. OpenSSL uses this file to track the serial numbers of certificates it creates. The serial file is typically given the same name as the CA with the extension .srl.

2. Create a certificate signing request (CSR):

   $ openssl req -new -out MyCA.csr

   Follow the instructions to create the CSR. This command also creates a private-key file, privkey.pem, containing the RSA private key of the CSR certificate and protected by a password.

3. Remove the private key password (Optional):

   $ openssl rsa -in privkey.pem -out MyCA.pem

   Note: Removing the password of a certificate authority's private key is not recommended.

4. Create a certificate from the Certificate Signing Request that is trusted by the Root Certificate Authority:

   $ openssl x509 -in MyCA.csr -out MyCA.crt -req -signkey MyCA.pem -CA MyRootCA.crt -CAkey MyRootCA.pem -days 365
**Note:** About the CSR and its private key:

- If you want an official Certificate Authority, you must send the CSR file to one of the self-established Certificate Authority companies on the Internet (instead of creating it with OpenSSL. See Encryption and authentication on page 3299).
- The CSR file is also used to encrypt messages that only its corresponding private key can decrypt.

**Related tasks**

Create a root certificate authority on page 3313
Generate a root certificate authority that signs a client certificate.

**Create a certificate**

This procedure allows you to create a certificate.

In this procedure you use the OpenSSL tool to perform the certificate creation tasks.

1. Create a Certificate Signing Request (CSR):

   ```
   $ openssl req -new -out MyCert.csr
   ```

   Follow the instructions to create the CSR. This command also creates a `privkey.pem` file containing the RSA private key of the CSR certificate that is protected by a password you provide.

2. Remove the private key password (Optional):

   ```
   $ openssl rsa -in privkey.pem -out MyCert.pem
   ```

   **Note:** The unprotected private key is output in `MyCert.pem`.

3. Create a certificate from the CSR that is trusted by the Certificate Authority:

   ```
   $ openssl x509 -in MyCert.csr -out MyCert.crt -req -signkey MyCert.pem -CA MyCA.crt -CAkey MyCA.pem -days 365
   ```

   **Note:** About the CSR and its private key:

   - If you want an official Certificate Authority, you must send the CSR file to one of the self-established Certificate Authority companies on the Internet (instead of creating it with OpenSSL. See Encryption and authentication on page 3299).
   - The CSR file is also used to encrypt messages that only its corresponding private key can decrypt.

**Related tasks**

Configure a WS client to access an HTTPS server on page 3305
Configuration steps to access a server in HTTPS.

**Create a certificate authority list**

This procedure allows you to create a certificate authority list using the OpenSSL command.

Concatenate all certificate authorities by order of importance, listing the most important first:

```
$ openssl x509 -in MyCA1.crt -text >> CAList.pem
$ openssl x509 -in MyCA2.crt -text >> CAList.pem
$ openssl x509 -in MyCA3.crt -text >> CAList.pem
```

**Related tasks**

Configure a WS client to access an HTTPS server on page 3305
Configuration steps to access a server in HTTPS.

**Import a certificate and its private key into the Windows® key store**

This procedure allows you to import a certificate and its private key.

1. Create a certificate.
See Create a certificate.

2. Create a specific PKCS12 file containing the certificate and its private key in one file using the following openssl command:

```bash
openssl pkcs12 -export -inkey MyCert.pem -in MyCert.crt -out MyCert.p12
```

**Note:** The .p12 generated file is protected by a password and can then be transported without any risk.

3. On a Windows® system, open this .p12 file and follow the instructions provided.

**Note:** If you select strong verification during the importation process, a pop-up displays each time an application accesses the private key asking the user whether the application is allowed to use it.

**Import a certificate authority into the Windows® key store**

This procedure allows you to import a certificate authority.

1. Create a certificate authority.

   See Create a certificate Authority.

2. Open the .crt certificate file

3. Click Install Certificate and follow the instructions provided.

   Windows® automatically places the certificate in the certificate authority list of the key store.

**Related tasks**

Import a certificate and its private key into the Windows key store on page 3315

This procedure allows you to import a certificate and its private key.

**View a certificate**

This procedure allows you to view a certificate using the openssl command.

To view a certificate, enter the x509 command:

```bash
openssl x509 -in MyCompanyCA.crt -noout -text
```

**Related concepts**

Troubleshoot certificate issues on page 3310

You may encounter known (and common) issues when completing the Genero Web Services tutorials or when adding Web services of your own. These issues and their solutions are presented in the following topics.

**Examining certificates**

When you receive a URL in HTTPS, you are prompted to either accept a certificate or the certificate has already been accepted. In the second case, you can still check the server certificate.

**Check the server certificate using FireFox**

This procedure allows you to check the server certificate using FireFox.

1. Type the HTTPS URL.
2. Once the page is displayed, click on the padlock icon.
   The Page Info for the certificate displays.
3. In the Security tab, click on the View button.
   The Certificate Viewer opens.
4. In the Details tab, view the Certificate Hierarchy.
Check the server certificate using Internet Explorer
This procedure allows you to check the server certificate using Internet Explorer.

1. Type the HTTPS URL.
2. Once the page is displayed, click on the padlock icon.
   The Certificate window displays.
3. On the Certification Path, view the certificate hierarchy.

Selecting the certificate to add
The certificate authority (CA) is the authority that validates the server. The certificate to add to the CA list is the authority certificate, not the server certificate.

There are default certificates known by browsers like:

Get the server issuer certificate (and all the parents, grandparents, and so on).

For example, if your server is validated by Thawte, add the Thawte certificate to the list.

To check whether your certificate is the CA certificate, search for the CNs (Common Names) in the .cer files.
The CA Subject entry should be the Issuer CN in the server certificate. Running the openssl command as follows outputs the CN:

```
openssl x509 -in server.pem -noout -subject
```

```bash
subject=/C=ZA/ST=Western Cape/L=Cape Town/O=Thawte Consulting cc/OU=CertificationServices Division/CN=Thawte Server CA /emailAddress=server-certs@thawte.com
```
To convert a .cer certificate to the .pem format used by Genero Web Services run the following OpenSSL command:

```
openssl x509 -inform DER -in server.cer -outform PEM -out server.crt
```

Troubleshoot common issues

You may encounter known (and common) issues when completing the Genero Web Services tutorials or when adding Web services of your own. These issues and their solutions are presented in the following topics.

HTTP 401 error message

An HTTP 401 error message means the server is requesting, but not receiving, user authentication (login and password).

This error message means authenticate.xxx.login and authenticate.xxx.password are not correctly configured. The login and password should be provided in your FGLPROFILE file.

Solution:

1. Open the FGLPROFILE file used by the application.
2. Add entries for authenticate.xxx.login and authenticate.xxx.password.
3. Save your changes.

Related concepts

- Web Services FGLPROFILE configuration on page 3489
- The configuration for the Genero Web Services is defined from entries in the FGLPROFILE file.
- Encryption, BASE64 and password agent with fglpass tool on page 3290
- Genero Web Services supports password encryption with fglpass as password agent.

The Diffie-Hellman key agreement algorithm

Understand how Genero Web Services supports different key encryption methods for shared secret communication using the Diffie-Hellman key-agreement algorithm and the GWS XML security classes.

The Diffie-Hellman key agreement algorithm is a method that allows two devices to communicate over a network by establishing a shared secret without exchanging any secret data. Knowing the used key-agreement algorithm, the two devices only need to exchange their public keys. Then, using the other peer's public key and its own private key, each device performs the algorithm specific key generation operation to obtain the shared secret. The shared secret is a ready-to-use symmetric key for further signed or encrypted exchanges between the two peers.

Genero Web Services provides several shared secret type for signature, encryption, or key encryption purposes. Using the Diffie-Hellman key agreement algorithm, one of the following types of shared secrets can be computed:

- Symmetric AES128 encryption key
- Symmetric AES192 encryption key
- Symmetric AES256 encryption key
- Symmetric TripleDES encryption key
- Symmetric key wrap AES128 key encryption key
- Symmetric key wrap AES192 key encryption key
- Symmetric key wrap AES256 key encryption key
- Symmetric key wrap TripleDES key encryption key
- Symmetric HMAC-SHA1 signature key

In the Diffie-Hellman key agreement algorithm, two shared constants (called parameters) are used in addition to the private and public key. These two parameters are:

- The modulus (called P): A very big prime number chosen at random.
- The generator (called g): A prime number between two and five. Genero Web Services only uses two (2) for the generator.
If the private key (Priv) is a big number (not necessarily prime) chosen randomly, the public key (Pub) is calculated using P, g, and Priv as follows:

\[ \text{Pub} = g^{\text{Priv}} \mod P \]

Both devices need to use the same parameters for P and g. There are two ways to ensure this happens: Either P and g are chosen by a third party (such as a security authority) or one of the devices chooses them and sends them to the other peer with its public key.

Genero Web Services allows the Web service to generate the parameters itself, to load them from a string or from a PEM or DER file. The public key and the parameters can also be exchanged using an XML file.

This diagram shows the Diffie-Hellman algorithm steps between two devices, A and B, that need to communicate. Device A is in charge of generating the parameters. The shared secret is labeled K.
Figure 125: The Diffie-Hellman algorithm

For complete details about the mathematical basics underlying the Diffie-Hellman algorithm, see [RFC2631].
It is nearly impossible to get the private key from the public key, even knowing the values of parameters \( P \) and \( g \). Therefore, a middle man will not be able to obtain the shared secret \( K \). While devices \( A \) and \( B \) exchange their public key, and maybe the parameters as well, these values pass through different intermediate points. It is critical that \( A \) receives the correct public key from \( B \), and that \( B \) receives the correct public key from \( A \), in order to establish a common shared secret. It may be possible for a middle man to corrupt or replace one public key with his own. If that happens, \( A \) and \( B \) would be able to communicate because they don't compute the same shared secret. No secret data will be exchanged that is readable to the middle man. To avoid this situation, one can use Digital Certificate that helps to deliver the public key and the parameters in an authenticated method.

Once the shared secret is established, the Diffie-Hellman public key, private key and parameters are no longer useful. The Diffie-Hellman key agreement algorithm is achieved.

With the library provided as part of Genero Web Services, the shared secret has been computed to fit given specifications such as HMAC, 3DES, AES128, AES192, AES256, KW-3DES, KW-AES128, KW-AES192, or KW-AES256. The shared secret is actually a symmetric key ready to be used in a signature (HMAC) or cipher algorithm. It allows devices \( A \) and \( B \) to finally communicate via an authenticated (HMAC) or encrypted method.

**Related concepts**

XML security classes on page 3047
XML Security classes handle encryption and signature of XML documents entirely in memory with keys and certificates.

**Related reference**

Supported kind of keys on page 3061
Types of keys supported by the `xml.CryptoKey` class.

---

**SOAP Web Services**

Write a Web Services client or server using the SOAP protocol.

The Genero APIs for creating Web services can be found in the Library section of this manual. See The com package on page 2811, The xml package on page 2931, and The util package on page 2726.

**Writing a Web Services client application**

Create, configure and deploy a Genero Web Services client using the SOAP protocol.

**Steps to write a WS client**

To access a SOAP Web service, you create a Genero BDL client application that requests the service's functions.

The Genero Web Services (GWS) package allows a BDL program to access Web services found on the Internet. GWS supports the WSDL1.1 specification of March 15, 2002. This example illustrates a client application that accesses the Add operation in the GWS Web Service `MyCalculator`.

See Writing a Web server application on page 3348 for information about the Service.

**Obtaining the WSDL information**

Before accessing a SOAP Web service, you must get the WSDL information from the service provider.

Sample services can be found through UDDI registries or on other sites such as XMethods (http://www.xmethods.net).

You can use the `fglwsdl` tool provided by the Genero Web Services package to obtain the necessary WSDL information. The following example obtains the WSDL information for the GWS Service `MyCalculator` created by the Writing a Web server application on page 3348:

```
fglwsdl -soap12 -o Example2Client http://localhost:8090/MyCalculator?WSDL
```

This generates two files:
- Example2Client.inc - the globals file containing the definitions of the input and output records, and the prototypes of the operations.
- Example2Client.4gl - a module containing the definitions of the functions that can be used in your GWS client application to perform the requested Web Service operation, and the code that manages the Web Service request.

Note: The MyCalculator GWS Service must be running on the specified port in order to provide the WSDL information.

The following definitions were generated in the globals file, Example2Client.inc:

### Input and Output records

```plaintext
DEFINE Add RECORD
  ATTRIBUTES( XMLName="Add",
    XMLNamespace="http://tempuri.org/webservices" )
  a INTEGER ATTRIBUTES( XMLName="a", XMLNamespace="" ),
  b INTEGER ATTRIBUTES( XMLName="b", XMLNamespace="" )
END RECORD

DEFINE AddResponse RECORD
  ATTRIBUTES( XMLName="AddResponse",
    XMLNamespace="http://tempuri.org/webservices" )
  r INTEGER ATTRIBUTES( XMLName="r",XMLNamespace="" )
END RECORD
```

The data types are defined as global or modular variables.

### Function prototypes for the Operations

This globals file contains the prototype of two functions for the Add operation.

The Add function uses input and output parameters, and returns the status and result. This function can only be used if the input and output parameters are not complex structures such as arrays or records. Using this function, developers do not access the global records directly.

The Add_g function can be used with the global input and output records. Before calling this function, you must set the values in the variables of the global input record.

```plaintext
Operation: Add
  #
  # FUNCTION: Add_g()
  # RETURNING: soapStatus
  # INPUT: GLOBAL Add
  # OUTPUT: GLOBAL AddResponse
  #
  # FUNCTION: Add(p_a, p_b)
  # RETURNING: soapStatus ,p_r
```

See fglwsdl on page 2081 and WS client stubs and handlers on page 3328 for more details regarding the fglwsdl tool, its output and the generated functions.

### Related concepts

SOAP 1.1 and 1.2 on page 3280
GWS is able to communicate with Web services provided with SOAP 1.1 or SOAP 1.2.

**Calling a web service**
Import the required libraries, specify globals, and write the program to call a SOAP Web Service.

**Step 1: Import the COM library of the GWS package**
The methods associated with creating and publishing a Web Service are contained in the classes that make up the Genero Web Services Library (com). If you use any of these methods in your client application, you must import the library. Since this example application sets the timeout period that the client will wait for the Service to respond, include the following line at the top of the module:

```plaintext
IMPORT com
```

If your generated .inc file uses XML class data types, you need to add `IMPORT xml.`

**Related concepts**
- The [com package](#) on page 2811
  The Genero Web Services com package provides classes and methods that allow you to perform tasks associated with creating Services and Clients, and managing the services.

**Step 2: Specify the globals file**
Use a GLOBALS statement to specify the globals file generated from the WSDL.

```plaintext
GLOBALS "Example2Client.inc"
```

**Related concepts**
- [Obtaining the WSDL information](#) on page 3321
  Before accessing a SOAP Web service, you must get the WSDL information from the service provider.

**Step 3: Write the MAIN program block**
Provide values for the input and output messages of the operation, and call one of the generated functions. Since the input and output messages are simple integers, we can call the Add function.

```plaintext
MAIN
  DEFINE op1 INTEGER
  DEFINE op2 INTEGER
  DEFINE result INTEGER
  DEFINE wsstatus INTEGER

  LET op1 = 1
  LET op2 = 2
  CALL Add(op1, op2) RETURNING wsstatus, result
  IF wsstatus = 0 THEN
    DISPLAY "Result: ", result
  ELSE
    -- Use the global wsError record
    DISPLAY "Error: ", wsError.description
  END IF
END MAIN
```

Alternatively, we can use the global input and output records directly, calling the Add_g function:

```plaintext
MAIN
  DEFINE wsstatus INTEGER

  LET Add.a = 1
  LET Add.b = 2
  LET wsstatus = Add_g()
  IF wsstatus != 0 THEN
```
-- Use the global wsError record
DISPLAY "Error ", wsError.Description
ELSE
  DISPLAY "Result: ", AddResponse.r
END IF
END MAIN

These examples are very basic versions of the code. For complete examples, see the code samples provided with the package in demo/WebServices.

Setting a time period for the response
To protect against remote server failure or unavailability, set a timeout value that indicates how long you are willing to wait for the server to respond to your request.

Use the SetOption() method of the WebServiceEngine class to set the readwritetimeout option.

For example, to wait no more than 10 seconds:

```
CALL com.WebServiceEngine.SetOption( "readwritetimeout", 10 )
```

A timeout value of -1 means "wait forever". This is the default value.

Related concepts
The WebServiceEngine class on page 2828
The com.WebServiceEngine class provides an interface to manage the Web Services engine.

Handling GWS server errors
When a Genero Web Services service operation returns a status that is non-zero, you can get a more detailed error description from the global record wsError.

This record is defined in the globals .inc file.

```
DEFINE wsError RECORD
  code STRING, -- Short description of the error
  codeNS STRING, -- The namespace of the error code
  description STRING, -- Long description of the error
  action STRING -- internal "SOAP action"
END RECORD
```

Compiling the client application
When compiling, remember to link in the WSHelper library.

It is recommended that the library file WSHelper.42m, included in the $FGLDIR/lib directory of the Genero Web Services package, is linked into every client or server program.

Assuming the example client code is in a module named clientmain.4gl, you can compile and link the client program:

```
fglcomp clientmain.4gl
fglcomp Example2Client.4gl
fgllink -o myclient.42r clientmain.42m Example2Client.42m WSHelper.42m
```

Related concepts
WSHelper library on page 3499
The WSHelper library.

Change WS client behavior at runtime
Various aspects of access to a Web Service may be implemented on the client side at runtime using global endpoint records.

Genero Web Services generates a global record called tGlobalEndpointType to change the client behavior at runtime without the need for modifying any generated client stub.
If WS-Addressing 1.0 is enabled, the global generated record is called tWSAGlobalEndpointType.

If needed, you can also access the HTTP layer via the Request and Response record of the binding record. See Access HTTP request and response headers for a service on page 3327.

**Global Endpoint type definition**
The client stub provides a global endpoint type, tGlobalEndpointType.

The following global type is used by any generated client stub to allow the programmer to change the client behavior at runtime.

```
TYPE tGlobalEndpointType RECORD # End point
  Address RECORD
    Uri STRING
  END RECORD,
  Binding RECORD
    Version STRING, # HTTP Version (1.0 or 1.1)
    Cookie STRING,
    ConnectionTimeout INTEGER,
    ReadWriteTimeout INTEGER
  END RECORD
END RECORD
```

Description of variables:

- **Address.Uri**: Represents the location of the server.

  **Important**: It replaces the global variable of type STRING generated prior to version 2.40, therefore it is mandatory to regenerate the client stub and to modify the location assignation in your application.

- **Binding.Version**: Represents the HTTP version to use for communication (only 1.0 or 1.1 allowed, default is 1.1).

- **Binding.Cookie**: Represents the HTTP cookie to use for communication (or NULL if there is no cookie to send).

- **Binding.ConnectionTimeout**: Represents the maximum time in seconds to wait for the establishment of the connection to the server.

- **Binding.ReadWriteTimeout**: Represents the maximum time in seconds to wait for a connection read or write operation before breaking the connection.

**Related concepts**

- Global Endpoint user-defined type definition on page 3329
- Bindings defined for the Web service can be changed at runtime.

**WS-Addressing 1.0 Global Endpoint type definition**
The client stub provides a global endpoint type for WS-Addressing, tWSAGlobalEndpointType.

The following global type is used by any generated client stub where support for WS-Addressing 1.0 is enabled. It allows the programmer to change the client behavior at runtime, and to send additional WS-Addressing 1.0 reference parameters to a server.

If this global type is used in your main application, you must add the IMPORT xml instruction.

```
TYPE tWSAGlobalEndpointType RECORD # End point
  Address RECORD # Address
    Uri STRING, # URI
  Parameters DYNAMIC ARRAY OF xml.DomDocument
    ATTRIBUTES(XMLNamespace="##any",XMLAny) # End point WSA reference parameters
  END RECORD,
  Binding RECORD # Binding
    Version STRING, # HTTP Version (1.0 or 1.1)
    Cookie STRING, # Cookie to be set
    ConnectionTimeout INTEGER,# Connection timeout
```

Description of variables:

- **Address.Parameters**: Represents the WS-Addressing 1.0 reference parameter to send to a WS-Addressing 1.0 compliant server.

  For a description of other parameters, see Global Endpoint type definition on page 3325.

**Related concepts**

- Global Endpoint user-defined type definition on page 3329
- Bindings defined for the Web service can be changed at runtime.

**Change server location**

If the Web service server location changes, you can update the address in the global endpoint record.

To change the server location at runtime, set the record `Uri` member with a valid URL of another service. All services must respect the same WSDL contract. If you leave the variable unset, the client will connect to the server URL defined in the WSDL at code generation time.

Example:

```plaintext
```

You can assign this variable with a URL set in the FGLPROFILE (see Logical Service location).

If you are migrating from a version prior to 2.40, see also Web Services changes on page 172.

**Change the HTTP protocol version**

The version parameter allows you to change the HTTP version binding if required.

To communicate with a service that communicates only on a given version of HTTP, set the record `Version` member with the desired value. If you leave the variable unset, the client will communicate in HTTP 1.1.

Example:

```plaintext
LET Calculator_CalculatorPortTypeEndpoint.Binding.Version = "1.0"
```

If you do not want the request to be split into chunks, set the HTTP protocol version to 1.0.

**Important**: On GMI mobile devices, the HTTP protocol version definition is ignored, it will always be version 1.1.

**Set an HTTP cookie**

Manage the cookie binding at runtime using the global endpoint record parameter.

To send an HTTP cookie to the service, set the record `Cookie` member with the cookie value. If you leave the variable unset, the client won't send any cookie.

Example:

```plaintext
LET Calculator_CalculatorPortTypeEndpoint.Binding.Cookie = "MyCookie=AValue"
```

Unset that variable if you don't need the cookie to be sent anymore.

**Set the connection timeout for a service**

A default connection timeout value is set, but you can configure the value at runtime, if required.

To change the default timeout value to establish a connection to the service, set the record member `ConnectionTimeout` with the timeout value in seconds. The `ConnectionTimeout` is a maximum time to establish the connection.
If the server is not available, you can get the response sooner than ConnectionTimeout. The ConnectionTimeout also avoids indefinitely waiting for a response when a server is reachable but not responding to requests.

Example:

```
LET Calculator_CalculatorPortTypeEndpoint.Binding.ConnectionTimeout = 15
```

**Important:** On GMI mobile devices, the max of ConnectionTimeout and ReadWriteTimeout will be used.

**Related concepts**

Set the read and write timeout for a service on page 3327

Set the ReadWriteTimeout value to indicate how long you are willing to wait for the server to respond to requests.

**Set the read and write timeout for a service**

Set the ReadWriteTimeout value to indicate how long you are willing to wait for the server to respond to requests.

To change the default time the application waits for a response from a service, set the ReadWriteTimeout parameter with a time out value in seconds.

Example:

```
LET Calculator_CalculatorPortTypeEndpoint.Binding.ReadWriteTimeout = 5
```

**Important:** On GMI mobile devices, the max of ConnectionTimeout and ReadWriteTimeout will be used.

**Related concepts**

Setting a time period for the response on page 3324

To protect against remote server failure or unavailability, set a timeout value that indicates how long you are willing to wait for the server to respond to your request.

Set the connection timeout for a service on page 3326

A default connection timeout value is set, but you can configure the value at runtime, if required.

**Access HTTP request and response headers for a service**

Configure additional headers for requests and responses by adding them to the globals endpoint record.

To access HTTP headers exchanged between the Genero client and a web service, you must use the following records in the binding section:

- one record called Request in order to customize HTTP headers to be sent to a web service
- one record called Response in order to retrieve all HTTP headers returned by a web service

```
TYPE tGlobalEndpointWithHttpLayerType RECORD # End point
  Address RECORD # Address
    Uri STRING # URI
  END RECORD,
  Binding RECORD # Binding
    Version STRING, # HTTP Version (1.0 or 1.1)
    Cookie STRING, # Cookie to be set
    Request RECORD
      Headers DYNAMIC ARRAY OF RECORD # HTTP Headers
        Name STRING,
        Value STRING
    END RECORD,
  Response RECORD
    Headers DYNAMIC ARRAY OF RECORD # HTTP Headers
        Name STRING,
        Value STRING
    END RECORD,
  ConnectionTimeout INTEGER,# Connection timeout
  ReadWriteTimeout INTEGER # Read write timeout
  CompressRequest STRING # HTTP compression mode (gzip or deflate)
```

Description of additional Request and Response variables:

- **Binding.Request.Headers**: Represents the additional HTTP headers to be sent to the web service. (Notice that client stub headers will replace user ones with the same name).
- **Binding.Response.Headers**: Represents the HTTP headers returned by a web service.

**Related reference**

- **Global Endpoint type definition** on page 3325
  The client stub provides a global endpoint type, tGlobalEndpointType.
- **WS-Addressing 1.0 Global Endpoint type definition** on page 3325
  The client stub provides a global endpoint type for WS-Addressing, tWSAGlobalEndpointType.

**WS client stubs and handlers**

To access a remote Web Service, you first must get the WSDL information from the service provider.

To access a remote Web service, you first must get the WSDL information from the service provider. Sample services can be found through UDDI registries (http://www.uddi.org), or on other sites such as XMethods.

**Generating stub files for a GWS Client**

Use the `fglwsdl` tool to generate the BDL stub from a WSDL URL or file.

This example requests the Calculator Web service information from the specified URL, and the output files will have the base name `ws_calculator`:

```
```

For a client application, `fglwsdl` generates two output files. It is not advised to modify these files:

- `filename.inc` - the globals file, containing declarations of global variables that can be used as input or output for functions accessing Web service operations, and the global `wsError` record. In our example, the file is `ws_calculator.inc`.
  
  This file must be listed in a GLOBALS statement at the top of any `.4gl` modules that you write for your GWS Client application.
- `filename.4gl` - containing the definitions of the functions that can be used in your GWS client application to perform the requested Web service operation, and the code that manages the Web service request. In our example, the file is `ws_calculator.4gl`.
  
  This file must be compiled and linked into your GWS Client application.

**Related concepts**

- **SOAP 1.1 and 1.2** on page 3280
  GWS is able to communicate with Web services provided with SOAP 1.1 or SOAP 1.2.

**Handling GWS server errors**

When a Genero Web Services service operation returns a status that is non-zero, you can get a more detailed error description from the global record `wsError`.

This record is defined in the globals `.inc` file.

```
DEFINE wsError RECORD
  code STRING,         -- Short description of the error
  codeNS STRING,       -- The namespace of the error code
  description STRING,  -- Long description of the error
  action STRING        -- internal "SOAP action"
END RECORD
```
Global Endpoint user-defined type definition

Bindings defined for the Web service can be changed at runtime.

The `fglwsdl` tool generates the globals (inc) file to use for a Web services client. Part of this globals file is a global endpoint user-defined type definition.

```plaintext
# Global Endpoint user-defined type definition

TYPE tGlobalEndpointType RECORD   # End point
    Address RECORD # Address
        Uri STRING                 # URI
    END RECORD,
    Binding RECORD # Binding
        Version STRING,             # HTTP Version (1.0 or 1.1)
        Cookie STRING,              # Cookie to be set
        ConnectionTimeout INTEGER,  # Connection timeout
        ReadWriteTimeout INTEGER,   # Read write timeout
        CompressRequest STRING      # HTTP request compression mode (gzip or deflate)
    END RECORD

# Location of the SOAP endpoint.
# You can reassign this value at run-time.
#
DEFINE EchoDocStyle_EchoDocStylePortTypeEndpoint tGlobalEndpointType
```

The `CompressRequest` entry is of type string. It is NULL by default, meaning that no request is compressed. To compress a request, set this variable to `gzip` or `deflate`. The server must support compression, otherwise the request will be rejected.

Related reference

**Global Endpoint type definition** on page 3325
The client stub provides a global endpoint type, `tGlobalEndpointType`.

**WS-Addressing 1.0 Global Endpoint type definition** on page 3325
The client stub provides a global endpoint type for WS-Addressing, `tWSAGlobalEndpointType`.

The generated functions

Defines the requirements for Genero Web Services (GWS) client functions.

Genero Web Services (GWS) client functions have the following requirements:

- The function cannot have input parameters.
- The function cannot have return values.
- The function's input message must be defined as a global or module RECORD.
- The function's output message must be defined as a global or module RECORD.

As a result, two types of GWS functions are generated for the Web Service operation that you requested:

- One function type uses global records for the input and output. The names of these functions end in "._g". Before calling the function in your GWS Client application, you must set the values in the global input record. After the function call, the status of the request is returned from the server, and the output message is stored in the global output record. In addition to performing the desired operation, this function handles the communication for the SOAP request and response, and sets the values in the `wsError` record as needed.
- The other function type serves as a "wrapper" for the "._g" function. It passes the values of input parameters to the "._g" function, and returns the output values and status received from the "._g" function. Your client application does not need to directly access the global records. This function can only be used if the parameters are simple variables (no records or arrays).
The generated .inc globals file contains comments that list the prototypes of the functions for the GWS operation, and the definitions of the global INPUT and OUTPUT records.

**The generated callback handlers**
Understand the function of callback handlers, and how to generate them in your client stub.

More and more Web services provide support for the different WS-* specifications. To enable a better interoperability with such services, the fglwssl on page 2081 tool allows the programmer to modify the SOAP request before it is sent, and to perform additional verifications of the SOAP response before it is returned from the BDL function.

When option -domHandler is used, the fglwssl tool performs the following two operations at once:

- It generates the client stub based entirely on the DOM API to ease the manipulation of the XML requests and responses.
- It generates additional calls for each operation of a service to execute one of the three callback handlers the programmer has to implement.

**Handler definition**
Describes the various callback handlers and how they are defined.

There are three kind of callbacks you must implement for each service generated with the -domHandler option.

- The request handler that allows the modification of the entire SOAP request before it is sent over the net.
  - It must be named `ServiceName_HandleRequest`, where `ServiceName` is the name of the service following the different prefix options used during generation.
  - It must return `TRUE` if you want the caller function to continue normally or `FALSE` to return from the caller function with a SOAP error you can define via the `wsError` record.

```plaintext
FUNCTION ServiceName_HandleRequest(operation, doc, header, body)
DEFINE operation STRING          -- Operation name of the
-- request to be modified.
DEFINE doc xml.DomDocument        -- Entire XML document of the request
DEFINE header xml.DomNode          -- XML node of the SOAP header
-- of the request
DEFINE body xml.DomNode           -- XML node of the SOAP body of the
-- request
CASE operation
  WHEN "Add"
    ... -- Use the DOM APIs to modify the request of the Add operation
  WHEN "Sub"
    ... -- Use the DOM APIs to modify the request of the Sub operation
  OTHERWISE
    DISPLAY "No modification for operation ":operation
END CASE
RETURN TRUE -- Continue normally in Add_g() or Sub_g()
END FUNCTION
```

- The response handler that allows the validation of the entire SOAP response before it is deserialized into the corresponding record.
  - It must be named `ServiceName_HandleResponse`, where `ServiceName` is the name of the service following the different prefix options used during generation.
  - It must return `TRUE` if you want the caller function to continue normally or `FALSE` to return from the caller function with a SOAP error you can define via the `wsError` record.

```plaintext
FUNCTION ServiceName_HandleResponse(operation, doc, header, body)
DEFINE operation STRING         -- Operation name of the
-- response to be checked.
DEFINE doc xml.DomDocument        -- Entire XML document of the response
DEFINE header xml.DomNode          -- XML node of the SOAP header of
-- the response
DEFINE body xml.DomNode           -- XML node of the SOAP body of the
```
The fault response handler that allows the verification of the entire SOAP fault response before it is de-serialized into the wsError record.

It must be named ServiceName_HandleResponseFault, where ServiceName is the name of the service according to the different prefix options used during generation. It must return TRUE if you want the caller function to continue normally or FALSE to return from the caller function with a SOAP error you can define via the wsError record.

```
FUNCTION ServiceName_HandleResponseFault(operation,doc,header,body)
  DEFINE operation STRING          -- Operation name of the fault response to be checked.
  DEFINE doc xml.DomDocument       -- Entire XML document of the fault response
  DEFINE header xml.DomNode        -- XML node of the SOAP header of the fault response
  DEFINE body xml.DomNode          -- XML node of the SOAP body of the fault response
  CASE operation
    WHEN "Add"
      ... -- Use the DOM APIs to verify the SOAP fault response of the Add operation
    WHEN "Sub"
      ... -- Use the DOM APIs to verify the SOAP fault response of the Sub operation
    OTHERWISE
      DISPLAY "No verification for operation :",operation
  END CASE
  RETURN TRUE -- Continue normally in Add_g() or Sub_g()
END FUNCTION
```

Example output
The example WSDL file for the Calculator Web Service provides information about the service.

The generated file ws_calculator.inc lists the prototype for the following functions: Add and Add_g functions, the asynchronous AddRequest_g and AddResponse_g functions, as well as the definitions of the global variables Add and AddResponse:

```
# Operation: Add## FUNCTION: Add_g() -- Function that uses the global input and output records## RETURNING: soapStatus -- An integer where 0 represents success## INPUT: GLOBAL Add## OUTPUT: GLOBAL AddResponse##
# FUNCTION: Add(p_a, p_b) -- Function with input parameters that correspond to the a and b variables## RETURNING: soapStatus ,p_r -- of the global INPUT record
```
--- Return values are the status
of the global OUTPUT record

# FUNCTION: AddRequest_g() -- Asynchronous function that uses the global input record
# RETURNING: soapStatus -- An integer where 0 represents success, -1 error and -2 means a previous request was sent and that a response is in progress.

# FUNCTION: AddResponse_g() -- Asynchronous function that uses the global output record
# RETURNING: soapStatus -- An integer where 0 represents success, -1 error and -2 means that the response was not yet received, and that a new call should be done later.

#VARIABLE : Add -- defines the global INPUT record
DEFINE Add RECORD ATTRIBUTES(XMLName="Add",
   XMLNamespace="http://tempuri.org/"
   a INTEGER ATTRIBUTES(XMLName="a",XMLNamespace=""),
   b INTEGER ATTRIBUTES(XMLName="b",XMLNamespace="")
END RECORD

# VARIABLE : AddResponse -- defines the global OUTPUT record
DEFINE AddResponse RECORD ATTRIBUTES(XMLName="AddResponse",
   XMLNamespace="http://tempuri.org/"
   r INTEGER ATTRIBUTES(XMLName="r",XMLNamespace="")
END RECORD

Multipart in the client stub
You can generate a client stub for a Web service that has multiple parts.

If the WSDL for a Web service indicates that the Web service uses multiple parts, the client stub generated will support multiple parts.

For the request
There are as many com.HTTPPart input parameters as parts defined for the input request, plus one AnyInputParts DYNAMIC ARRAY OF com.HTTPPart parameter, to manage the optional parts a user can add to the request.

For example:

FUNCTION xxx_g(InputHttpPart_1, ..., InputHttpPart_n, AnyInputParts)
   DEFINE InputHttpPart_1 com.HTTPPart
   ...
   DEFINE InputHttpPart_n com.HTTPPart
   DEFINE AnyInputParts DYNAMIC ARRAY OF com.HTTPPart
   ...
   RETURN wsstatus
END FUNCTION
For the response

There are as many `com.HTTPPart` variables as are described in the WSDL, plus one `AnyOutputParts` `DYNAMIC ARRAY OF com.HTTPPart` to handle the optional parts that may be returned by a service.

For example:

```gregated_code
FUNCTION xxx_g()
  DEFINE wsstatus INTEGER
  DEFINE OutputHttpPart_1 com.HTTPPart
  DEFINE AnyOutputParts DYNAMIC ARRAY OF com.HTTPPart
  ...
  RETURN wsstatus, OutputHttpPart_1, AnyOutputParts
END FUNCTION
```

Related concepts

**SOAP multipart style requests in GWS** on page 3297
This topic describes multipart support with Genero Web Services

**Using the generated functions**
Learn how to use the functions and global records in the stub file to write client applications that use the Web service.

The information about the Add function from the `ws_calculator.inc` file, for example, allows you to write code in your own `.4gl` module that uses this operation as part of your Client application.

**Using parameters and return values**
You call the Add function in your Client application, defining variables for the parameters and return values. The input variables for our example are simple integers.

```gregated_code
FUNCTION myWScall()
  DEFINE op1 INTEGER
  DEFINE op2 INTEGER
  DEFINE result INTEGER
  DEFINE wsstatus INTEGER
  ...
  LET op1 = 6
  LET op2 = 8
  CALL Add(op1, op2)
  RETURNING wsstatus, result ...
  DISPLAY result
```

**Using global records**
An alternative option is to call the `Add_g` function instead, using the global records `Add` and `AddResponse` directly. If the input variables are complex structures like records or arrays, you are required to use this function.

```gregated_code
FUNCTION myWScall()
  DEFINE wsstatus INTEGER
  ...
  LET Add.a = 6
  LET Add.b = 8
  LET wsstatus = Add_g()
  ...
  DISPLAY AddResponse.r
```

In this case, the status is returned by the function, which has also put the result in the `AddResponse` global record.

See **Tutorial: Writing a Client Application** for more information. The `demo/WebServices` subdirectory of your Genero installation directory contains complete examples of Client Applications.
Using asynchronous calls

If you don't want your application to be blocked when waiting for the response to a request, it is recommended that you first call `AddRequest_g();` this will send the request using the global `Add` record to the server. It returns a status of 0 (zero) if everything goes well, -1 in case of error, or -2 if you tried to resend a new request before the previous response was retrieved.

```bdl
FUNCTION sendMyWScall()
    DEFINE wsstatus INTEGER
    ...
    LET Add.a = 6
    LET Add.b = 8
    LET wsstatus = AddRequest_g()
    IF wsstatus <> 0 THEN
        DISPLAY "ERROR ", wsError.code
    END IF
    ...
```

You can then call `AddResponse_g` to retrieve the response in the `AddResponse` global record of the previous request. If the returned status is 0 (zero), the response was successfully received; -1 means that there was an error, and -2 means that the response was not yet received and that the function needs to be called later.

```bdl
FUNCTION retrieveMyWScall()
    DEFINE wsstatus INTEGER
    ...
    LET wsstatus = AddResponse_g()
    CASE wsstatus
        WHEN -2
            DISPLAY "No response available, try later"
        WHEN 0
            DISPLAY "Response is ", AddResponse.r
        OTHERWISE
            DISPLAY "ERROR ", wsError.code
    END CASE
    ...
```

You can mix the asynchronous call with the synchronous one as they are using two different requests. In other words, you can perform an asynchronous request with `AddRequest_g();` then a synchronous call with `Add_g();` and then retrieve the response of the previous asynchronous request with `AddResponse_g();`.

**Important:** In development mode, a single BDL Web Service server can only handle one request at a time, and several asynchronous requests in a row without retrieving the corresponding response will lead to a deadlock. To support several asynchronous requests in a row, it is recommended that you are in deployment mode with a GAS as the front-end.

**Using logical names for service locations**

Using a logical reference for the Web service, instead of the real URL, in your client application URL binding has advantages for working with and deploying applications.

Genero Web Services, starting with version 2.00, provides a repository for Web Service locations using FGLPROFILE. To achieve maximum flexibility, you can map a logical reference used by your Web Services client application to an actual URL. This is subject to the network configuration and access rights management of the deployment site.

**Important:** On GMI mobile devices, FGLPROFILE settings for logical names are not supported.

**FGLPROFILE entry**

FGLPROFILE entries can be used to define server URLs by using URL patterns.

The following entry in the FGLPROFILE file maps the logical reference "myservice" to an actual URL:

```plaintext
```
**Related concepts**

*FGLPROFILE: Server URL patterns* on page 3498

FGLPROFILE entries can be used to define multiple server URLs, by using URL patterns.

**Logical reference in the client application**

Specify a logical name for the Web service in the endpoint record of your client application.

When you generate a Client stub from WSDL information using the tool fglwsdl, a global variable for the URL of the Web Service is contained in the .inc file.

For example:

```plaintext
# Location of the SOAP server.
# You can reassign this value at run-time.
#
# DEFINE Calculator_CalculatorPortTypeEndpoint tGlobalEndpointType
```

You can assign a logical name, instead of the Web service URL, to this global variable in your Web Services client application:

```plaintext
LET Calculator_CalculatorPortTypeEndpoint.Address.Uri = "alias://myservice"
```

When the client application accesses the service, the actual location will be supplied by the entry in your FGLPROFILE file on the client machine. This allows you to provide the same compiled .42r application to different customers. The entries in the FGLPROFILE file on each customer's machine would customize the Web Service location for that customer.

If you are migrating from a version prior to 2.40, see also *Web Services changes* on page 172.

**Related concepts**

*Global Endpoint user-defined type definition* on page 3329

Bindings defined for the Web service can be changed at runtime.

**Logical reference in the URL**

Specify a Web service URL in the FGLPROFILE file.

When you deploy a Genero Web Service with a GAS behind a Web Server, the service can be accessed by two different URLs. You can use a logical name in the URL binding record, mapping the actual location of the Web Service in your FGLPROFILE file, depending on the location of the client machine.

For example:

- For internal clients: http://zeus:6394/ws/r/myservice
- For clients using the Web: http://www.myServer.com/...

These two URLs can be mapped in the FGLPROFILE file on the client machine, each specifying the location of the Service.

**Related concepts**

*Logical reference in the client application* on page 3335

Specify a logical name for the Web service in the endpoint record of your client application.

*FGLPROFILE entry* on page 3334

FGLPROFILE entries can be used to define server URLs by using URL patterns.

**Configure a WS client to access an HTTPS server**

Configuration steps to access a server in HTTPS.

To configure access to an HTTPS server, you need a client certificate.

**Important:** On GMI mobile devices the iOS SSL/TLS layer is used for HTTPS, and the device Keychain® typically holds the server certificate authority list. Therefore, the FGLPROFILE file security settings are ignored
with the exception of the following: `security.global.ca`, `security.global.ca.lookuppath`, and `security.global.systemca`.

Before you begin, there are options to consider depending on how you wish to use the client certificate:

- If you do not have the certificate information in your FGLPROFILE file, Genero Web Services creates a certificate for you. This is an implicit or temporary certificate that is valid for a session only. For more information see HTTPS configuration on page 3305.
  
  **Note:** For the implicit certificate, no configuration is required.

- Alternatively, for stronger security, you generate a client certificate of your own, configure your application to use the client certificate generated, and add the configuration details to the FGLPROFILE file. Follow the steps outlined in this section.
  
  **Note:** In a production environment, some servers provide a client certificate and you use the certificate as provided, and add the configuration details to the FGLPROFILE file.

The openssl command line tool can be used to create your own certificates for the configuration of secured communication. The following steps outline the configuration process:

**Related concepts**

- Configuring the Apache Web server for HTTPS on page 3460
  
  Configuration steps to secure a Web service for Apache server in HTTPS.

- Certificates in practice on page 3313
  
  Procedures and tools for creating, importing, and viewing certificates and keys.

- Encryption and authentication on page 3299
  
  A scenario involving a person (Georges) and his bank guides you through the concepts of secured communication, certificates, and certificate authorities.

- Accessing secured services on page 3303
  
  Security and authentication are important. Genero Web Services provides various communications options for a client to connect to a Web service.

**Create the client certificate**

Generate a root certificate authority that signs a client certificate.

In this task you create your own root certificate authority and client certificate using the OpenSSL command line tool.

1. Create the root certificate authority.
   
   a) Create the root certificate authority serial file.
      
      ```
      $ echo 01 > MyCompanyCA.srl
      ```
      
      This command creates a serial file with an initial HEX value 01. OpenSSL uses this file to track the serial numbers of certificates it creates. The serial file is typically given the same name as the CA with the extension .srl.
      
      b) Create the Certificate Signing Request (CSR) and private key.
         
         ```
         $ openssl req -new -out MyCompanyCA.csr -keyout MyCompanyCA.pem
         ```
         
         Follow the instructions to create the CSR.
      
      c) Create the Root Certificate Authority for a period of validity of 2 years.
         
         ```
         $ openssl x509 -trustout -in MyCompanyCA.csr -out MyCompanyCA.crt -req -signkey MyCompanyCA.pem -days 730
         ```
         
         **Note:** The private key file (MyCompanyCA.pem) of a Root Certificate Authority must be handled with care. This file is responsible for the validity of all other certificates it has signed. As a result, it must not be accessible by other users.

2. Create the client's X.509 certificate and private key.
a) Create the client's Certificate Signing Request and private key.

```
$ openssl req -new -out MyClient.csr
```

**Note:** By default, openssl outputs the private key in the `privkey.pem` file. If you want to specify a different file name, or if your openssl version does not output the private key by default, add the `-keyout <myprivkey>.pem` to the command.

b) Remove the password from the RSA private key.

```
$ openssl rsa -in privkey.pem -out MyClient.pem
```

**Note:** The unprotected private key is output in `MyClient.pem`.

c) Create the client's certificate (self-signed X.509 certificate valid for a period of 1 year) trusted by the Root Certificate Authority created in step 1.

```
$ openssl x509 -in MyClient.csr -out MyClient.crt -req
-sigkey MyClient.pem -CA MyCompanyCA.crt
-CAkey MyCompanyCA.pem -days 365
```

**Note:** Most servers do not check the identity of the clients. For these servers, the client's certificate does not necessarily need to be trusted; it is only used for data encryption purpose. If, however, the server performs client identification, you must trust a Certificate Authority in which it has total confidence concerning the validity of the client's certificates.

**Note:** The purpose of the client's certificate is to identify the client to any server; therefore the subject of the certificate must correspond to the client's identity as it is known by the servers.

**Note:** To import the certificate in a keystore you can create a pkcs12 certificate. See Import a certificate and its private key into the Windows key store on page 3315.

**What to do next**

Configure your FGLPROFILE file for the client certificate. See Configure for the client certificate on page 3307.

**Related concepts**

Encryption and authentication on page 3299
A scenario involving a person (Georges) and his bank guides you through the concepts of secured communication, certificates, and certificate authorities.

The OpenSSL (openssl) tool on page 3313
The openssl command line tool creates certificates for the configuration of secured communications.

**Related tasks**

View a certificate on page 3316
This procedure allows you to view a certificate using the openssl command.

**Configure for the client certificate**

Configure your application to use the client certificate generated. For production systems, you add the configuration details to your FGLPROFILE file.

You have generated a client certificate as described in Create the client certificate on page 3306.

In this task you add the certificate information to your FGLPROFILE file.

Add the client's security configuration to your FGLPROFILE file.

The client security entry defines the certificate and the associated private key used by the Genero Web Services client during an HTTPS communication. The security entry must be defined with an unique identifier (`id1` in this example).

```python
security.id1.certificate = "MyClient.crt"
security.id1.privatekey = "MyClient.pem"
```
Note: If the private key is protected with a password, you must remove it or create a script that returns the password on demand. See Create the client certificate.

Your applications are configured to use the client certificate.

What to do next

Create the client's certificate authority list and configure the global certificate authority list (security.global.ca) in your FGLPROFILE file. See Configure for the certificate authority list on page 3308.

Related concepts

FGLPROFILE entries for Web Services on page 3489
The FGLPROFILE entries relating to Genero Web Services are divided between five categories: security, basic or digest HTTP authentication, proxy configuration, server configuration, and XML cryptography.

FGLPROFILE: HTTP(S) Proxy Authentication on page 3496
FGLPROFILE entries can be used to define a connection to an HTTPS server via a proxy, and with HTTP and Proxy Authentication.

Encryption and authentication on page 3299
A scenario involving a person (Georges) and his bank guides you through the concepts of secured communication, certificates, and certificate authorities.

Related tasks

View a certificate on page 3316
This procedure allows you to view a certificate using the openssl command.

Configure for the certificate authority list

The client needs to check to see if the server certificate is trusted. This is done using a certificate authority list.

In this task you create the client's certificate authority list using the OpenSSL command line tool, and set the global certificate authority list in your FGLPROFILE file.

Important: On GMI mobile devices the iOS SSL/TLS layer is used for HTTPS, and the device Keychain typically holds the server certificate authority list. Therefore, the FGLPROFILE file security settings are ignored with the exception of the following: security.global.ca, security.global.ca.lookuppath, and security.global.systemca.

1. Create the client's certificate authority list.

   a) Save the certificates of the HTTPS server to disk.

      Type the server's URL in your browser. When prompted, save all the certificates from the Certificate Hierarchy. For more information see, Selecting the certificate to add on page 3317 and Missing certificates on page 3311.

   b) Create the client's Certificate Authority List by running the following command for each of the certificates that you saved to disk.

      $ openssl x509 -in ServerCertificate.crt -text >> ClientCAList.pem

      Note: All trusted certificate authorities are listed. These are checked following a chain of child to parent certificates until a certificate is reached which is trusted. All other certificates that were trusted by the Root Certificate Authority will also be considered as trusted by the client. For more information see Certificate authorities on page 3301.

2. Set the global certificate authority list in your FGLPROFILE file.

   The global certificate authority list entry defines the file containing the certificate authority list used by the Genero Web Services client to validate all certificates coming from the different servers it will connect to.

   security.global.ca = "ClientCAList.pem"
Note: The file is located based on the current execution directory. If you use Genero Studio, for instance, fglrun may not be executed in the same directory as when you use the command line. The recommended practice therefore is to specify an absolute path for the .pem, for example:

security.global.ca = "/opt/usr/certs/ClientCAList.pem"

If security.global.ca is not defined, Genero Web Services will look to see whether the operating system has a keystore, otherwise security.global.ca.lookuppath will be used. For further information, see Certificate authorities in HTTPS configuration on page 3305.

The client application is configured to use the appropriate certificate authority list to validate a server's certificate.

What to do next
In your FGLPROFILE file ensure there are configuration entries (ws.*) for the HTTPS server URL and for the HTTP authentication when accessing the HTTPS server. See Add configuration entries in your FGLPROFILE file on page 3309.

Related concepts
Error: Peer certificate is issued by a company not in our CA list on page 3310
When a client connects to a server using HTTPS, the client needs to trust the server it is in communication with. So the client needs to add the server's CAs (certificate authorities lists) to its trusted CAs.

Enable OCSP on page 3310
To enable OCSP (Online Certificate Status Protocol), set the security.global.ocsp.enable and security.global.ocsp.url entries in FGLPROFILE.

FGLPROFILE entries for Web Services on page 3489
The FGLPROFILE entries relating to Genero Web Services are divided between five categories: security, basic or digest HTTP authentication, proxy configuration, server configuration, and XML cryptography.

FGLPROFILE: HTTP(S) Proxy Authentication on page 3496
FGLPROFILE entries can be used to define a connection to an HTTPS server via a proxy, and with HTTP and Proxy Authentication.

Accessing secured services on page 3303
Security and authentication are important. Genero Web Services provides various communications options for a client to connect to a Web service.

Add configuration entries in your FGLPROFILE file
Add a set of configuration entries that specify the security configuration for GWS clients accessing an HTTPS server.

In this task you add configuration entries (ws.*) in your FGLPROFILE file for the HTTPS server URL and for HTTP authentication when accessing the HTTPS server. For an example, see FGLPROFILE: HTTP(S) Proxy Authentication on page 3496.

Important: On GMI mobile devices the iOS SSL/TLS layer is used for HTTPS, and the device Keychain® typically holds the server certificate authority list. Therefore, the FGLPROFILE file security settings are ignored with the exception of the following: security.global.ca, security.global.ca.lookuppath, and security.global.systemca.

Add configuration entries for the server.
The following entries must be defined with an unique identifier (such as "myserver"):

   (Line breaks added for document readability)
b) ws.myserver.security = "id1"

Where the value (id1 in this example) must match the unique identifier defined by the client security entry created in Configure for the client certificate on page 3307.

Tip: The unique identifier "myserver" can be used in the BDL client code in place of the actual URL.
Related concepts

FGLPROFILE entries for Web Services on page 3489

The FGLPROFILE entries relating to Genero Web Services are divided between five categories: security, basic or digest HTTP authentication, proxy configuration, server configuration, and XML cryptography.

Accessing secured services on page 3303

Security and authentication are important. Genero Web Services provides various communications options for a client to connect to a Web service.

Troubleshoot certificate issues on page 3310

You may encounter known (and common) issues when completing the Genero Web Services tutorials or when adding Web services of your own. These issues and their solutions are presented in the following topics.

Configure a WS client to connect via an HTTP Proxy

Configuration steps to connect via a HTTP proxy.

Important: On GMI mobile devices, FGLPROFILE settings are ignored: The device configuration for proxy will always be used.

1. Add the location of the proxy to your FGLPROFILE file with the proxy.http.location entry.

   Add the entry proxy.http.location to your fglprofile. For the value, provide the IP address of the HTTP proxy and the port number where the HTTP proxy is listening, separated by a colon. For example, to have a client connect via a HTTP proxy located at the IP address "10.0.0.170" and listening on port number "8080", add this entry to your fglprofile:

   ```
   proxy.http.location = "10.0.0.170:8080"
   ```

   Note: To configure the client to connect via an HTTPS proxy, replace http with https.

2. Define the list of host names the client will not have to connect to via a proxy with the proxy.http.list entry.

   Add the entry proxy.http.list to your FGLPROFILE file. For the value, provide a semi-colon separated list of clients. For example, to exclude all hosts beginning with "www.mycompany.com" or "www.google." from connecting via a HTTP proxy, add this entry to your fglprofile:

   ```
   proxy.http.list = "www.mycompany.com;www.google."
   ```

Related reference

Proxy configuration on page 3492

Configure a WS client to use IPv6

Configuration steps to customize IPv6 for a WS client.

A Web Services client program can access a WS server using IPv6.

URLs that map to IPv6 addresses will be automatically handled by the Web Services library. It is also possible to specify an IPv6 address directly as URL in your BDL code by enclosing the address in [] square brackets, for example:

```ruby
LET myURL = "http://[fe80::20c:29ff:fe05:9ca3]:80/index.html"
```

By default, the WS library will automatically use IPv6 addresses if available, and fallback to IPv4 if not. To overcome the default behavior, you can explicitly specify the IP version.

The platform where WS client programs execute must support IPv6.

1. If needed, force the IP version with the ip.global.version entry in your fglprofile file, by specifying "4" for IPv4 or "6" for IPv6.

   For example, to force IPv4 (when IPv6 is available):

   ```
   ip.global.version = "4"
   ```
2. When using IPv6 for link-local addresses, if several network interfaces exist on the machine, you can explicitly specify which interface must be used with the `ip.global.v6.interface.name` or `ip.global.v6.interface.id` entry in fglprofile.

In order to specify the IPv6 network interface by name, use:

```
ip.global.v6.interface.name = "eth0"
```

**Important:** The `ip.global.v6.interface.name` entry is not supported on Microsoft® Windows® platforms.

In order to specify the IPv6 network interface by id, use:

```
ip.global.v6.interface.id = "2"
```

**Related reference**
IPv6 configuration on page 3493

**Authenticate the WS client to a server (HTTP basic authentication)**
Configuration steps to authenticate the client to a server.

See [Basic or digest HTTP authentication](#) on page 3491. For an example, see FGLPROFILE: HTTP(S) Proxy Authentication on page 3496.

**Important:** On GMI mobile devices, FGLPROFILE settings are ignored: Use the `com.HTTPRequest.setAuthentication` API instead.

1. Add HTTP authenticate entries to your FGLPROFILE file.
   To connect to a server with HTTP Authentication, define the client login and password with the same values as registered on the server side. These entries must be defined with an unique identifier (`httpauth` in this example) to define a HTTP Authentication with "mylogin" as login and "mypassword" as password:

   ```
   authenticate.httpauth.login = "mylogin"
   authenticate.httpauth.password = "mypassword"
   ```

   See [RFC2617](#) for more details.

2. Encrypt the password.
   Due to security leaks, it is recommended that you NOT have a password in clear text. The Genero Web Services package provides the tool `fglpass`, which encrypts a password with a certificate that is readable only with the associated private key. To encrypt the HTTP authentication password:
   a) Encrypt the clear text password with `fglpass` using the client certificate.

   ```
   $ fglpass -e -c MyClient.crt
   Enter password :mypassword
   ```
   ```
   Note: fglpass outputs the encrypted password on the console but can be redirected to a file.
   ```
   ```
   b) Modify the HTTP authentication password entry by specifying the security configuration to use to decrypt it ("id1" in our example)

   ```
   authenticate.httpauth.password.id1="HWTFu8Q8E2t3e5D4joy7js8mB95oOGTzLmcAor9j5DS+CloillhGWzvZ9eWpflmIWSOn9IwoiJheYxfnu20uaGGmmiUGiHxT6341ePXNSicu32Nt1Vp9t6RcS0wN/p9a6D4XtiD9iHW7iQvXhqC9uamd3gI9Q3GhHwXOMM1Y/c8Y="
   ```
   ```
   Note: Hard returns have been added to the code sample above, for the purpose of printing and viewing within this document. The value for `authenticate.httpauth.password.id1` is a single string with no spaces.
   ```
   ```
   Note: The size of the encrypted password depends on the size of the public key, and can change based on the certificate used to encrypt it.
   ```
   ```
   3. Configure the client to authenticate to a server.
As a client is able to connect to different servers that do not know the client with the same login and password, it is necessary to specify the login and password that corresponds to each server. To authenticate the client known as "myclient" with the password passphrase by the server myserver, add the following entry:

```plaintext
ws.myserver.authenticate = "httpauth"
```

**Related tasks**

- Configure a WS client to access an HTTPS server on page 3305
- Configuration steps to access a server in HTTPS.

**Authenticate the WS client to a proxy**

Configuration steps to authenticate the client to a proxy (proxy authentication).

For an example, see FGLPROFILE: HTTP(S) Proxy Authentication on page 3496.

**Important:** On GMI mobile devices, FGLPROFILE settings are ignored: The device configuration for proxy will always be used.

1. Add an HTTP authenticate entry to your FGLPROFILE file.
   - To connect via a proxy with HTTP Proxy Authentication, it is necessary to define the client login and password as registered on the HTTP proxy.
   - The following two entries must be defined with an unique identifier (`proxyauth` for our example) to define a HTTP Proxy Authentication with `myapplication` as login and `mypassword` as password:

   ```plaintext
   authenticate.proxyauth.login = "myapplication"
   authenticate.proxyauth.password = "mypassword"
   ```

   See [RFC2617] for more details.

2. For proxy authentication, an entry must be made to the HTTP proxy configuration in order to authenticate a client.
   - To authenticate a client known as `myapplication` with `mypassword` as password by the HTTP Proxy, add the following entry to the HTTP proxy configuration:

   ```plaintext
   proxy.http.authenticate = "proxyauth"
   ```

   **Note:** To authenticate the client to a HTTPS proxy, replace `http` with `https`.

**Related reference**

- Proxy configuration on page 3492

**Writing a Web Services server application**

Create a Genero Web Services server using the SOAP protocol.

**Writing a Web services server function**

You create a standard Genero function and publish it as a Web function (Web services operation) using methods from the classes in the COM library.

There are some restrictions on the function - input and output parameters are not allowed. By using global or module variables, however, you can work around this exception.

See also Tutorial: Writing a GWS Server Application

The steps for writing a Web Services function:

**Define the input parameters**

Define a record for the input message of the Web function.

Even though input parameters are not allowed in Genero Web Service operations, each Web function can have one global variable or module variable that defines the input message of the function. This variable must be a record in which each field represents one of the input parameters of the Web function.
The name of each field corresponds to the name used in the SOAP request. These fields are filled with the contents of the SOAP request by the Web Services engine just before executing the corresponding BDL function.

**Example**

```plaintext
DEFINE add_in RECORD
    a INTEGER,
    b INTEGER
END RECORD
```

**Note:** Genero version 2.0 allows you to add optional attributes to the definition of data types. You can use attributes to map the BDL data types in a Genero application to their corresponding XML data types. See Attributes to Customize XML Mapping for additional information.

### Define the output parameters

Define a record for the output message of the function.

Even though output parameters in Genero Web functions are not allowed, each Web function can have one global variable or module variable that defines the output message of the function. This message must be a record where each field represents one of the output parameters of the Web function.

The name of each field corresponds to the name used in the SOAP request. These fields are retrieved from the Web Services engine immediately after executing the BDL function, and sent back to the client.

**Example**

```plaintext
DEFINE add_out RECORD
    r INTEGER
END RECORD
```

**Note:** GWS 2.0 allows you to add optional attributes to the definition of data types. You can use attributes to map the BDL data types in a Genero application to their corresponding XML data types. See Attributes to Customize XML Mapping for additional information.

### Write the BDL function

Your function defines an operation of the service.

A Web service function is a normal BDL function that uses the input and output records that you have defined.

**Example**

```plaintext
FUNCTION add()
    LET add_out.r = add_in.a + add_in.b
END FUNCTION
```

### Create and publish the Web services operation

Provide your Web service and its operation to users who can access it on the net.

Methods are available in the Genero Web Services library (com) to:

- Define the Web Service, by creating a WebService object
- Define the Web Services operation for your function, by creating a WebOperation object
- Publish the operation - associate it with the Web Service object that you defined.

The com library must be imported into each module of a Web Services Server application.

The following abbreviated example is from the Web Services Server tutorial:

```plaintext
IMPORT com
...
FUNCTION createservice()
    DEFINE serv com.WebService    # A WebService
    DEFINE op com.WebOperation    # Operation of a WebService
```
--Create WebService object
    "http://tempuri.org/webservices")

--Create WebOperation object
LET op = com.WebOperation.CreateRPCStyle("add", "Add", add_in, add_out)

--Publish the operation, associating it with the WebService object
CALL serv.publishOperation(op, NULL)
...
END FUNCTION

See the Writing a Web server application on page 3348 and Choosing a web services style on page 3358 for complete examples and explanations.

**WS server stubs and handlers**

Describes using a server stub from a compatible Web service that you can use in your GWS server application.

Sample Web services can be found through UDDI registries (http://uddi.xml.org), or on other sites such as XMethods.

**Generating files for a GWS server**

Use the fglwsdl -s tool server option to generate the BDL server stub from a WSDL.

You can write a Genero Web Services server application for a Web Service that you have created; see Tutorial: Writing a Server Application.

If you want to make sure your Web Service is compatible with that of a third-party (an accounting application vendor, for example), you can use the fglwsdl tool to obtain the WSDL information that complies with that vendor's standards, and to generate corresponding files that can be used in your GWS server application.

This example requests the Calculator Web Service information from the specified URL, and the output files will have the base name "ws_calculator".

```bash
```

For a server application, fglwsdl generates two files. It is not advised to modify these files:

- *filename*.inc - the globals file, containing declarations of global variables that can be used as input or output to functions accessing the Web Service operations. In our example, the file is *ws_calculatorService.inc*.
  
  This file must be listed in a GLOBALS statement at the top of any .4gl modules that you write for your GWS Server application.

- *filename*.4gl - containing a function that creates the service described in the WSDL, publishes the operations of the service, and registers the service. In our example, the file is *ws_calculatorService.4gl*.

  This file must be compiled and linked into your GWS Server application.

**Server handlers**

Create and register your callback handlers (request, and response) to modify the WSDL.

The COM library allows you to intercept high-level Web services operation on the server side. You can now define three BDL functions via the following methods of the Web service class. They will be executed at different steps of a web service request processing in order to modify the SOAP request, response or the generated WSDL document before or after the SOAP engine has processed it. This helps handle WS-* specifications not supported in the web service API.

- Method registerWSDLHandler()
- Method registerInputRequestHandler()
- Method registerOutputRequestHandler()
All three kinds of BDL callback functions must conform to this prototype:

```java
FUNCTION CallbackHandler( doc xml.DomDocument )
    RETURNING xml.DomDocument
```

Example 1: Modify the generation of a WSDL

Register your handler with:

```java
CALL serv.registerWsdlHandler("WSDLHandler")
```

where `serv` is of class `com.WebService` and `WSDLHandler` is the following function:

```java
FUNCTION WSDLHandler(wsdl)
    DEFINE wsdl Xml.DomDocument
    DEFINE node Xml.DomNode
    DEFINE list Xml.DomNodeList
    DEFINE ind INTEGER
    DEFINE name STRING
    # Add a comment
    LET node = wsdl.createComment("First modified WSDL via a BDL callback function")
    CALL wsdl.prependDocumentNode(node)
    # Rename input and output parameter in UPPERCASE
    -- first input parameter for selectByXPath above
    -- one string, no spaces!
    FOR ind=1 TO list.getCount()
        LET node = list.getItem(ind)
        LET name = node.getAttribute("name")
        LET name = name.toUpperCase()
        CALL node.setAttribute("name",name)
    END FOR
    RETURN wsdl
END FUNCTION
```

If NULL is returned from the callback function, an HTTP error will be sent and the `ProcessServices()` returns error code -20.

Example 2: Change the SOAP incoming request

Register your handler with:

```java
CALL serv.registerInputRequestHandler("InputRequestHandler")
```

where `serv` is of class `com.WebService` and `InputRequestHandler` is this function:

```java
FUNCTION InputRequestHandler(in)
    DEFINE in Xml.DomDocument
    DEFINE ind INTEGER
    DEFINE node Xml.DomNode
    DEFINE copy Xml.DomNode
    DEFINE tmp Xml.DomNode
    DEFINE parent Xml.DomNode
    DEFINE name STRING
    DEFINE list Xml.DomNodeList
    # Change input parameter below myrecord in lower case
    # to follow high-level web service
```
 FOR ind = 1 TO list.getCount()
 LET node = list.getItem(ind)
 LET parent = node.getParentNode()
 LET name = node.getLocalName()
 LET copy = in.createElementNS(node.getPrefix(), name.toLowerCase(), node.getNamespaceURI())
 LET tmp = node.getFirstChild()
 LET tmp = tmp.clone(true)
 CALL copy.appendChild(tmp)
 CALL parent.replaceChild(copy,node)
 END FOR
 RETURN in
END FUNCTION

If NULL is returned from the callback function, a SOAP fault will be sent (but can be changed from the output handler) and the ProcessServices() returns error code -18.

**Example 3: Modify the SOAP outgoing request**

Register your handler with:

CALL serv.registerOutputRequestHandler("OutputRequestHandler")

where serv is of class com.WebService and OutputRequestHandler is this function:

FUNCTION OutputRequestHandler(out)
 DEFINE out Xml.DomDocument
 DEFINE ind INTEGER
 DEFINE node Xml.DomNode
 DEFINE copy Xml.DomNode
 DEFINE tmp Xml.DomNode
 DEFINE parent Xml.DomNode
 DEFINE name STRING
 DEFINE list Xml.DomNodeList
 # Change output parameter below myrecord in uppercase
 # before sending back to the client
 FOR ind = 1 TO list.getCount()
 LET node = list.getItem(ind)
 LET parent = node.getParentNode()
 LET name = node.getLocalName()
 LET copy = out.createElementNS(node.getPrefix(),name.toUpperCase(), node.getNamespaceURI())
 LET tmp = node.getFirstChild()
 LET tmp = tmp.clone(true)
 CALL copy.appendChild(tmp)
 CALL parent.replaceChild(copy,node)
 END FOR
 RETURN out
END FUNCTION

If NULL is return from the callback function, a SOAP fault will be sent and the ProcessServices() returns error code -19.
Example output
What you can expect to find in the server stub files.

In the generated file `ws_calculatorService.inc`, the definitions of the variables for the input and output record are the same as those generated for the Web Service client application:

```plaintext
#VARIABLE : Add -- defines the global INPUT record
DEFINE Add RECORD ATTRIBUTES(XMLName="Add", 
XMLNamespace="http://tempuri.org/")
   a INTEGER ATTRIBUTES(XMLName="a",XMLNamespace=""),
   b INTEGER ATTRIBUTES(XMLName="b",XMLNamespace=""
END RECORD

# VARIABLE : AddResponse -- defines the global OUTPUT record
DEFINE AddResponse RECORD ATTRIBUTES(XMLName="AddResponse", 
XMLNamespace="http://tempuri.org/")
   r INTEGER ATTRIBUTES(XMLName="r",XMLNamespace=""
END RECORD
```

The generated file `ws_calculatorService.4gl` contains a single function that creates the Calculator service, creates and publishes the service operations, and registers the Calculator service:

```plaintext
FUNCTION Createws_calculatorService()
   DEFINE service com.WebService
   DEFINE operation com.WebOperation
   ...  # Create Web Service
   LET service = com.WebService.CreateWebService("Calculator", 
                "http://tempuri.org/")
   # Publish Operation : Add
   LET operation = com.WebOperation.CreateRPCStyle("Add","Add", 
                Add,AddResponse)
   CALL service.publishOperation(operation,"")  ...
   # Register Service
   CALL com.WebServiceEngine.RegisterService(service)
   RETURN 0
```

Writing your functions
Write functions that implement the functions in the stub modules. It allows you to create your own version of the function.

The `ws_calculator.inc` WSDL file provides you with the global input and output records and function names that allow you to write your own code implementing the operations.

Your new code should not be written in the generated modules. For example, do not add your own version of the Add function to the generated `ws_calculator.4gl` module; it can be included in your module containing the MAIN program block, or in a separate module to be included as part of the Web server application. The function must use the generated definitions for the global input and output records.

In your version of the Add operation, for example, this function adds 100 to the sum of the variables in the input record:

```plaintext
FUNCTION Add()
   LET AddResponse.r = (Add.a + Add.b) + 100
END FUNCTION
```

See Tutorial: Writing a Server application for more information. The demo/WebServices subdirectory of your Genero installation directory contains complete examples of Server Applications.
**Writing a Web server application**
Follow examples showing you how to write a complete Web service application for the SOAP protocol.

This tutorial guides you through the steps to create a server application for a Genero Web Service that can be accessed over the web by client applications. A complete example is provided at `$FGLDIR/demo/WebServices`.

You can write your server application based on input/output records that you have defined. Or, you can use the `fglwsdl` tool to include third-party WSDL information in your server application.

**Including the web services library**
Import the `com` class.

The methods associated with creating and publishing a Web Service are contained in the classes that make up the Genero Web Services Library (com). Include this line at the top of each module of your GWS server application to import the library:

```
IMPORT com
```

**Example 1: Writing the entire server application**
Design a simple Web service.

You can define a Web Service in your application and write definitions for the input and output records that will be used by the Service. This example illustrates a Service that has one operation, `Add`, to provide the sum of two numbers.

**Step 1: Define input and output records**
Define records for the input and output messages of the Web function.

Based on the desired functionality of the operations that you plan for the Service, define the input and output records for each operation. BDL functions that are written to implement a Web Service operation cannot have input parameters or return values. Instead, each function's input and output message must be defined as a global or module RECORD.

**The Input message**
The fields of the global or module record represent each of the input parameters of the Web function. The name of each field in the record corresponds to the name used in the SOAP request. These fields are filled with the contents of the SOAP request by the Web Services engine just before executing the corresponding BDL function.

**The Output message**
The fields of the global or module record represent each of the output parameters of the Web function. The name of each field in the record corresponds to the name used in the SOAP request. These fields are retrieved from the Web Services engine immediately after executing the BDL function, and sent back to the client.

Your Genero Web Services service has one planned operation that adds two integers and returns the result. The input and output records are defined as follows:

```
GLOBALS
DEFINE
    add_in RECORD   # input record
        a INTEGER,
        b INTEGER
    END RECORD,
    add_out RECORD  # output record
        r INTEGER
    END RECORD
END GLOBALS
```

**Related concepts**
Step 2: Write a BDL function for each service operation on page 3349
Each function defines an operation of the service.

**Step 2: Write a BDL function for each service operation**

Each function defines an operation of the service.

You will need to write a function to implement each operation, using the input and output global records.

To implement your **Add** operation:

```plaintext
#User Public Functions
FUNCTION add()
    LET add_out.r = add_in.a + add_in.b
END FUNCTION
```

**Related concepts**

**Step 3: Create the service and operations** on page 3349

Describes how you provide your Web service and its operations to users who can access it on the net.

**Step 3: Create the service and operations**

Describes how you provide your Web service and its operations to users who can access it on the net.

The Genero Web Services library (**com**) provides classes and methods that allow you to use Genero BDL to configure a Web Service and its operations.

- **WebService class** - this is a container for web operations.
- **WebOperation class** - describes the operation.

**Define variables for the WebService and WebOperation objects**

```plaintext
FUNCTION createservice()
    DEFINE serv  com.WebService    # A WebService
    DEFINE op    com.WebOperation  # Operation of a WebService
END FUNCTION
```

**Choose a Namespace**

**XML** uses namespaces to group element and attribute definitions, and to avoid conflicting names. In practice, a namespace must be a unique identifier (URI: Uniform Resource Identifier). If you do not know the unique identifier to use, your company's Web site domain name is guaranteed to be unique (such as "www.mycompany.com"); then, append any string.

Examples of valid namespaces for the fictional "My Company" company:

- "http://www.mycompany.com/MyServices"
- "http://www.mycompany.com/any_string"

Another option (for testing only) is to use the temporary namespace "http://tempuri.org/".

**Create the WebService object**

Call the constructor method of the **WebService** class. The parameters are:

1. Service name
2. Valid namespace

This example uses the temporary namespace and creates a service named "MyCalculator".

```plaintext
LET serv =
```
Create the WebOperation object

A WebService object can have multiple operations. The operations can be created in RPC or Document style by calling the corresponding constructor method of the WebOperation class. The parameters are:

1. the name of the BDL function that is executed to process the XML operation
2. the name you wish to assign to the XML operation
3. the input record defining the input parameters of the operation (or NULL if there is none)
4. the output record defining the output parameters of the operation (or NULL if there is none)

To create the operation for the previously defined `add` function in RPC style:

```plaintext
LET op = com.WebOperation.CreateRPCStyle("add", "Add", add_in, add_out)
```

To create the operation for the previously defined `add` function in Document style:

```plaintext
LET op = com.WebOperation.CreateDOCStyle("add", "Add", add_in, add_out)
```

Mixing RPC style and Document style operations in the same service is not recommended, as it is not WS-I compliant. See Web Services Styles for additional information about styles.

The rest of the code in your application is the same, regardless of the Web Services style that you have chosen.

Publish the operation

Once an operation is defined, it must be associated with its corresponding WebService (the operation must be published). The publishOperation method of the WebService object has the following parameters:

- The WebOperation to be published.
- A string to identify the operation if several operations have the same name; if this is NULL, the default value is an empty string.

For example, to publish the `Add` operation of the Calculator service, which was defined as `op`:

```plaintext
CALL serv.publishOperation(op, NULL)
```

Related concepts

Step 4: Register the service on page 3350
Register the service with the Genero Web Services (GWS) server.

Step 4: Register the service
Register the service with the Genero Web Services (GWS) server.

Once the Service and operations are defined and the operations are published, the WebService and WebOperation objects have completed their work. Registering a service puts the Genero DVM in charge of the execution of all the operations of that service - dispatching the incoming message to the right service, returning the correct output, and so on. The same service may be registered at different locations on the Web.

The WebServiceEngine is a global built-in object that manages the Server part of the Genero DVM. Use the RegisterService class method of the WebServiceEngine class. The parameter is:

1. The name of the WebService object

To register the "Calculator" service, for example, created in Step 3: Create the service and operations on page 3349:

```plaintext
CALL com.WebServiceEngine.RegisterService(serv)
END FUNCTION
```

Note: If you want to create a single GWS Server DVM containing multiple Web Services, define additional input and output records and repeat steps 2 through 6 for each Web Service. In Step 5: Start the GWS server and process requests on page 3351, a GWS Server DVM is started, containing as many Web Services as you have defined.
See Web services server program deployment on page 3459 for additional discussion of GWS Services and GWS Servers.

**Related concepts**

**Step 5: Start the GWS server and process requests**

Code to start the Genero Web Services (GWS) Server.

The GWS Server is located on the same physical machine where the application is being executed (in other words, where fglrun executes).

This is the MAIN program block of your application.

**Define a variable for status**

Define a variable to hold the returned status of the request:

```main
DEFINE ret INTEGER
```

Call the function that you created, which defined and registered the service and its operations:

```main
CALL createservice()
```

**Start the GWS Server**

Use the Start class method of the WebServiceEngine class to start the server.

```main
CALL com.WebServiceEngine.Start()
```

**Process the requests**

This example uses the ProcessServices method of the WebServiceEngine class to process each incoming request. It returns an integer representing the status. The parameter specifies the timeout period (in seconds) the method may wait to process a service. The value -1 specifies an infinite waiting time.

```main
WHILE TRUE
    # Process each incoming requests (infinite loop)
    LET ret = com.WebServiceEngine.ProcessServices(-1)
    CASE ret
        WHEN 0
            DISPLAY "Request processed."
        WHEN -1
            DISPLAY "Timeout reached."
        WHEN -2
            DISPLAY "Disconnected from application server."
            EXIT PROGRAM
        WHEN -3
            DISPLAY "Client Connection lost."
        WHEN -4
            DISPLAY "Server interrupted with Ctrl-C."
        WHEN -10
            DISPLAY "Internal server error."
            EXIT PROGRAM
        WHEN -15
            DISPLAY "Server was not started."
            EXIT PROGRAM
        OTHERWISE
```
Note: For testing purposes only, the GWS Server can be started in standalone mode. In a production environment, the Genero Application Server (GAS) is required to manage your application. For deployment, the GWS Server application must be added to the GAS configuration. See the Genero Application Server User Guide.

**Example 2: Writing a server using third-party WSDL (the fglwsdl tool)**

Describes using a server stub from a third-party Web service in your GWS server application.

To write a Web Service that is compatible with the specification of the input and output records defined by a third-party (for example, a vendor of manufacturing software, or a WSDL specialist in your company), you can use the fglwsdl tool to obtain the WSDL information and generate a part of the Server application. See fglwsdl on page 2081 for a complete description of the tool and its use.

**Step 1: Get the WSDL description and generate files**

Use the fglwsdl -s tool server option to generate the BDL server stub from a WSDL.

This tutorial uses fglwsdl and the Calculator Service defined in Example 1: Writing the entire server application on page 3348 to obtain the WSDL information and generate two corresponding BDL files:

- The **globals file**, containing declarations of global variables that can be used as input or output to functions accessing the Web Service operations.
- A **.4gl file** containing a function that creates the service described in the WSDL, publishes the operations of the service, and registers the service.

`fglwsdl -s -o example1 http://localhost:8090/MyCalculator?WSDL`

Note: The "MyCalculator" Genero Web Services service created in Example 1: Writing the entire server application on page 3348 must be running in order to obtain the WSDL information.

**The generated globals file**

The globals file `example1Service.inc` provides the definition of the global input and output records as described in Step 1: Define input and output records on page 3348 of the Example 1: Writing the entire server application on page 3348 GWS Server program. The names of the input and output records have been assigned by fglwsdl, in accordance with the Style of the Web Service "MyCalculator" (created as RPC Style in the Example1 program). Do not modify this file.

Input and output records:

```plaintext
# VARIABLE : Add
DEFINE Add RECORD
    ATTRIBUTES( XMLName="Add",
                XMLNamespace="http://tempuri.org/webservices" )
    a INTEGER ATTRIBUTES(XMLName="a",XMLNamespace=""),
    b INTEGER ATTRIBUTES(XMLName="b",XMLNamespace="")
END RECORD

# VARIABLE : AddResponse
DEFINE AddResponse RECORD
    ATTRIBUTES(XMLName="AddResponse",
                XMLNamespace="http://tempuri.org/webservices" )
```
The generated .4gl file

The example1Service.4gl file contains a single function that creates the service, publishes the operation, and registers the service. The Web Service Style that is created is determined by the style specified in the WSDL information. The functions in this file accomplish the same tasks as Step 3: Create the service and operations on page 3349 and Step 4: Register the service on page 3350 of Example 1. Do not modify this file.

```4gl
# example1Service.4gl
# Generated file containing the function Createexample1Service

IMPORT com
GLOBALS "example1Service.inc"

# FUNCTION Createexample1Service
#   RETURNING soapstatus
FUNCTION Createexample1Service()
DEFINE service com.WebService
DEFINE operation com.WebOperation
# Set ERROR handler
WHENEVER ANY ERROR GOTO error
# Create Web Service
LET service = com.WebService.CreateWebService(
    "MyCalculator",
    "http://tempuri.org/webservices")

# Operation: Add
# Publish Operation : Add
LET operation = com.WebOperation.CreateRPCStyle(
    "Add",
    "Add",
    Add,
    AddResponse)
CALL service.publishOperation(operation,"")
# Register Service
CALL com.WebServiceEngine.RegisterService(service)
RETURN 0
# ERROR handler
LABEL error:
RETURN STATUS
# Unset ERROR handler
WHENEVER ANY ERROR STOP
END FUNCTION
```

Related concepts

**Step 2: Write a BDL function for your service operation** on page 3353
Write functions that implement the functions in the stub file. This allows you to create your own version of the function.

**Step 2: Write a BDL function for your service operation**
Write functions that implement the functions in the stub file. This allows you to create your own version of the function.

Using the information from the files generated in Step 1: Get the WSDL description and generate files on page 3352, the Add operation from Example 1: Writing the entire server application on page 3348 is rewritten to have different functionality but to still be compatible with the WSDL description of the operation.
This step accomplishes the same thing as Step 2: Write a BDL function for each service operation on page 3349 in Example 1. In this version of the add operation, the sum of the two numbers in the input record is increased by 100.

```plaintext
# my_function.4gl -- file containing the function
IMPORT com -- definition
IMPORT the Web Services library
GLOBALs "example1Service.inc" -- use the generated globals file
#User Public Functions
FUNCTION add() -- new version of the add function
LET AddResponse.r = (Add.a + Add.b) + 100 -- the global input and output
END FUNCTION
```

Related concepts

Step 3: Create service, start server and process requests on page 3354
Code to start the Genero Web Services (GWS) Server.

Step 3: Create service, start server and process requests
Code to start the Genero Web Services (GWS) Server.

Create your own MAIN module that calls the function from the generated .4gl file to create the service, then starts the Genero Web Services Server and manages requests as in Step 5: Start the GWS server and process requests on page 3351 of Example 1: Writing the entire server application on page 3348.

```plaintext
# example2main.4gl file -- contains the MAIN program block
IMPORT com
GLOBALs "example1Service.inc"
MAIN
    DEFINE create_status INTEGER
    DEFER INTERRUPT
    CALL Createexample1Service() -- call the function generated
        RETURNING create_status -- in example1Service.4gl
        IF create_status <> 0 THEN
            DISPLAY "error"
        ELSE
            # Start the server and manage requests
            CALL ManageService()
        END IF
END MAIN

FUNCTION ManageService()
    DEFINE ret INTEGER
    CALL com.WebServiceEngine.start()
    WHILE TRUE
        # continue as in Step 5 of Example 1
        ...
END FUNCTION
```

Enabling MTOM on the server side
Enable the Message Transmission Optimization Mechanism (MTOM) feature to efficiently send binary data to and from Web services.

By default, MTOM is not enabled. As a result, any BYTE represented as xsd:base64 or xsd:hexbinary will be inlined in the SOAP request or response.
To enable MTOM on server side, set the MTOM feature to TRUE.

```
# Create Web Service
LET service =
CALL service.setFeature("MTOM",true)
```

With MTOM enabled on the server side, the generated WSDL includes the MTOM policy.

**Compiling GWS server applications**

When compiling, remember to link in the WSHelper library.

It is recommended that the library file WSHelper.42m, included in the $FGLDIR/lib directory of the Genero Web Services package, is linked into every GWS Server application.

If your application uses the fglwsdl tool to generate information, link the .4gl generated file into the application.

**Examples**

Compiling the Example 1: Writing the entire server application on page 3348 program:

```
fglcomp example1.4gl
fgllink -o example1.42r example1.42m WSHelper.42m
```

Compiling the Example 2: Writing a server using third-party WSDL (the fglwsdl tool) on page 3352 program:

```
fglcomp example2main.4gl my_function.4gl example1Service.4gl
fgllink -o example2.42r example2main.42m my function.42m example1Service.42m WSHelper.42m
```

**Testing the GWS service in stand-alone mode**

Test that your service is reachable and that it can generate the WSDL.

For testing and development purposes only, the Genero Web Services Server application can be executed directly, without using the Genero Application Server (GAS).

1. Use the Genero fglrun command to execute the GWS Server application. The application must reside on the same machine:

```
fglrun <gws-application>
```

This will start the GWS Server on the port specified by the FGLAPPSERVER environment variable. For example, if FGLAPPSERVER is set to 8090, the server will be started on that port. If this environment variable is not set for the user, port number 80 is used.

**Warning:** The user must not set the FGLAPPSERVER variable in production environments, since the port number is selected by the Genero Application Server.

2. Obtain the WSDL information for your Service and write a test Client application. For example, if the GWS Server in step 1 was started on your local machine and FGLAPPSERVER was set to 8090, the command to get the WSDL information would be:

```
fglwsdl -o <test-client> http://localhost:8090/<service-name>?WSDL
```

**Configuring the Genero Application Server for the GWS application**

Prepare for a production environment.

The final step is to configure the Genero Application Server (GAS) to handle the GWS application. In a production environment, Genero Web Services becomes a part of a global application architecture handled by the application server of the GAS package. See Web services server program deployment on page 3459, as well as Packaging web applications on page 2183 and Deploying apps with Genero Archive in the Genero Application Server User Guide.
Making the GWS service available
Provide your Web service to users who can access it on the net.

Once you compiled and deployed your Genero Web Services Server application (see Web services server program deployment on page 3459), it can be used by others to obtain the WSDL information and write a client application that accesses your Genero Web Service. See Steps to write a WS client on page 3321.

Your company can provide the location of the GWS Server to potential users of your Web Service in various ways. For example:

- Provide the location on a company web site
- Register the Web Service with UDDI (Universal Description, Discovery, and Integration) - the XML-based registry providing Internet listings for companies worldwide
- Communicate directly with your potential users

Get HTTP headers information at WS server side
To work with HTTP headers exchanged between the Genero client and a web service, you must register variables to receive and send them.

In high level (SOAP) Web services, we now give access to the HTTP headers request and response.

The Web service can get information from the request headers and reply with custom headers and status.

1. Declare variables to receive or send HTTP headers.
2. Register these variables to the Web service server.

Declare variables to receive or send HTTP headers
The variable for the request headers:

```
DEFINE http_in RECORD
    verb STRING,
    url STRING,
    headers DYNAMIC ARRAY OF RECORD
        name STRING,
        value STRING
    END RECORD
END RECORD
```

After the Web service operation has been processed, the variable is set to NULL.

The variable for the response headers:

```
DEFINE http_out RECORD
    code INTEGER,
    desc STRING,
    headers DYNAMIC ARRAY OF RECORD
        name STRING,
        value STRING
    END RECORD
END RECORD
```

After the Web service operation has been processed, the variable is set to NULL.

**Note:** While the variables must follow the structure shown, the variable name can be any name you choose.

The Web service engine headers have precedence. For example, if you set the "Content-Length" value, the one that is taken into account is the one defined by the Genero Web Services engine.

Register the variables to the server
This code example uses two methods, which use the defined variables:
• `com.WebService.registerInputHttpVariable(http_in)` where `http_in` is the RECORD variable for the request headers.

• `com.WebService.registerOutputHttpVariable(http_out)` where `http_out` is the RECORD variable for the response headers

Example

FUNCTION CreateService()

    DEFINE serv com.WebService # WebService
    DEFINE op com.WebOperation # Operation of a WebService

    TRY
        # Create a Web Service
        LET serv = com.WebService.CreateWebService("EchoHttpHeadersService", Namespace)

        # Create Document Style Operations
        # EchoDOCRecord
        LET op = com.WebOperation.CreateDOCStyle("echoDocRecord", "EchoDOCRecord", echoRecordDoc_in, echoRecordDoc_out)

        CALL serv.publishOperation(op,NULL)

        # Register HTTP input
        CALL serv.registerInputHttpVariable(http_in)

        # Register HTTP output
        CALL serv.registerOutputHttpVariable(http_out)

        # Register service
        CALL com.WebServiceEngine.RegisterService(serv)
        DISPLAY "EchoHttpHeadersService Service registered"
    CATCH
        DISPLAY "Unable to create 'EchoHttpHeadersService' Web Service : ", STATUS||"("||SQLCA.SQLERRM||")"
        EXIT PROGRAM (-1)
    END TRY

END FUNCTION

FUNCTION echoDocRecord()

    DEFINE ind INTEGER
    DEFINE ok BOOLEAN

    # Check incoming VERB
    IF http_in.verb != "POST" THEN
        LET http_out.code = 400
        LET http_out.desc = "Bad request: method should be POST"
        RETURN
    END IF

    # Check incoming query string
    IF http_in.url.getIndexOF("?MyQuery=OK",1)<=0 THEN
        LET http_out.code = 400
        LET http_out.desc = "Bad request: URL should have MyQuery=OK"
        RETURN
    END IF
# Check incoming header called MyPersonal
LET ok = FALSE
FOR ind = 1 TO http_in.headers.getLength()
    DISPLAY ind||" ",http_in.headers[ind].name,
    "=" http_in.headers[ind].value
    IF http_in.headers[ind].name == "MyPersonal" THEN
        IF http_in.headers[ind].value == "Header" THEN
            LET ok = TRUE
        END IF
    END IF
END FOR
IF NOT ok THEN
    LET http_out.code = 400
    LET http_out.desc = "Bad request: expected additional header called MyPersonal"
    RETURN
END IF

# assign the output record
LET echoRecordDoc_out.MyRecord.MyInt = echoRecordDoc_in.MyRecord.MyInt
LET echoRecordDoc_out.MyRecord.MyFloat = echoRecordDoc_in.MyRecord.MyFloat

# Add MyPersonalHeader=MyPersonalValue http headers
LET http_out.headers[1].name = "MyPersonalHeader"
LET http_out.headers[1].value = "MyPersonalValue"

END FUNCTION

Related concepts
The WebService class on page 2811
The com.WebService class provides an interface to create and manage Genero Web Services.

Choosing a web services style
Genero Web Services contains style options for creating SOAP Web services. Your choice is dependent on the type of service, (Document or RPC), and the encoding mechanism (literal or encoded) required.

Genero Web Services 2.0 allows you to create SOAP Web services operations in the following styles:

Table 717: Web Services Styles

<table>
<thead>
<tr>
<th>Web Services Style</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remote Procedure Call (RPC) Style Service (RPC/Literal)</td>
<td>Generally used to execute a function, such as a service that returns a stock option.</td>
</tr>
<tr>
<td>Document Style Service (Doc/Literal)</td>
<td>Generally used for more sophisticated operations that exchange complex data structures, such as a service that sends an invoice to an application, or exchanges a Word document; this is the MS.Net default.</td>
</tr>
<tr>
<td></td>
<td>Both RPC/Literal and Doc/Literal Styles are Web Services Interoperability (WS-I) organization compliant.</td>
</tr>
</tbody>
</table>
The style of service to be created is specified in the Genero application for the Web Service, using the following methods of the `WebOperation` class from the (Web Services COM Library). The parameters are the same for both methods:

1. The name of the BDL function that is executed to process the Web service operation
2. The name you wish to assign to the Web service operation
3. The `input` record defining the input parameters of the operation (or NULL if there is none)
4. The `output` record defining the output parameters of the operation (or NULL if there is none)

```
LET op = com.WebOperation.CreateRPCStyle("add","Add",add_in,add_out)
LET op = com.WebOperation.CreateDOCStyle("checkInvoice","CheckInvoice",invoice_in,invoice_out)
```

Calling the appropriate function for the desired style is the only difference in your Genero code that creates the service. The remainder of the code that describes the service is the same, regardless of whether you want to create an RPC or Document style of service.

**Important:** Do not use the `setInputEncoded()` and `setOutputEncoded()` methods of the `WebService` class from the Web Services COM Library (`com`), as they apply only to RPC/Encoded Style, which is not recommended.

**Note:** If you add headers to your RPC Style service, choose the Literal serialization mechanism by setting the `encoded` parameter of the `createHeader()` method to FALSE:

```
CALL serv.createHeader(var,FALSE)
```

**Note:** GWS release 2.0 allows you to create RPC Style and Document Style operations in the same Web service. However, we do not recommend this, as it is not WS-I compliant.

## How Do I ... ?

These topics provide you with the information needed to perform specific tasks related to Genero Web Services using the SOAP protocol.

**Tip:** See [here](#) for other Web services "How Do I … ?" topics.

### Fix Genero 2.10 to 2.11 WSDL generation issue

How to convert a WSDL generated from a Genero 2.11 (or later) application to a WSDL as generated by Genero 2.10.

Since Genero 2.11, each BDL variable generates an associated named complexType in the WSDL and references it. While this does not impact the Web service, some tools will generate additional client stubs to follow the WSDL definition with the named complexType. As a result, any client program written from a WSDL generated in 2.10 must be reviewed if it uses a WSDL generated in 2.11 or later.

If you do not want to modify your application, you can use the WSDL conversion tool program to remove the named complexType and add the unnamed equivalent as child node of the parameter variable of all Web service operations, as if the WSDL had been generated in 2.10.
**WSDL conversion tool**

This program reads a WSDL, looks for all named complexType used in all the web operation parameters and modifies them in order to have unnamed complexType instead.

```plaintext
IMPORT XML
MAIN
DEFINE
    doc xml.DomDocument,
    list, elist, tlist xml.DomNodeList,
    node, enode, nnode xml.DomNode,
    i, j, k, idx INTEGER,
    ename, tname STRING

IF num_args() <> 1 THEN
    CALL display_help()
    RETURN 0
END IF

TRY
    LET doc = xml.DomDocument.Create()
    CALL doc.setFeature("whitespace-in-element-content",FALSE)
    CALL doc.load(arg_val(1))
    # get the list of input/output message
    # check if their names (x) are defined as elements with types (y)
    # if yes then
    # copy the complextype y definition to element name x
    # and remove the complexe type y definition
    # for example:
    # message
    # <wsdl:message name="is_OKIn">
    #   <wsdl:part name="parameters" element="fjs:is_OKRequest" />
    # </wsdl:message>
    # <wsdl:message name="is_OKOut">
    #   <wsdl:part name="parameters" element="fjs:is_OKResponse" />
    # </wsdl:message>
    # element
    # <xsd:element name="is_OKResponse"
    #   type="s1:is_OKResponse_is_OKResponse" />
    # type
    # <xsd:complexType name="is_OKRequest_is_OKRequest">
    LET list =
        doc.selectByXPath("//wsdl:part[@name='parameters']/@element",
                         "wsdl","http://schemas.xmlsoap.org/wsdl/")
    IF list IS NULL THEN
        DISPLAY "Nothing to convert."
    END IF
    FOR i=1 TO list.getCount()
        LET node = list.getItem(i)
        LET ename = node.getNodeValue()
        LET idx = ename.getIndexOf(":",1)
        IF idx <> 0 THEN
            LET ename = ename.subString(idx+1,ename.getLength())
        END IF
        # get the element
        LET elist =
            doc.selectByXPath("//xsd:element[@name='" || ename || "]", "xsd","http://www.w3.org/2001/XMLSchema")
        IF elist IS NOT NULL THEN
            FOR j=1 TO elist.getCount()
                LET enode = elist.getItem(j)
                LET tname = enode.getAttribute("type")
                CALL enode.removeAttribute("type")
            END FOR
        END IF
    END FOR
```
Example of use:

$ fglrun Convert Genero2_21.wsdl
$ Document is saved in result.wsdl

How to handle WS security

The Genero Web Services engine does not entirely manage WS-Security; however, Genero BDL provides XML APIs to help the development of Web Services with security. Exploring the demo Web Service included with the FGLGWS installation will help you handle security in your own SOAP Web service.

These topics describe how to handle Web Services security using the wssecurermessage demo, located in $FGLDIR/demo/WebServices. See the readme.txt file provided with the demo for information and instructions on how to run the service.

You are encouraged to treat the demo as an example that you can adapt to your needs. It is based on the WS-Security (WSS) standard. For more information, refer to the WS-Security Policy documentation.

The demo demonstrates a secure messaging service using a WS-Security policy to exchange messages. It involves three clients exchanging secured messages. Those clients post and retrieve messages on a secured server. Each client is identified by a certificate that signs its messages.

The demo assumes that all the clients have sent their public keys to the other clients and to the server. Those keys are kept in each host's (server or clients) keystore.
As a prerequisite, we recommend that you become familiar with security concepts described in the Encryption and Authentication Concepts page.

**Important:** The certificates included in this package are provided for demonstration purposes only. As they are distributed with this package, anybody using this product can decrypt the messages exchanged. Do NOT use them in production.

**Server side**
Learn the steps the server uses to process messages it sends and receives. Identify the Genero handlers that help implement the security policy on the server side.

We provide 3 handlers to handle WS Security:

- Method `com.WebService.registerWSDLHandler()` to modify the WSDL to add WS policy.

In this demo (located in `$FGLDIR/demo/WebServices/wssecuritymessage`), a received message is processed:

1. Identify the sender and validate the sender (search in keystore)
2. Decrypt the symmetric key with the server private key
3. Decrypt the body
4. Check the signature with the sender public key
5. Store the message in the box (thanks to the "To" field, "subject" and "message")
6. Create the outgoing message
7. Sign the outgoing message
8. Encrypt the outgoing message with a generated symmetric key. This symmetric key is then encrypted with the client public key.

**Client side**
From the client side, identify the steps that need to be performed to send and receive secure messages.

The client function consists of sending a message and retrieving messages clients have sent to it.

Before you begin, create the client stub from the WDSL:

```
• fglwsdl -domHandler myservice.wsdl
```

The client stub references callback handlers:

- `SecureMessageBox_HandleRequest`
- `SecureMessageBox_HandleResponse`
- `SecureMessageBox_HandleResponseFault`

For more details about client SOAP handlers see Client stub and handlers.

What to do when a message is sent:

1. Sign and encrypt the request for the server (WS-Security)
   - sign with client private key
   - encrypt with server public key
2. Send key information in the request
   - key to identify the sender/client
   - key to identify the recipient/server
   - key used to encrypt the data (usually a symmetric key encrypted by the recipient public key)
3. If the message has to be encrypted for the final recipient (XML-Security)
   - sign the message
   - encrypt the message
What to do to retrieve messages:
1. Identify the sender and validate the sender (search in keystore)
2. Identify the recipient (should be the server itself)
3. Decrypt the request
4. Check the signature
5. Retrieve messages for the recipient

**SOAP security standards**
In this section read about the security policy standards, such as bindings, and the options for signing and encryption.

A Web service security policy requirements are included in the Web Services Description Language (WSDL) file. The policy is divided into these sections:

- Security bindings
- Security options
- Signed parts
- Encrypted parts

The following defines a policy:

```xml
```

The following section includes policy rules.

```xml
<wsp:ExactlyOne>
<wsp:All>
```

Only one assertion is fulfilled.

```xml
<wsp:All>
```

All the assertions included within this tag are fulfilled.

**Demo WSDL**
For a complete WS security policy document, see the WSDL file included with the demo "wssecuritymessage", located in `$FGLDIR/demo/WebServices`. For more information on the standard, refer to [WS-SecurityPolicy](#).

**Security bindings**
Understand the mechanism of bindings that allows the secure exchange of SOAP messages over HTTP. The demo application security policy is referenced to illustrate this.

There are 3 types of security bindings:

- TransportBinding
- SymmetricBinding
- AsymmetricBinding

The demo (located in `$FGLDIR/demo/WebServices/wssecuritymessage`) uses Asymmetric binding.

**Asymmetric Binding**
This section of the security policy is divided in sub sections:

- InitiatorToken
- RecipientToken
- AlgorithmSuite
- Layout
- Additional assertions
AsymmetricBinding is the root node of the security policy for protection description.

```xml

InitiatorToken
InitiatorToken is the message sender (client)
For example:

```xml
<sp:InitiatorToken>
  <wsp:Policy>
    <sp:X509Token sp:IncludeToken="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/IncludeToken/AlwaysToRecipient">
      <wsp:Policy>
        <sp:RequireThumbprintReference />
        <sp:WssX509V1Token10 />
      </wsp:Policy>
    </sp:X509Token>
  </wsp:Policy>
</sp:InitiatorToken>
```

**Note:** The value for the `sp:IncludeToken` attribute is one contiguous string with no spaces. For this document, it is shown covering two lines.

The token is used for the message signature from initiator to recipient and encryption from recipient to initiator.

The initiator key is a X509 certificate that is always sent to the recipient.

- `sp:IncludeToken` attribute indicates if the token must be included.
- `IncludeToken/AlwaysToRecipient` means each requests sent to the recipient must include the initiator token. But the token is not to be included in messages from recipient to initiator.

The token must send its Thumbprint Reference.

The token must be of type X509 version 1 as defined in "X509 token profile 1.0".

What needs to be done in BDL is described in **Client Side** section.

To retrieve the thumbprint reference you can use the API function `xml.CryptoX509.getThumbprintSHA1`

To create the x509 certificate, use an appropriate tool like `openssl`.

RecipientToken
RecipientToken is the message receiver (server)

```xml
<sp:RecipientToken>
  <wsp:Policy>
      <wsp:Policy>
        <sp:RequireThumbprintReference />
        <sp:WssX509V3Token10 />
      </wsp:Policy>
    </sp:X509Token>
  </wsp:Policy>
</sp:RecipientToken>
```

**Note:** The value for the `sp:IncludeToken` attribute is one contiguous string with no spaces. For this document, it is shown covering two lines.
The token is used for encryption from initiator to recipient, and for the message signature from recipient to initiator. The recipient key is a X509 certificate that is never sent to the initiator.

`sp:IncludeToken` attribute indicates if the token must be included.

Use of the `IncludeToken/Never` means the token is never to be included in any requests between the initiator and the recipient. This is the required and recommended setting.

Instead the recipient ThumbprintReference is sent.

The token must be of type X509 version 3 as defined in "X509 token profile 1.0"

What needs to be done in BDL is described in Server Side section. To retrieve the thumbprint reference you can use the API function `xml.CryptoX509.getThumbprintSHA1`. To create the appropriate certificate use an appropriate tool like `openssl`.

**AlgorithmSuite**

AlgorithmSuite specifies which algorithm is used to encrypt the data.

```xml
<sp:AlgorithmSuite>
   <wsp:Policy>
      <sp:TripleDesRsa15 />
   </wsp:Policy>
</sp:AlgorithmSuite>
```

`TripleDesRsa15` refers to key `http://www.w3.org/2001/04/xmlenc#tripledes-cbc`.

**Layout**

Layout describes the way information is added to the message header.

```xml
<sp:Layout>
   <wsp:Policy>
      <sp:Strict />
   </wsp:Policy>
</sp:Layout>
```

For example, with `Strict` layout, tokens that are included in the message must be declared before use. For more details on the rules to follow see the WS-SecurityPolicy specifications section 7.7.

**Additional Assertions**

PartsToSign

```xml
<sp:OnlySignEntireHeadersAndBody />```

The assertion means if there is any signature on the header or the body it applies to the entire header and the entire body not to their child element.

**SOAP message security options**

Describes the Wss10 SOAP Message Security 1.0 options that are supported.

```xml
   <sp:MustSupportRefKeyIdentifier />
   <sp:MustSupportRefIssuerSerial />
</sp:Wss10>
```

- `MustSupportRefKeyIdentifier` means that initiator and recipient are able to generate and process key identifier reference.
- `MustSupportRefIssuerSerial` means that initiator and recipient are able to generate and process issuer and token serial reference.
**SignedParts**
The SignedParts section of the policy specifies which part of the message should be signed.

```xml
  <sp:Body />
</sp:SignedParts>
```

- Only the body needs to be signed

**EncryptedParts**
The EncryptedParts section of the policy specifies which part of the message should be encrypted.

```xml
  <sp:Body />
</sp:EncryptedParts>
```

- `sp:Body` indicates the body message needs to be encrypted

Encrypt the body using the algorithm referenced in assertion **AlgorithmSuite**:

- Create an encryption key using TripleDesRsa15 algorithm (it generates a TripleDES symmetric key and then encrypts it with a RSA1.5 public key), like in example2 that uses AES256 in the CryptoKey chapter.
- Encrypt the body with the created key.

To find the exact syntax of security message read the specifications "Web Services Security: SOAP Message Security 1.0".

**Useful links**
For more information on SOAP message security, follow the links to the policy standards.

- Security Policy specifications v1.2
- SOAP Message Security 1.0
- X.509 Token Profile 1.1

**How to migrate I4GL web service to Genero**
Provides instructions for migrating Web Services and client application from I4GL to Genero BDL.

**Migrate an I4GL web service provider to Genero**
Migrate a Web service from I4GL to Genero BDL using the SOAP protocol.

This section explains how to migrate an I4GL Web service provider to a Genero application providing the same Web service; allowing access to all clients already accessing that service - unmodified (excepted for the host name of course).

**Note:** The migration is based on the SOA zipcode demo in the I4GL package.

**Step 1: Use the I4GL function and the I4GL.4cf configuration file**
The I4GL .4cf configuration file has all the information you need about the I4GL Web service.

For example, the I4GL zipcode demo has the following .4cf configuration file:

```plaintext
[SERVICE]  TYPE            = publisher
          INFORMIXDIR    = /dbs/32bits/ifx/11.70.uc2
          DATABASE      = i4glsoa
          CLIENT_LOCALE = en_US.8859-1
          DB_LOCALE     = en_US.8859-1
          INFORMIXSERVER = ol_moscou1170uc2
          HOSTNAME      = moscou.strasbourg.4js.com
          PORTNO        = 9876
          I4GLVERSION   = 7.50.xC4
          WSHOME        = /dbs/32bits/ifx/11.70.uc2/AXIS2C
          WSVISION      = AXIS1.5
          TMPDIR        = /tmp/zipcodedemo
```
SERVICENAME = ws_zipcode

[FUNCTION]
NAME = zipcode_details

[INPUT]
[VARIABLE] NAME = pin TYPE = CHAR(10) [END-VARIABLE]
[END-INPUT]

[OUTPUT]
[VARIABLE] NAME = city TYPE = CHAR(100) [END-VARIABLE]
[VARIABLE] NAME = state TYPE = CHAR(100) [END-VARIABLE]
[END-OUTPUT]

[END-FUNCTION]

[DIRECTORY]
NAME = /home/f4gl/fg/i4gl
FILE = soademo.4gl,
[END-DIRECTORY]

[END-SERVICE]

Then simply copy your I4GL function without any modification into a new Genero file and add the Genero IMPORT com instruction at the beginning of the file.

For example, the I4GL SOA demo contains the zipcode_details service (soademo.4gl)

IMPORT com

FUNCTION zipcode_details(pin)
DEFINE state_rec RECORD
    pin CHAR(10),
    city CHAR(100),
    state CHAR(100)
END RECORD,

pin CHAR(10),
sel_stmt CHAR(512);

LET sel_stmt= "SELECT * FROM statedetails WHERE pin = ?";
PREPARE st_id FROM sel_stmt;
DECLARE cur_id CURSOR FOR st_id;
OPEN cur_id USING pin;
FETCH cur_id INTO state_rec.*;
CLOSE cur_id;
FREE cur_id;
FREE st_id;
RETURN state_rec.city, state_rec.state

END FUNCTION

Note: You may need to make some minor code modification for compatibility.

Step 2: Create a BDL RECORD for the input parameters

Define a BDL record for the input message of the Web function.

Add a new modular BDL record where all members map to one of your I4GL Web service input parameter, and keep the parameter order as defined in I4GL .4cf file.

You must then specify the Web service input message name via the Genero XML attribute called XMLName, and assign it to the FUNCTION NAME as defined in the I4GL .4cf file.

For example, in the I4GL zipcode demo there is only one parameter: pin. So add the following record at the beginning of the Genero file:

DEFINE zipcode_details_in RECORD ATTRIBUTES(XMLName="zipcode_details")
    pin CHAR(10)
END RECORD
**Note:** Genero Web Services supports complex data types as input parameters.

**Step 3: Create a BDL RECORD for the output parameters**
Define a BDL record for the output message of the Web function.

Add another modular BDL record where all members map to one of your I4GL Web service output parameter, and keep the parameter order as defined in I4GL .4cf file.

You must then specify the Web service output message name via the Genero XML attribute called `XMLName`, and assign it to the FUNCTION NAME as defined in the I4GL .4cf file concatenated to response.

For example, in the I4GL zipcode demo there are two parameters: `city` and `state`. So add the following record at the beginning of the Genero file:

```coffice
DEFINE zipcode_details_out RECORD
  ATTRIBUTES(XMLName="zipcode_detailsresponse")
  city CHAR(100),
  state CHAR(100)
END RECORD
```

**Note:** Genero Web Services supports complex data types as output parameters.

**Step 4: Create a BDL wrapper function**
Create the wrapper function that uses BDL records to call the I4GL function.

Create a Genero BDL wrapper function without any parameters that will then use the input and output record created at Step 2 and Step 3 to call the I4GL function passing it the parameters retrieved from the records.

For example, in the I4GL zipcode demo there are 1 input and 2 output parameters. So the BDL wrapper function must use these records to call the I4GL function as follows:

```coffice
FUNCTION zipcode_details_g()
  CALL zipcode_details(zipcode_details_in.pin)
  RETURNING zipcode_details_out.city,zipcode_details_out.state
END FUNCTION
```

**Step 5: Publish the wrapper function as a Genero web service**
Based on the details in the configuration file (.4cf) file, create a function that registers your Web service with the Genero Web Service server.

Use the COM APIs to publish the I4GL function as a Web service based on I4GL .4cf configuration file to get a compatible Genero Web service.

To create a new BDL function in charge of the service publication, you will need the following elements of the I4GL .4cf configuration file:

- The name of the service that is defined in the `SERVICENAME` entry
- The namespace of the service that is defined as `http://www.ibm.com/` concatenated to the `FUNCTION NAME`
- The name of the function to be published that is defined in the `FUNCTION NAME` entry

For example, the I4GL zipcode demo has one function published as a Doc/Literal service.

```coffice
FUNCTION create_zipcode_details_web_service()
  DEFINE serv com.WebService
  DEFINE op com.WebOperation
  # Create the web service based on the entries of the .4cf file
  # SERVICENAME: The name of service is 'ws_zipcode'
  # FUNCTION NAME: The namespace of the service is built from
  # the base url 'http://www.ibm.com/' concatenated to
  # the NAME of the I4GL function 'zipcode_details'
```
    "http://www.ibm.com/zipcode_details")

# Create and publish the Doc/Literal web function based on
# step 2, step 3 and step 4
# and from the FUNCTION NAME defined in the .4cf file
# LET op = com.WebOperation.CreateDOCStyle("zipcode_details_g",
#     "zipcode_details",
#     zipcode_details_in,
#     zipcode_details_out)
# CALL serv.publishOperation(op,NULL)
# Register the service into the SOAP engine
# CALL com.WebServiceEngine.RegisterService(serv)
END FUNCTION

Note: I4GL supports only Doc/Literal services.

Note: Genero Web Services can contain several BDL functions in the same service. In other words, you can group several I4GL services in the same Genero service.

Step 6: Create the server
Provide a file with a BDL function that starts your Web service with the Genero Web Service server instead of Axis.

I4GL uses Axis as server for its services, but Genero has its own server programmable via the COM library. Create a new file and add the IMPORT com instruction at the beginning of the server file, then simply create the main loop in BDL that will process any incoming HTTP request.

The port of the service defined in the I4GL .4cf configuration file (via the PORTNO entry) can be reused by setting the FGLAPPSERVER environment variable to the same value before running the server. However, only on development or for tests, on production Genero Web Services requires an application server called GAS in charge of load balancing. See the Genero Application Server User Guide for more details about port configuration for deployment purpose.

For example, to migrate the I4GL zipcode demo, the service must be created in the server before running the main loop as follows:

MAIN
    DEFINE ret INTEGER
    DEFER INTERRUPT

    # Create zipcode_details service
    CALL create_zipcode_details_web_service()

    # Start the server on port set in FGLAPPSERVER
    # (to be set to same value as PORTNO defined in the .4cf file)
    CALL com.WebServiceEngine.Start()

    # Handle any incoming request in a WHILE loop...
    WHILE TRUE
        LET ret = com.WebServiceEngine.ProcessServices(-1)
        CASE ret
            WHEN 0
                DISPLAY "Request processed."
            WHEN -1
                DISPLAY "Timeout reached."
            WHEN -2
                DISPLAY "Disconnected from application server."
EXIT PROGRAM  # The Application server has closed the connection
WHEN -3
  DISPLAY "Client Connection lost."
WHEN -4
  DISPLAY "Server interrupted with Ctrl-C."
WHEN -10
  DISPLAY "Internal server error."
END CASE
IF int_flag<>0 THEN
  LET int_flag=0
  EXIT WHILE
END IF
END WHILE
END MAIN

Note: With Genero Web Services, one server can contain several services. In other words, you can put all your I4GL services into one server.

Related information
Process the requests on page 3351

Step 7: Configure the database
Based on the DATABASE entry in the I4GL .4cf configuration file, use the Genero instruction to connect to the Informix® database at server startup.

For example, in the I4GL zipcode demo the service accesses the database called: "i4glsoa". So add the following instruction at the beginning of the server file created in step 6:

DATABASE i4glsoa
MAIN
...
END MAIN

Step 8: Compile and run the Genero service
Describes how to compile and run the service to test it.

Compile and link the 2 Genero files created and run your Genero service. It will be directly available for any client, and will provide the WSDL when requested via a HTTP GET with WSDL as query string.

Example
The Genero web service is accessible on URL: http://hostname:9876/ws_zipcode and can return the WSDL on URL: http://hostname:9876/ws_zipcode?WSDL.

$ fglcomp -M genero_service.4gl
$ fglcomp -M genero_server.4gl
$ fgllink -o genero_zipcode genero_service.42m genero_server.42m
$ export FGLAPPSEVER=9876
$ fglrun genero_zipcode.42r

Note: The host name depends on the machine your Genero application is started.

Note: For deploying the service on production sites you will need the Genero application server (GAS) to load-balance the service. See the Genero Application Server User Guide about Web services when deployment is required.

Migrate an I4GL web service consumer to Genero
Migrate a client application from I4GL to a Genero BDL application using the SOAP protocol.

This section explains how to migrate an I4GL Web service consumer to a Genero application accessing the same Web service.

Note: The migration is based on the SOA demo in the I4GL package.
Step 1: Generate the Genero web service stub from an I4GL WSDL

Use the fglwsdl tool to get the WSDL information from the service provider.

Use the I4GL WSDL located on the Axis server to generate the Genero Web service client stub via the tool called fglwsdl.

For example, the WSDL file of the I4GL zipcode demo is located on $INFORMIXDIR/AXIS2C/services/ws_zipcode/zipcode_details.wsdl. So run the following command:

```
$ fglwsdl -noFacets zipcode_details.wsdl
```

It will generate these two Genero files:

- `ws_zipcode_zipcode_detailsservice.4gl`
  - It contains the Genero functions to connect to the server in SOAP over HTTP.
  - Take a look at the file if you are interested in Genero HTTP and XML low-level APIs.
- `ws_zipcode_zipcode_detailsservice.inc`
  - It contains the Genero data types and variables used for XML serialization.
  - Take a look at the file if you are interested in Genero XML to BDL variable mapping.

Note: Option `-noFacets` is required for this demo because the I4GL CHAR data type will be generated as string in Genero, which can lead to an XML serialization error if not present.

Step 2: Modify the Genero .inc stubs to fix wrong I4GL WSDL

Remove the namespace attributes in the .inc file stub.

The I4GL WSDL contains namespace declaration for all I4GL Web service data types, but in practice the I4GL axis server doesn’t care about namespaces, but Genero does. So you have to open the generated Genero .inc file and remove all attributes called `XMLNamespace` and `XSTypeNamespace`.

For example, the generated .inc file from the I4GL WSDL must be modified as shown in the following sample:

```plaintext
# File: ws_zipcode_zipcode_detailsservice.inc
# GENERATED BY fglwsdl 101601
# THIS FILE WAS GENERATED. DO NOT MODIFY.

GLOBALS
...

# TYPE : tzipcode_details

TYPE tzipcode_details RECORD
  ATTRIBUTES(XMLSequence,XSTypeName="zipcode_details")
  #,XSTypeNamespace="http://www.ibm.com/zipcode_details")
  pin STRING ATTRIBUTES(XMLName="pin")
END RECORD

# TYPE : tzipcode_detailsresponse

TYPE tzipcode_detailsresponse RECORD
  ATTRIBUTES(XMLSequence,XSTypeName="zipcode_detailsresponse")
  #,XSTypeNamespace="http://www.ibm.com/zipcode_details")
  city STRING ATTRIBUTES(XMLName="city")
  state STRING ATTRIBUTES(XMLName="state")
END RECORD
```
# Operation: zipcode_details

# FUNCTION: zipcode_details_g()
#   RETURNING: soapStatus
#   INPUT: GLOBAL zipcode_details
#   OUTPUT: GLOBAL zipcode_detailsresponse
#
# FUNCTION: zipcode_details(p_pin)
#   RETURNING: soapStatus ,p_city ,p_state
#
# FUNCTION: zipcode_detailsRequest_g()
#   RETURNING: soapStatus
#   INPUT: GLOBAL zipcode_details
#
# FUNCTION: zipcode_detailsResponse_g()
#   RETURNING: soapStatus
#   OUTPUT: GLOBAL zipcode_detailsresponse
#
# VARIABLE : zipcode_details
DEFINE zipcode_details tzipcode_details
ATTRIBUTES(XMLName="zipcode_details")
#,XMLNamespace="http://www.ibm.com/zipcode_details")
#--------------------------------------------------------------

# VARIABLE : zipcode_detailsresponse
DEFINE zipcode_detailsresponse tzipcode_detailsresponse
ATTRIBUTES(XMLName="zipcode_detailsresponse")
#,XMLNamespace="http://www.ibm.com/zipcode_details")
END GLOBALS

Note: For Genero Web Services options for mapping BDL and XML, see XML mapping attributes.

**Step 3: Include the generated stub in your I4GL application**

Use a GLOBALS statement to specify the Web service your application uses.

Add in all I4GL files calling a Web service in the generated .inc stub with a GLOBALS instruction.

For example, in the I4GL zipcode demo, only the clsoademo.4gl file uses Web services. So add the following line at the beginning of the file:

GLOBALs "ws_zipcode_zipcode_detailsservice.inc"
MAIN
...
END MAIN

Note: This allows access to the Genero global variables and data types used in the Web service call, so the Genero global wsError record can retrieve error codes if any.

**Step 4: Modify the I4GL web service function call**

Rename I4GL function name names to Genero Web service function names.

The Genero Web service function name is defined in the generated .4gl file and must be used instead of the I4GL function name.

For example, in the I4GL zipcode demo, the Web service function name is cons_ws_zipcode, which must be renamed to zipcode_details as follows:

FUNCTION func_cons_ws_zipcode()
  DEFINE state_rec RECORD
    pin CHAR(10),
    city CHAR(100),
    state CHAR(100)
  END RECORD;
# Genero web service status returning
# whether web function call was successful or not
# DEFINE soapstatus INTEGER
#
# I4GL web service function name is 'cons_ws_zipcode'
# CALL cons_ws_zipcode("97006")
# RETURNING state_rec.city, state_rec.state
# Genero web service function name is 'zipcode_details'
CALL zipcode_details("97006")
  RETURNING soapstatus, state_rec.city, state_rec.state
...
END FUNCTION

Note: In Genero Web Services there is an additional returned parameter, soapstatus. If it contains 0 the operation was a success, otherwise an error occurred.

Step 5: Handle Genero Web Services errors
Describes how to code to check the error status returned by the I4GL Web service, and get details of the error.

I4GL web service errors are returned on a non-conventional SOAP fault that cannot be handled in Genero. However, the errors are handled through the additional returned parameter soapstatus that must be checked after each Web service call. If its value is not zero, an error has occurred and can be retrieved via the global Genero wsError record defined in the generated .inc file.

Example

In the Genero Web Service you must check the soap status after each Web service call:

FUNCTION func_cons_ws_zipcode()
  DEFINE state_rec RECORD
    pin CHAR(10),
    city CHAR(100),
    state CHAR(100)
  END RECORD;
  # Genero web service status returning
  # whether web function call was successful or not
  # DEFINE soapstatus INTEGER
  # Genero web service function call
  CALL zipcode_details("97006")
    RETURNING soapstatus, state_rec.city, state_rec.state
  # Check soap status for errors after zipcode_details call
  IF soapstatus<>0 THEN
    # Display error information from the server
    DISPLAY "Error:
    DISPLAY " code :",wsError.code
    DISPLAY " ns :",wsError.codeNS
    DISPLAY " desc :",wsError.description
    DISPLAY " actor:"wsError.action
  ELSE
    # Display results
    DISPLAY "\n ------------------------- \n" 
    DISPLAY "SUPPLIED ZIP CODE: 97006 \n"
    DISPLAY " ------------------------- \n"
    DISPLAY "RESPONSE FROM WEB SERVICE \n"
    DISPLAY " ------------------------- \n"
    DISPLAY " CITY:"state_rec.city
    DISPLAY " \n STATE:"state_rec.state
    DISPLAY " \n ------------------------- \n"
**Step 6: Compile and run the Genero client**
Describes how to compile and run the client to test it.

Then simply compile your modified I4GL application for Genero and execute it. Your application will then connect to the Web service; passing and returning the parameters as if they were only simple BDL function calls.

For example, to compile your I4GL web service application for Genero, you must execute the following commands:

```
$ fglcomp -M ws_zipcode_zipcode_detailsservice.4gl
$ fglcomp -M clsoademo.4gl
$ fgllink -o clsoademo.42r clsoademo.42m
   ws_zipcode_zipcode_detailsservice.42m
$ fglrun clsoademo.42r
```

**Standalone Axis server is buggy**
Describes a bug you can expect using the I4GL standalone axis server.

The I4GL standalone axis server adds an extra CR LF after the body of the SOAP HTTP post response what leads the Genero client to return the error message: `Body content bigger than expected`. This is not allowed as defined in HTTP [RFC2616].

**Important:** Axis works as expected if loaded from Apache server.

---

**REST Web services**

Genero Web Services allows you to write Web Services server and client applications for the REST Web standard architecture.

**Designing REST Web services**

Planning the resources that the client can interact with is essential to REST Web service design.

Each component in a RESTful Web service is a resource that can be accessed using standard HTTP methods.

The starting point in identifying resources is to analyze your business domain and extract the nouns (and most appropriately pluralized nouns) that are relevant to your business needs. For example, “customers” and “accounts” are resources in a typical business domain.

Once the nouns (resources) have been identified, then the interactions with REST web services can be modeled as HTTP verbs against these nouns. REST relies on this resource/HTTP method combination.

For example, imagine you have a Web service like the "StockQuote" service mentioned in the Introduction to Web services on page 3270. In this simplification of the business domain, we can begin to identify some types of resources:

- quotes
  - /id
  - /price
  - /category
  - etc.
- companies
  - /id
  - /name
  - /rating
  - etc.
- users
Here's how you might begin to build resource identifiers. For example, "users" is a collection resource and "user" is a single resource in this domain. You can identify "users" using the URI with path "/users" to locate the resource. You can identify a single user resource using the path "/users/{id}". The path or endpoint is a logical not a physical URL that determines the resource you are requesting.

**Note:** You may find it helpful also to think of the resource in terms of CRUD, where each resource supports the action of Create, Read, Update, and Delete using the HTTP verbs PUT, GET, POST, DELETE.

For instance, the following may now represent endpoints to the StockQuote service for the users entity.

<table>
<thead>
<tr>
<th>Users</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET <a href="http://localhost:8090/StockQuote/users">http://localhost:8090/StockQuote/users</a>       - Return a list of all users (requires authentication)</td>
</tr>
<tr>
<td>GET <a href="http://localhost:8090/StockQuote/users/id">http://localhost:8090/StockQuote/users/id</a>    - Return the user with that id</td>
</tr>
<tr>
<td>POST <a href="http://localhost:8090/StockQuote/users">http://localhost:8090/StockQuote/users</a>      - Create a new user. Return a 201 status code and the newly created id</td>
</tr>
<tr>
<td>PUT   <a href="http://localhost:8090/StockQuote/users/id">http://localhost:8090/StockQuote/users/id</a>   - Update the user with that id</td>
</tr>
<tr>
<td>DELETE <a href="http://localhost:8090/StockQuote/users/id">http://localhost:8090/StockQuote/users/id</a> - Delete the user with that id</td>
</tr>
</tbody>
</table>

Functions in the REST Web service now need to be created to respond to requests for services based on those endpoints. For this it is recommended to always use the GWS REST high-level framework in your Web service. Functions require considerably less code in your application and are easier to use in comparison with low-level APIs.

**Related concepts**

**REST** on page 3274
Represenational State Transfer (REST) is a Web standard architecture that provides a method for communication between a Web service and a client over HTTP.

**Planning a Web service** on page 3271
Creating a Web service application requires planning for the future use and reuse of the service.

**RESTful Web services with high-level framework**
These topics give you the information you need to begin working with RESTful Web services applications using BDL function with support for attributes.

**Steps to writing a REST server application**
To create a RESTful Genero Web server application, you need to create a Genero BDL module that defines the service functions. When you publish it, a service description is available when required.

**Getting started with REST high-level**
Prepare an environment for testing, create the server application, and run and test the service.

If you are on a local server, you can start working with the GWS as follows:

1. Set the runtime environment. For example, run the script file envcomp in the Genero BDL installation directory to make sure that your FGLDIR and PATH are set correctly to run compiler and runtime system tools.
2. Set FGLAPPSERVER to 8090. This will start the GWS Server on port 8090. If this environment variable is not set, port number 80 is used.

**Warning:** The user must not set the FGLAPPSERVER variable in production environments, since the port number is selected by the Genero Application Server.
3. Set FGLWSDEBUG environment variable to 3. This ensures there is detail in the output of the HTTP requests and responses to see the interaction between the client and the server and for debugging purposes. See FGLWSDEBUG on page 283.

4. Create your application module. For example, create a file named `myService.4gl`. Add service functions using the GWS REST function attributes. See the examples in the Define functions in a module on page 3376 section.

5. Create the application’s main service module. See Publish a REST service on page 3404

6. Compile your main application and service module.

7. Publish the service, see Publish a REST service on page 3404.

8. To test, use the `fglrun` command to execute the GWS Server application in direct mode. The service and the runtime must reside on the same machine:

   ```
   fglrun <gws application>
   ```

   This will start the GWS Server on the port specified by the FGLAPPSERVER environment variable.

9. If the GWS Server in step 2 was started on your local machine, for example, the OpenAPI information would be at:

   ```
   http://localhost:8090/<service-name>?openapi.json
   ```

   or

   ```
   http://localhost:8090/<service-name>?openapi.yaml
   ```

**Define functions in a module**

A GWS REST service is defined in a single module.

You write a BDL function for each service function you wish to make public. In your functions input and output parameters you can set attributes. These attributes allow you to specify the communication between the server and the client in terms of HTTP:

- Path, query, header, cookie
- Request body
- Response body
- Multipart request and response

**Function REST parameters**

Through function parameters and their attributes, you define how the resources of your Web service are to be accessed and used:

- REST operation attributes define the REST behavior (such as the HTTP GET, or POST operation).
- Path templating parameters (with `WSParam`, `WSPath` attributes) define the location of a resource. The path parameter value is directly related to a resource and is part of the resources URL path.
- REST attributes on the input parameters and output return values that define how parameters are serialized. For example, parameter attributes specify if data is carried in the body or header, as a cookie, as a query, or as multipart.

For details of all available attributes and descriptions of their usage, see the High-level RESTful Web service attributes on page 3463 section.

See the examples for defining your REST Web service functions.

**Related concepts**

REST Web services on page 3374

Genero Web Services allows you to write Web Services server and client applications for the REST Web standard architecture.

**Related reference**

High-level RESTful Web service attributes on page 3463
**Define your resource operations**

Operations are the HTTP methods used to manipulate the resources of your Web service.

For each resource path, you define an operation (HTTP method) that can be used to access the resource. An HTTP operation such as GET, POST, PUT, or DELETE, must be represented in your REST function in order to provide its service.

The same resource path can support several operations, for example, GET /users to get a list of users and POST /users to add a new user, etc. However, only one operation is allowed per function and therefore you must provide separate functions for these operations.

In the OpenAPI specification the combination of resource path and an HTTP method defines a unique function. Two GET or two POST functions for the same path are not allowed; even if they have different parameters. Input parameters or return values do not effect the uniqueness of the function.

**Define functions with operation attributes**

In GWS REST the attributes that perform operations are identifiable by the HTTP verb in the name, for example, **WSGet** on page 3467, **WSDelete** on page 3466, **WSPost** on page 3467, and **WSPut** on page 3468, etc. Your REST function must have one of these attributes in order to define its service. You can use these attributes in the attributes clause of your function.

**Publishing operations**

When you generate the service description, the operations you declared are published in a node. For example, there is a "get" node if it describes a GET operation. You find the operation node under the path for that resource in your OpenAPI specification file.

**Related concepts**

- **Designing REST Web services** on page 3374
Planning the resources that the client can interact with is essential to REST Web service design.

- **Generate service description on demand** on page 3406
You can access a description of your GWS REST Web service through a query string on the URL of the Web service.

Example: Get operation with WSGet

Example of methods you can use to get data from a resource with the WSGet attribute.

**Example using WSGet**

In this sample REST function the number of users from the users table of a database is returned.

```plaintext
PUBLIC FUNCTION getNumberUsers()
    ATTRIBUTES (WSGet,
                  WSPath="/users/count",
                  WSDescription="Returns a count of users")
    RETURNS INTEGER
    DEFINE cnt INTEGER
    SELECT COUNT(*) INTO cnt FROM users
    RETURN cnt
END FUNCTION
```

**Example using WSGet to return all users**

In this function all users are returned. An example of the resource URL is:

```plaintext
http://host:port/gas/ws/r/group/xcf/Account/users
```

```plaintext
IMPORT com

TYPE profileType RECORD ... END RECORD
```
PUBLIC FUNCTION getAllUsers()
    ATTRIBUTES (WSGet,
        WSPath="/users",
        WSDescription="Returns all user profiles",
        WSThrows="400:Invalid,404:Not found")
    RETURNS (DYNAMIC ARRAY ATTRIBUTE (WSName="All_users", WSMedia="application/xml") OF profileType
        ATTRIBUTE (XMLName="User") )

    DEFINE arr DYNAMIC ARRAY OF profileType
        # ... function code ...

    RETURN arr
END FUNCTION

The output as XML is customized by WSName on page 3475 and XMLName on page 3548

Related concepts
Resource operation attributes on page 3465
Attributes that map an HTTP operation in a function to a REST resource.

Example: create resource with WSPost
Create a new resource with the WSPost attribute.

Example creating resource with WSPost
In this sample REST function a new user resource is created. The user record is defined as a data type in the module. This is referenced in the input parameter thisUser.

IMPORT com

TYPE profileType RECORD ... END RECORD

PUBLIC DEFINE userError RECORD ATTRIBUTE (WSError="User error")
    message STRING
END RECORD

PUBLIC FUNCTION createUser( thisUser profileType )
    ATTRIBUTES (WSPost,
        WSPath="/users",
        WSDescription="Create a user profile",
        WSThrows="400:@userError")
    RETURNS STRING

    DEFINE ret STRING
    WHENEVER ERROR CONTINUE
        INSERT INTO users VALUES (thisUser.*)
        WHENEVER ERROR STOP
        CASE 
            WHEN SQLCA.SQLCODE == 0
                LET ret = SFMT("Created user: %1",thisUser.name)
            OTHERWISE 
                LET userError.message = SFMT("SQL error:%1 [%2]",SQLCA.SQLCODE, SQLERRMESSAGE)
                CALL com.WebServiceEngine.SetRestError(400,userError)
        END CASE
    END CASE

    RETURN ret
END FUNCTION

Related concepts
Resource operation attributes on page 3465
Attributes that map an HTTP operation in a function to a REST resource.

**Set a request body** on page 3383
Functions that create or update a resource need to set a request body for the incoming payload. You specify the request body in an input parameter.

**Example: Update resource with WSPut**
Update a resource with the WSPut attribute.

**Example updating a resource with WSPut**
In this sample REST function a user resource is updated.

```sql
IMPORT com

TYPE profileType RECORD ... END RECORD

PUBLIC DEFINE userError RECORD ATTRIBUTE(WSError="User error")
  message STRING
END RECORD

PUBLIC FUNCTION updateUsers(
  id INTEGER ATTRIBUTES(WSParam),
  thisUser profileType)
  ATTRIBUTES(
    WSPut,
    WSPATH="/users/{id}",
    WSDESCRIPTION="Update a user profile",
    WSThrows="400:@userError"
  )
  RETURNS STRING
  DEFINE ret STRING
  WHENEVER ERROR CONTINUE
  UPDATE users
    SET name = thisUser.name,
    email = thisUser.email
  WHERE @id = id
  WHENEVER ERROR STOP
  CASE
  WHEN SQLCA.SQLCODE = 0
    # test for processed rows
    IF SQLCA.SQLERRD[3] = 1 THEN
      LET ret = SFMT("Updated user with ID: %1",id)
    ELSE
      LET ret = SFMT("No user with ID: %1",id)
    END IF
  OTHERWISE
    LET ret=SFMT("Error updating user with ID: %1",id)
    LET userError.message = SFMT("SQL error:%1 [%2]",SQLCA.SQLCODE,
    SQLERRMESSAGE)
    CALL com.WebServiceEngine.SetRestError(400,userError)
  END CASE
  RETURN ret
END FUNCTION
```

**Related concepts**

**Resource operation attributes** on page 3465
Attributes that map an HTTP operation in a function to a REST resource.

**Example: Delete a resource with WSDelete**
Delete a resource with the WSDelete attribute.

**Example deleting a resource with WSDelete**

In this sample REST function a user resource is deleted.

```gws
PUBLIC FUNCTION deleteUsers( id INTEGER ATTRIBUTES(WSParam) )
    ATTRIBUTES(WSDelete,
        WSPATH="/users/{id}",
        WSDescription="Delete a user profile")
    RETURNS STRING
    DEFINE ret STRING
    DELETE FROM users WHERE @id = id
    LET ret = SFMT("Deleted user with ID: %1",id)
    RETURN ret
END FUNCTION
```

**Related concepts**

- [Resource operation attributes](#) on page 3465
  - Attributes that map an HTTP operation in a function to a REST resource.

- **Set resource path with WSParam and WSPATH**
  - Path parameters allow you to specify variables in the resource URL.

  Path parameters are variable parts of a resource path. You would use them if you needed in your function to point to a specific resource within a collection, such as a user identified by ID, and when a pattern could match many similar resources. This is an example of the format of a resource path with path template.

  `/users/{id}

  In the OpenAPI specification, the parts of the path that are replaceable with parameters are path templates. They are denoted with curly brackets `{}`. The path template is substituted with actual values when a GWS client makes a call to the resource.

  For instance, if a client requests data for a specific user, the user id (22) is provided in the URI:

  ```text
  http://host:port/gas/ws/r/group/xcf/Account/users/22
  ```

  A function in your web service with the path pattern above processes the request.

  In GWS REST the **WSPath** on page 3469 and **WSParam** on page 3469 attributes support path templating. Path parameters are set in the attributes clause of the function with the `WSPATH` attribute. For example:

  ```gws
  ATTRIBUTES (WSGet,WSPATH="/members/{membersId}/books/{bookId}")
  ```

  A call to the function with this path checks if a specified member checked out a book from the library. The input parameters that correspond to the templates in the path are described in **Table 718: Path parameters** on page 3380.

**Table 718: Path parameters**

<table>
<thead>
<tr>
<th>Parameter with WSParam attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>membersId INTEGER ATTRIBUTE(WSParam)</td>
<td>A unique identifier for a library user resource.</td>
</tr>
<tr>
<td>booksId INTEGER ATTRIBUTE(WSParam)</td>
<td>A unique identifier for a book resource.</td>
</tr>
</tbody>
</table>

Be aware that for each BDL function input parameter with a `WSParam` attribute, there must be a matching template value in the `WSPATH` attribute value. `fglcomp` checks to ensure that there is one template value per parameter, otherwise compilation `error-9111` is thrown.
Example path templating with WSPath and WSParam

```plaintext
IMPORT com

TYPE checkoutType RECORD
  booksId INT,
  title VARCHAR(100),
  author VARCHAR(100),
  m_name VARCHAR(100),
  checkout_date DATETIME YEAR TO SECOND
END RECORD

DEFINE myError RECORD ATTRIBUTE(WSError="My error")
  code INTEGER,
  reason STRING
END RECORD

PUBLIC FUNCTION getBooksCheckedOut(
  p_membersid INTEGER ATTRIBUTE(WSParam) )
  ATTRIBUTES (WSGet,
    WSPath="members/{p_membersid}/books/",
    WSDescription="Get books checked out by a given member of the
    library",
    WSThrows="400:Invalid,406:Should not happen" )
  RETURNS (  
    DYNAMIC ARRAY ATTRIBUTE(WSName="Books_checked_Out",WSMedia="application/xml") OF checkoutType
    ATTRIBUTE(XMLName="Book") )
  DEFINE bookList DYNAMIC ARRAY OF checkoutType
  INITIALIZE bookList TO NULL
  TRY
    DECLARE c1 CURSOR FOR SELECT b.booksid, b.title, b.author, m.name, 
    c.checkoutdate FROM books b 
    INNER JOIN checkouts c ON b.booksid = c.booksid AND c.membersid = m.membersid 
    INNER JOIN members m ON c.membersid = p_membersid 
    IF SQLCA.SQLCODE = 0 THEN 
      # ... function code
    END if
    CATCH 
    WHEN SQLCA.SQLCODE = NOTFOUND 
      LET myError.reason = SFMT("Nothing found for member: %1", p_membersid )
      CALL com.WebServiceEngine.SetRestError(400,myError) 
    OTHERWISE 
      LET myError.reason = SFMT("Error in SQL execution: %1 [%2]", SQLCA.SQLCODE, SQLERRMESSAGE )
      CALL com.WebServiceEngine.SetRestError(406,myError)
    END CASE
  END TRY
  FREE c1
  RETURN bookList
END FUNCTION
```

The variable part of the path is replaced by the value /48/ in this URL example:

```
http://host:port/gas/ws/r/group/xcf/MyLibrary/members/48/books
```

The XML output is customized by WName on page 3475 and XMLName on page 3548.
When to use WSPath or WSQuery

It is important to know when it is appropriate to use these attributes in your Web service. Basically the difference is that a path parameter is used to return a single entity given its id field, while a query parameter is used for any other field other than a resource id and a number of entities may be returned in the response.

For example, /users/{id} specifies the entity in the resource path. Usually you get a HTTP response code 404 (Not found) if an entity is not found by the id specified as path parameter.

On the other hand if you think in the case of a search or filter, you expect to return a number of entities in the response, unlike the single entity that you expect to get as the response for sending id as path parameter. Therefore, WSQuery on page 3471 is the logical attribute to use for filtering, searching, or sorting resources:

For example, in /users?lastname=smith the search criteria is specified in a query parameter of the URL.

Usually you get an HTTP response code 200 (OK) with an empty array as the response body, if no entities are found matching the search or filter parameters.

Related concepts

Designing REST Web services on page 3374
Planning the resources that the client can interact with is essential to REST Web service design.

Set query, header, or cookie parameters (if needed)

You define these parameters in your function if the resource needs data to be passed as query, cookie, or header.

The GWS REST engine provides support for the OpenAPI specification requirements for the following standard parameters:

1. Query parameters. These are set at the end of the resource URL after a question mark (?), with different name=value pairs separated by ampersands (&). Query parameters can be required and optional.
2. Header parameters, such as X-MyHeader:Value. These are custom headers to be sent with an HTTP request or response.
3. Cookie parameters, are passed in the Cookie header. For example, Cookie:ctoken=BUSE35dohU4O1MZxDCU.

Parameter attributes

You use the following attributes if there is requirement for them in your Web service function.

1. WSQuery on page 3471. You can specify this attribute only in an input parameter of your function. Its use is suited to filtering on resource collections, such as /users?country=france.
2. WSHeader on page 3472. Header attributes may be specified in an input parameter or in a return value of your function. The Content-Type:text/html header, for example, indicates the media type of the response sent to the client by the server. This helps the client in processing the response body correctly. Custom headers are commonly used for informational purposes, but you can also use them for passing data to implement logic on the server or client side.
3. WSCookie on page 3473. The cookie attribute is dedicated to the need for cookies in your function.

You can set these parameters as optional with the WSOptional on page 3476 attribute as shown in the sample function.

Example use of parameter attributes

```plaintext
TYPE profileType RECORD ...
END RECORD

PUBLIC FUNCTION getUsersStatus(
    stat STRING ATTRIBUTE(WSQuery, WSOptional, WSName="status"),
    sort_field STRING ATTRIBUTE(WSHeader, WSOptional, WSName="sortby"),
    country STRING ATTRIBUTE(WSCookie, WSOptional, WSName = "ccode" )
) ATTRIBUTES (WSGet,
    WSPath = "/users",
```
Your client application has the option of providing the custom header and cookie parameter with values in a call to the function.

The `WSQuery` parameter, if specified, forms the query string of the resource path, `status=active` in this example of the resource URL:

```
http://host:port/gas/ws/r/group/xcf/Accounts/users?status=active
```

The XML output is customized by `WSName` on page 3475 and `XMLName` on page 3548 attributes.

**Related concepts**

- **Designing REST Web services** on page 3374
- Planning the resources that the client can interact with is essential to REST Web service design.

- **Retrieve HTTP headers** on page 3385
- You can retrieve HTTP headers in your REST operation.

**Set a request body**

Functions that create or update a resource need to set a request body for the incoming payload. You specify the request body in an input parameter.

**Note:**

A message body is required when you perform an HTTP PUT, POST or PATCH request to a resource, otherwise the request results in the `error-9106`.

In your BDL function, you specify a request body by declaring an input parameter that does not have any `WSHeader`, `WSQuery`, `WSCookie`, or `WSParam` attributes.

In the example the single input parameter ("lname") is defined as a string. It is not defined with a REST attribute and its content is therefore sent in the body of the HTTP request.

**Note:** If you have more than one of such input parameters, it means your payload is delivered in a multipart body request.

If the data type is not a primitive data type, you may need to define a type suitable for the structure in your module. You reference the type in your input parameter. For an example, see Example: create resource with `WSPost` on page 3378. The function needs to receive data of the required structure for the resource in either a JSON or XML representation.

**Example specifying a request body**

```plaintext
IMPORT com
PUBLIC
DEFINE myerror RECORD ATTRIBUTE(WSError="My error")
    code INTEGER,
    reason STRING
END RECORD
PUBLIC FUNCTION updateUsersField(
```
id INTEGER ATTRIBUTES(WSParam, WSDescription="User id")
, lname STRING ATTRIBUTE(WSDescription="User's last name")

ATTRIBUTES(WSPut,
  WSPATH="/users/{id}",
  WSDescription="Update name of a given user",
  WSThrows="400:@myerror")

RETURNS STRING

DEFINE ret STRING
WHENEVER ERROR CONTINUE
UPDATE users SET
  user_name=lname
WHERE @user_id = id
WHENEVER ERROR STOP
CASE
  WHEN SQLCA.SQLCODE == 0
    LET ret = SFMT("Updated user: %1", lname)
  WHEN SQLCA.SQLCODE == NOTFOUND
    LET ret = SFMT("No row was found with ID: %1", id)
  OTHERWISE
    LET myerror.reason = SFMT("SQL error:%1 [%2]", SQLCA.SQLCODE, SQLERRMESSAGE)
    CALL com.WebServiceEngine.SetRestError(400, myerror)
  END CASE
RETURN ret
END FUNCTION

| PUT /courseGWS/users/1 HTTP/1.1 |
| Accept: text/plain, application/json, application/xml, text/xml |
| Content-Type: text/plain |
| User-Agent: GWS Agent (Build 1560416742) |
| Host: localhost:8090 |
| Connection: close |
| Date: Mon, 24 Jun 2019 15:52:22 GMT |
| Content-Length: 11 |
| change this |

**Figure 126: Output of the HTTP request**

**Related concepts**

HTTP verbs and attributes on page 3465
HTTP verbs are defined by the high-level RESTful attributes. Some verbs have requirements for request or response body and others do not.

**Set a response body and header**

Functions that create, update, delete, or get a resource need to set a response body. You specify this in a return parameter without an attribute. Other return values can be sent in headers, using the WSHdrer attribute.

**Note:**
A message body in the response is required when you perform an HTTP GET, POST, PUT, DELETE operation on a resource, otherwise the response results in the error-9106.

**Example responses in header and body**

PUBLIC FUNCTION help()

ATTRIBUTES (WSGet,
  WSPATH="/help")

RETURNS (INTEGER ATTRIBUTE(WSHeader, WSDescription="Reference number"), STRING)
The help function's RETURNS clause has two return values:

- An integer is returned in a header. It is specified with the WSHdr attribute.
- A string is returned in the body. It is specified without an attribute.

**Figure 127: Output of the HTTP response**

![HTTP response output](image)

In the output the header is given a default name, "rv0", at runtime. You can change default header naming via the WSName attribute, for example with:

```
RETURNS (INTEGER ATTRIBUTE(WSHeader, WSName="MyHeader"), STRING)
```

**Related concepts**

- Multipart requests or responses on page 3394

In GWS REST there is support for the standard multiple part message, in which more than one different sets of data are combined in a single body.

**Retrieve HTTP headers**

You can retrieve HTTP headers in your REST operation.

There are different ways for retrieving HTTP headers; such as using the WSHdr attribute as part of a function parameter, or retrieving headers via WSContext on page 3481.

The option you choose depends on your requirements:

- If you need to retrieve the header as part of the REST operation, you use WSHdr on page 3472 and WSName on page 3475 attributes in your function parameter. You can retrieve any HTTP header in this way. In case the header is not present, you must also set the WSOptl attribute on page 3476 attribute. For example, the remote address can be passed in the header by coding:

  ```
  ip_address STRING ATTRIBUTES(WSHeader, WSOptl, WSName="X-FourJs-Environment-Variable-REMOTE_ADDR")
  ```

- If you do not want the header to be part of the REST operation, you can use the WSContext on page 3481 mechanism. For example, the remote address and server name can be retrieved by setting a context dictionary variable (for example `context`) at the modular level with the WSContext attribute. This allows you to retrieve all X-FourJs-Environment-xxx set by the GAS by referencing a dictionary key value for the environment variable:

  ```
  PRIVATE DEFINE context DICTIONARY ATTRIBUTE(WSContext) OF STRING
  ```
Web services

DISPLAY context["Variable-REMOTE_ADDR"] -- displays the remote ADDR if set by the GAS
DISPLAY context["Variable-SERVER_NAME"] -- displays the server name if set by the GAS

Related concepts
Set query, header, or cookie parameters (if needed) on page 3382
You define these parameters in your function if the resource needs data to be passed as query, cookie, or header.

Handling file attachments with REST
The Genero REST high-level framework provides two mechanisms for handling attachments.

• Send (or receive) the complete file as a simple attachment.
• Send (or receive) a file in a MIME multipart message body.

Which method to use

<table>
<thead>
<tr>
<th>Situation</th>
<th>Method to choose</th>
</tr>
</thead>
<tbody>
<tr>
<td>If your REST service needs one file, whether it is a zip file, image, or PDF.</td>
<td>Send the complete file as a simple attachment. See Attach files with WSAttachment and WSMedia on page 3386.</td>
</tr>
<tr>
<td>If the design of your REST Web services requires multipart to attach files.</td>
<td>Send the file as multipart. See Multipart requests or responses on page 3394.</td>
</tr>
<tr>
<td>If you have several attachments to send in a single HTTP request or response message.</td>
<td>Send the file as multipart. See Multipart requests or responses on page 3394.</td>
</tr>
<tr>
<td>If you have a large data object, whether it is a binary, text file, or a BLOB in a database.</td>
<td>Send the data in a BYTE or TEXT object in the message body. See Transfer data in large objects on page 3390</td>
</tr>
</tbody>
</table>

Related concepts
Generate service description on demand on page 3406
You can access a description of your GWS REST Web service through a query string on the URL of the Web service.

Attach files with WSAttachment and WSMedia
In GWS REST attachments are handled via the WSAttachment and WSMedia attributes.

If a parameter has a WSAttachment on page 3477 attribute, the REST engine treats the parameter value as a path to a file to be attached, while the data format of the file can be specified in the WSMedia on page 3478 attribute.

Attaching files in request and response
In this function, an image file is sent to the server in the request and another image is returned to the client. The wildcard (image/*) in WSMedia allows for all image types:

```
IMPORT os
PUBLIC FUNCTION EchoFile( input STRING ATTRIBUTE (WSAttachment, WSMedia="image/*") )
    ATTRIBUTES(WSPost)
    RETURNS STRING ATTRIBUTE (WSAttachment, WSMedia="image/*")
    DEFINE ok INTEGER
    LET ok = os.path.rename(input, "MyFile.png")
    RETURN "/usr/local/MyOtherFile.jpg"
END FUNCTION
```
Related concepts

Multipart requests or responses on page 3394
In GWS REST there is support for the standard multiple part message, in which more than one different sets of data are combined in a single body.

Transfer data in large objects on page 3390
Use the BYTE and TEXT data types to transfer large objects (LOBs) with a Genero RESTful web service.

Generate service description on demand on page 3406
You can access a description of your GWS REST Web service through a query string on the URL of the Web service.

Download a file as an attachment to a response
This example demonstrates how to return a file as an attachment using the WSAttachment attribute

In order to return a file as an attachment to a client, set the WSAttachment attribute on page 3477 attribute on an output parameter.

**Example 1: downloading file using WSAttachment**

```plaintext
IMPORT com
IMPORT os

PUBLIC DEFINE myerror RECORD ATTRIBUTE(WSError="My error")
  code INTEGER,
  reason STRING
END RECORD

PUBLIC FUNCTION downloadImageFile()
  ATTRIBUTES (WSGet,
    WSPath="/files3/images",
    WSDescription="download image file to the client with WSAttachment",
    WSThrows="400:myerror")
  RETURNS (STRING ATTRIBUTE(WSAttachment, WSMedia="image/*") )
  DEFINE ret, fname STRING
  DEFINE ok INTEGER

  LET fname="favicon.ico"
  LET ok = os.Path.exists(fname)
  IF ok THEN
    LET ret = fname
  ELSE
    LET myerror.reason = SFMT("File (%1) does not exist", fname)
    CALL com.WebServiceEngine.SetRestError(400,myerror)
  END IF
  RETURN ret
END FUNCTION
```

In this example:

- A STRING type is defined with a WSAttachment attribute.
- A WSMedia attribute is added to handle the data format for images.
In Figure 128: Output of HTTP response to download image file on page 3388 the Content-Disposition response header in the output indicates that the content is expected as an attachment. Therefore, when the function is called, the file is downloaded and saved to the client locally in its TMP directory.

Related concepts

Multipart requests or responses on page 3394
In GWS REST there is support for the standard multiple part message, in which more than one different sets of data are combined in a single body.

Download a large object in the response body on page 3390
This example demonstrates how to send a large object in the message response body.

Example: upload a file as attachment in request body
Shows a sample function uploading an image file as attachment in the message request body.

In order to send a file as an attachment to the Web service, set the WSAttachment on page 3477 attribute on an input parameter.
Example 1: upload file as attachment using WSAttachment

```plaintext
IMPORT os

PUBLIC FUNCTION UploadFile( fname STRING ATTRIBUTES(WSAttachment,
    WSMedia="image/*") )
    ATTRIBUTES (WSPost,
        WSPath="/files2/fetch")
    RETURNS STRING ATTRIBUTE(WSMedia="text/plain")
    DEFINE ret, new_fname STRING
    DEFINE ok INTEGER
    LET new_fname = os.Path.pwd()||os.Path.separator()||new_fname
    LET ok = os.Path.rename(fname,new_fname)
    LET ret = SFMT("Got image: %1",new_fname)
    RETURN ret
END FUNCTION
```

In this example:
- "fname" is defined as type STRING with a WSAttachment attribute.
- The WSMedia attribute handles the data format for images.

The GWS stores the file in a temporary directory, and returns the absolute path to the file in the input parameter (fname). When the call to the function ends, the file will be removed unless you save it to a location on disk. You can use the `os.Path.rename` instruction to move the file from the temporary directory to a directory of your choice.

![Image file upload as attachment](image.png)

**Figure 129: Output of HTTP request to upload file**

In **Figure 129: Output of HTTP request to upload file** on page 3389 the image is sent as an attachment in the message body.

**Related concepts**

Upload a large object in the request body on page 3391
This example demonstrates how to upload a large object in the message response body.

Example: upload a file in a multipart request on page 3397
Shows an example of a method you might use from a client to upload an image along with other data in a function using a form-data type HTTP multipart request.

Transfer data in large objects

Use the BYTE and TEXT data types to transfer large objects (LOBs) with a Genero RESTful web service.

Specify an input or return parameter of a Web service function as a TEXT on page 307 or BYTE on page 290 type in order to transfer a large object (LOB) to a web service server or client.

**Which object to use**

<table>
<thead>
<tr>
<th>Situation</th>
<th>Data type to use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Your web service transfers binary file contents. For example: an image, PDF, zip file, etc.</td>
<td>BYTE</td>
</tr>
<tr>
<td>You have a binary large object (BLOB) stored in a table and your web service must write it out to a file. For example, an image.</td>
<td>BYTE</td>
</tr>
<tr>
<td>Your web service transfers ASCII content. For example: text, HTML, etc.</td>
<td>TEXT</td>
</tr>
</tbody>
</table>

This method transfers data in the request body.

**Important:** To send a LOB from a REST client function you must, LOCATE it in memory or file to load the data before calling the server function. On the server REST function, you do not need to locate the input parameters TEXT/BYTE types as they are implicitly located in memory when the server function is called. See code samples in [Upload a large object in the request body](#) on page 3391.

**Related concepts**

Attach files with WSAattachment and WSMedia on page 3386

In GWS REST attachments are handled via the WSAattachment and WSMedia attributes.

Multipart requests or responses on page 3394

In GWS REST there is support for the standard multiple part message, in which more than one different sets of data are combined in a single body.

Download a large object in the response body

This example demonstrates how to send a large object in the message response body.

To transfer a large object (LOB) to a client, specify a return parameter using a TEXT on page 307 or BYTE on page 290 data type. The large object is returned in the response body.

**Example: downloading a TEXT object**

```public function help3()
  attributes (wsget,
    wsdescription="download text to the client in a TEXT object",
    wspath="/help/file3")
  returns ( INTEGER ATTRIBUTE(WShereader), TEXT )
  define t TEXT
  locate t in memory
  call t.readFile("file3.txt")
  return 3, t
end function```

In this example:
- The TEXT variable (t) is created to hold the file content and is located in memory.
- The readFile() method reads the content of the text file (file3.txt) into the TEXT variable (t).
- The TEXT variable (t) is transferred in the message body.
- The INTEGER variable, defined as a return parameter with the WSHeader attribute, is transferred in the response header.

```
GET /Myservice/help/file3 HTTP/1.1
Host: localhost:8090
Content-Type: text/plain; charset=UTF-8
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
HTTP/1.1 200 Success
rv0: 3
```  

**Figure 130: Output of HTTP response**

The header is given a default name ("rv0") at runtime. Use the WSName on page 3475 attribute to rename the header.

```
RETURNS (INTEGER ATTRIBUTE(WSHeader, WSName="MyHeader"), TEXT)
```

**Related concepts**
- Download a file in a multipart response on page 3396
  This example demonstrates how to return a file in a form-data type HTTP multipart response while also transferring data of other types in the same message response.
- Download a file as an attachment to a response on page 3387
  This example demonstrates how to return a file as an attachment using the WSAttachment attribute

Upload a large object in the request body
This example demonstrates how to upload a large object in the message response body.

To upload a large object (LOB) to a web service server, specify an input parameter using a TEXT on page 307 or BYTE on page 290 data type. The large object is sent in the request body. Any binary file contents (image, PDF, zip, etc.) may be uploaded using this method.

**Example: Uploading a BYTE object and writing to a file**

```
IMPORT os

PUBLIC FUNCTION UploadByteToFile(b BYTE)
```
ATTRIBUTES (WSPost,  
WSDescription="Upload BYTE and write to file on server",  
WSPath="/help/pdf")  
RETURNS STRING 
CONSTANT filename = "help.pdf" 
DEFINE ret STRING  
TRY  
CALL b.writeFile(filename)  
LET ret = SFMT("Data received, saved to file: %1",filename)  
CATCH  
LET ret = SFMT(" Error getting data, not saved to file: %1",filename)  
END TRY  
RETURN ret  
END FUNCTION

In this example:

- The large object is transferred in the BYTE input parameter b in the message body.
- The writeFile() method writes the content of the variable b to a file (help.pdf) on the server side.

**Example: Client app calling the function**

The client needs to load a BYTE object before calling the function.

```
IMPORT util
IMPORT FGL clientGWSStub

MAIN
  DEFINE wsstatus INTEGER
  DEFINE retS STRING

  DEFINE b BYTE
  LOCATE b IN MEMORY
  CALL b.readFile("help.pdf")

  CALL clientGWSStub.UploadByteToFile(b) RETURNING wsstatus, retS
  CASE wsstatus
    WHEN clientGWSStub.C_SUCCESS
      DISPLAY SFMT("Got %1 ", retS)
    OTHERWISE
      DISPLAY "wsstatus is : ", wsstatus
      DISPLAY "Error in getting file uploaded"
  END CASE

END MAIN
```

In this example:

- A BYTE variable b is created to hold the large object and the LOCATE instruction specifies where it is stored (MEMORY or FILE).
- The readFile() method reads the contents of a PDF file (help.pdf) into the BYTE variable b.
- A call to the server function is made specifying b as the input parameter. The data is transferred in the request body.

**Related concepts**

- **Example: upload a file in a multipart request** on page 3397
  Shows an example of a method you might use from a client to upload an image along with other data in a function using a form-data type HTTP multipart request.

- **Example: upload a file as attachment in request body** on page 3388
Shows a sample function uploading an image file as attachment in the message request body.

**Set data format with WSMedia**

It is important to set the correct MIME type for a Web service request or response. You can specify the data format via the WSMedia attribute.

For records and arrays in the message response the default MIME type can be JSON (**application/json**) or XML (**application/xml**). **WSMedia** on page 3478 can contain a comma-separated list of MIME types. The payload format is chosen according to that specified in the Accept header received from the client using the Web service.

In the example the **WSMedia** attribute is set for XML to send the list of users to the client in XML format. The output is customized by **WSName** on page 3475 and **XMLName** on page 3548 attributes.

**Example WSMedia with record type**

```plaintext
IMPORT com

TYPE profileType RECORD ...

PUBLIC DEFINE myError RECORD ATTRIBUTE(WSError="My error")
  code INTEGER,
  reason STRING
END RECORD

PUBLIC FUNCTION getUsersList()
  ATTRIBUTES (WSGet,
    WSPath="/users",
    WSDescription="Get list of users",
    WSThrows="400:Invalid,404:NotAvailable"
  )
  RETURNS (DYNAMIC ARRAY ATTRIBUTE(WSName="All_users_list",WSMedia="application/xml") OF profileType
    ATTRIBUTE(XMLName="User")
  )
  DEFINE usersList DYNAMIC ARRAY OF profileType
  DEFINE i INTEGER
  WHENEVER ERROR CONTINUE
  DECLARE c1 CURSOR FOR SELECT * FROM users
  WHENEVER ERROR STOP
  CASE
    WHEN SQLCA.SQLCODE == 0
      INITIALIZE usersList TO NULL
      LET i=1
      FOREACH c1 INTO usersList[i].*
        LET i=i+1
      END FOREACH
      CALL usersList.deleteElement(i)
    WHEN SQLCA.SQLCODE == NOTFOUND
      LET myError.reason = "Nothing found"
      CALL com.WebServiceEngine.SetRestError(404,myError)
    OTHERWISE
      LET myError.reason = SFMT("SQL error:%1 [ %2],SQLCA.SQLCODE, SQLERRMESSAGE)
      CALL com.WebServiceEngine.SetRestError(400,myError)
  END CASE
  FREE c1
  RETURN usersList
END FUNCTION
```
Example WSMedia with image

For WSMedia examples using images or files, see Example: upload a file in a multipart request on page 3397.

Customize XML serialization with WSName and XMLName

Use the WSName and XMLName attributes to customize serialization at runtime and improve the readability of the XML output.

Formatting XML output with WSName and XMLName

```
IMPORT com

TYPE profileType RECORD ... END RECORD

PUBLIC DEFINE myError RECORD ATTRIBUTE(WSError="My error")
    code INTEGER,
    reason STRING
END RECORD

PUBLIC FUNCTION getUsersList()
    ATTRIBUTES (WSGet,
        WSPath="/users",
        WSDescription="Get list of users",
        WSThrows="400:Invalid,404:NotAvailable"
    )
    RETURNS (DYNAMIC ARRAY ATTRIBUTE(WSName="All_users_list",WSMedia="application/xml")
        OF profileType
    )
    DEFINE usersList DYNAMIC ARRAY OF profileType
    ... function code
    RETURN usersList
END FUNCTION
```

The function returns the output in XML format as a string. The output contains the tags specified by the attributes.

- The **WSName** attribute for the array gives `<All_users_list>` as the root tag for the output.
- The **XMLName** attribute (XMLName="User") maps the BDL record (profileType) and tags each element with `<User>` in the XML document.

For example, when viewed in an XML viewer, it would display as shown.

```
<All_users_list>
    <User>
        ... record content
    </User>
    <User>
        ... record content
    </User>
    ...
</All_users_list>
```

**Note:** If you do not use the attributes, the names in the XML output are default names derived from DVM names, for example,"rv0", which may not be user-friendly.

**Multipart requests or responses**

In GWS REST there is support for the standard multiple part message, in which more than one different sets of data are combined in a single body.

You can specify an HTTP multipart request or response for transferring data of several MIME types, such as JSON, XML, simple string, and to upload or download files.
**Multipart/form-data support**

In the multipart/form-data type, entire files can be included in the data transfer without encoding. This is ideally suited for uploading and downloading image or text files to a Web service.

Separate parts are identified by the GWS REST engine's naming convention sequence for headers and body parts that start with "rv0" and goes to the "rvn"th number of parts. You can change default header naming by adding a value in the WSNName on page 3475 attribute to your parameter.

The GWS REST engine also creates a "boundary" string in the "Content-Type: " header. This boundary, is placed between the various parts, and at the beginning and end of the body of the message.

**Define multipart request body**

You define a multipart request body when you have more than one input parameters that do not have WSHeader, WSQuery, WSCookie, or WSParam attributes. The GWS handles this request as a multipart of form-data type.

**Important:**

A message body is required when you perform an HTTP PUT, POST or PATCH request to a resource, otherwise the request results in the error-9106.

**Example multipart request**

The example is of a multipart request function. The input parameter values "addressfld1" and "date2" are sent in separate parts in the request body:

```java
IMPORT com
IMPORT util

PUBLIC DEFINE myError RECORD ATTRIBUTE(WSError="My error")
  code INTEGER,
  reason STRING
END RECORD

PUBLIC FUNCTION updateUsersAddress(
  id INTEGER ATTRIBUTES(WSParam),
  addressfld1 STRING,
  date2 STRING)
ATTRIBUTES(WSPut,
  WSPut="/users/{id}",
  WSPutQuery="Update user address and date updated field",
  WSThrows="400:@myerror")
RETURNS STRING

DEFINE ret STRING
DEFINE dt DATETIME YEAR TO SECOND
LET dt = util.Datetime.parse(date2, "%Y-%m-%d %H:%M:%S")
IF (dt IS NULL) THEN
  LET ret = "Invalid date"
ELSE
  WHENEVER ERROR CONTINUE
  UPDATE course2 SET users_address= addressfld1, date_updated=dt
  WHERE @users_id = id
  WHENEVER ERROR STOP
  CASE
  WHEN SQLCA.SQLCODE == 0
    LET ret = SFMT("Updated user with ID: %1",id)
  WHEN SQLCA.SQLCODE == NOTFOUND
    LET ret = SFMT("No resource found for ID: %1",id)
  OTHERWISE
    LET myError.reason = SFMT("SQL error:%1 [%2]",SQLCA.SQLCODE, SQLERRMESSAGE)
    CALL com.WebServiceEngine.SetRestError(400,myError)
  END CASE
```

```java
END CASE
```
Define multipart response body

You define a multipart response body if you have more than one return values without WSHeader attributes. The GWS handles this as a multipart response of type form-data.

Important:

A message body in the response is required when you perform an HTTP GET, POST, PUT, DELETE operation on a resource, otherwise the response results in the error-9106.

Example multipart response

The example is a multipart response sample REST function. There are two string values defined in the RETURNS clause of the function. These do not have WSHeader attributes, so values are sent in separate parts in the response body:

PUBLIC FUNCTION help()
  ATTRIBUTES (WSGet,
    WSPath="/help")
  RETURNS (INTEGER ATTRIBUTE(WSHeader), STRING, STRING)
  RETURN 3, "Hello world", "Have a nice day."
END FUNCTION

Related concepts

Generate service description on demand on page 3406
You can access a description of your GWS REST Web service through a query string on the URL of the Web service.

Transfer data in large objects on page 3390
Use the BYTE and TEXT data types to transfer large objects (LOBs) with a Genero RESTful web service.

Attach files with WSAttachment and WSMedia on page 3386
In GWS REST attachments are handled via the WSAttachment and WSMedia attributes.

Download a file in a multipart response

This example demonstrates how to return a file in a form-data type HTTP multipart response while also transferring data of other types in the same message response.

Example multipart response

PUBLIC FUNCTION help2()
  ATTRIBUTES (WSGet,
    WSPath="/help2")
  RETURNS (INTEGER ATTRIBUTE(WSHeader),
    STRING,
    TEXT ATTRIBUTE(WSName="txt_file") )
  DEFINE t TEXT
  CALL t.readFile("mytext.txt")
  RETURN 3, "hello world", t
END FUNCTION

In this example the help2 function returns an integer in the header and a string and a text file as multipart in the body.
Figure 131: Output of multipart HTTP response

In the output the header is given the default name, "rv0", at runtime.

In the multipart response parts are combined into one or more sets of data in the body. Parts are separated by boundaries, the \--MIME_boundary in the output example.

The text file part is identified by the name ("txt_file") that is defined via the WSName on page 3475 attribute.

Related concepts
- Download a file as an attachment to a response on page 3387
  This example demonstrates how to return a file as an attachment using the WSAttachment attribute
- Download a large object in the response body on page 3390
  This example demonstrates how to send a large object in the message response body.

Example: upload a file in a multipart request
Shows an example of a method you might use from a client to upload an image along with other data in a function using a form-data type HTTP multipart request.

Example multipart request

```
IMPORT os
IMPORT com

PUBLIC DEFINE myerror RECORD ATTRIBUTE(WSError="My error")
  code INTEGER,
  reason STRING
END RECORD

PUBLIC FUNCTION FetchFiles(
  id INTEGER,
  image STRING ATTRIBUTE(WSAttachment,WSMedia="image/*"),
  submit STRING)
ATTRIBUTES (WSPost,
  WSPath="/files/fetch",
  WSDescription="upload image file to the server from browser",
  WSThrows="400:@myerror")
RETURNS STRING ATTRIBUTE(WSMedia="text/html")
```
DEFINE ret, new_fname STRING
DEFINE ok INTEGER
IF id == 0 THEN
    LET myerror.code=999
    LET myerror.reason="Missing id"
    LET ret = SFMT("<HTML><body><h1>Must have an ID %1</h1></body></HTML>",id)
    CALL com.WebServiceEngine.SetRestError(400,myerror)
ELSE
    TRY
        LET new_fname = os.Path.basename(image)
        LET new_fname = os.Path.pwd()||"/"||new_fname
        LET ok = os.Path.RENAME(image,new_fname)
        LET ret = SFMT("Got image with ID %1",id)
    CATCH
        LET myerror.reason="Error uploading file"
        LET ret = SFMT("Error uploading file %1",id)
        CALL com.WebServiceEngine.SetRestError(400,myerror)
    END TRY
END IF
RETURN ret
END FUNCTION

The image file format is specified via the WSMedia on page 3478 attribute. The attachment is handled through the STRING data type and the WSAttachment and WSMedia attributes.

The "FetchFiles" function needs to be called through an HTML form for the user to select the file to upload. The form needs to have two input fields and one submit button, according to the function's input parameters.

1. Create the form from the code sample. Remember to change the action tag value to the URL of your server.
2. Save as, for example, upload.html on the client side.
3. Open the file locally in a browser to upload a file. Ensure the service is running.

```html
<!DOCTYPE html>
<html>
  <FORM NAME="upload" method="post" action="http://localhost:8090/Myservice/files/fetch" enctype='multipart/form-data'>
    <div>
      <label for="name">ID:</label>
      <input type="text" name="id"/>
    </div>
    <div>
      <label>pictures:</label>
      <input type="file" name="image" accept="image/png, image/jpeg" />
    </div>
    <input type="submit" name="submit" value="Send"/>
  </FORM>
</html>
```

In the output you see that the image is sent in parts that are combined into one or more sets of data in the body. Parts are separated by boundaries, the

```
---------------------------70482736912266
```

string in the output example.

On successfully receiving the file, the server returns the message to display in the browser. Otherwise, an error is returned and displayed.
POST /MyService/files/fetch HTTP/1.1
WS-DEBUG (Receive)
Host: localhost:8090
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:64.0) Gecko/20100101 Firefox/64.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip, deflate
Content-Type: multipart/form-data; boundary=----------------------------70482736912266
Content-Length: 697
Connection: keep-alive
Upgrade-Insecure-Requests: 1

WS-DEBUG (Receive)
----------------------------70482736912266
Content-Disposition: form-data;
name="id"

-----------------------------70482736912266
Content-Disposition: form-data;
name="name"
filename="gwc-jg_hamburger_icon.png"

-----------------------------70482736912266
Content-Disposition: form-data;
name="image"
filename="image.png"

-----------------------------70482736912266
Content-Disposition: form-data;
name="image" filename="image.png"

-----------------------------70482736912266
Content-Disposition: form-data;
name="image"
filename="image.png"

HTTP/1.1 200 Success
Content-Type: text/html; charset=UTF-8
Server: GWS Server (Build 1544519153)
Connection: close
Date: Fri, 25 Jan 2019 13:02:31 GMT
Content-Length: 0
Content-Encoding: 

<HTML><body><h1>Got image with ID 2</h1></body></HTML>

Figure 132: Sample output of multipart HTTP request

Related concepts
Example: upload a file as attachment in request body on page 3388
Shows a sample function uploading an image file as attachment in the message request body.

Upload a large object in the request body on page 3391
This example demonstrates how to upload a large object in the message response body.

Set data format with WSMedia on page 3393
It is important to set the correct MIME type for a Web service request or response. You can specify the data format via the WSMedia attribute.

Handling application level errors
There are many situations in which you need to notify an error to a client using your REST Web service.

Errors in calls to Web services may occur for various reasons; for example:

- the user input is incorrect
- the resource does not exist
- the server response failed
- the client does not have access to the resource
- the database connection failed
In these cases, you normally return an HTTP status code in the range 400 to 599 with its default message. The most common ones you encounter are as follows:

- **client (400-499)**
  - 400 Bad Request
  - 401 Unauthorized
  - 403 Forbidden
  - 404 Not Found
- **server (500-599)**
  - 500 Internal Server Error
  - 501 Not Implemented

For information on the full range of HTTP status codes see RFC 2616.

There are some situations where it is more useful to describe the error specifically, and in your REST function you can use the error handling attributes for this purpose.

**Handling errors with attributes**

In GWS REST you can perform error handling using two attributes:

- with the `WSThrows` attribute, set on the `ATTRIBUTES()` clause of the function, you list errors that may be encountered accessing a resource. From this the client stub generated from the OpenAPI specification, handles the error corresponding to those HTTP status codes.
- with the `WSError` attribute, set on a public variable defined at the modular level, you can provide a description of the status-code when an error is encountered at runtime to replace the standard HTTP status code description.

In your function, you code to trap an error at runtime with a call to the `SetRestError()` method, to return the HTTP status code and a description to the client.

**Publishing errors**

When you generate the service description, the errors you declared in the `WSThrows` attribute, the code reference and the error detail, are published in the "responses" section for that function in the OpenAPI specification file. The modular variables declared with `WSError` attribute are found in the "component" section at the end of the specification file.

**Related concepts**

- Generate service description on demand on page 3406

You can access a description of your GWS REST Web service through a query string on the URL of the Web service.

**Example: handling expected application errors**

Error handling is supported with the `WSThrows` and `WSError` attributes.

In your REST function expected errors, such as application level errors, are listed in the `WSThrows` attribute defined in the `ATTRIBUTES` clause of the function.

**Example using WSError with WSThrows**

In this example the expected error is for when a resource is not found so the HTTP status code "400" is listed in the `WSThrows` attribute. It also references the "userError" variable (@userError) defined with the `WSError` on page 3483 attribute for the description of the error at runtime.

**Note:** In the example, the variable is a record but you can define this as any suitable Genero BDL simple type.

You code to trap the error at runtime when a customer record is not found by updating the `WSError` attributed variable with a specific description of the HTTP status code, and calling the `SetRestError()` method to return it.

```import com```
PUBLIC DEFINE userError RECORD ATTRIBUTE(WSError="User error")
  message STRING
END RECORD

PUBLIC FUNCTION GetCustomersNameById( id STRING ATTRIBUTE(WSQuery) )
  ATTRIBUTES(WSGet,
    WSPATH="/customers",
    WSThrows="400:@userError")
RETURNS STRING
  DEFINE s STRING
  WHENEVER ERROR CONTINUE
    SELECT lname INTO s FROM customers WHERE @id = id
  WHENEVER ERROR STOP
  IF SQLCA.SQLCODE == NOTFOUND THEN
    LET userError.message = SFMT("Could not find customer id :%1",id)
    CALL com.WebServiceEngine.SetRestError(400,userError)
  END IF
  RETURN s
END FUNCTION

Related concepts
Example: handling an unexpected error on page 3401
Example of a method you can use to code for unexpected errors.

Example: handling an unexpected error
Example of a method you can use to code for unexpected errors.

Warning: In general, the recommended option is to list all the possible errors in order to have them generated in the OpenAPI specification file and trapped on the client side.

Not all errors can be anticipated. In this case, an unexpected error can be handled in your REST function. Unexpected error are the ones not listed in the WSThrows attribute.

Example: managing unexpected errors
In this example, an expected error is listed in the WSThrows attribute. In the code, the OTHERWISE clause traps an error which is not listed in the WSThrows attribute.

The call to the SetRestError() method returns the HTTP status code to the client. The error description is set to NULL, which allows the HTTP standard error description to be returned, but you could also return a description in this case; optionally using the variable defined with the WSError on page 3483 attribute.

IMPORT com

TYPE accountType RECORD ... END RECORD

PUBLIC DEFINE myError RECORD ATTRIBUTE(WSError="user error")
  message STRING
END RECORD

PUBLIC FUNCTION queryAccountsById( id VARCHAR(10) ATTRIBUTE(WSParam) )
  ATTRIBUTES(WSGet,
    WSPATH="/accounts/{id}",
    WSThrows="400:@myError")
RETURNS accountType ATTRIBUTES(WSName="body")
  DEFINE thisAccount accountType
  WHENEVER ERROR CONTINUE
    SELECT * INTO thisAccount.* FROM accounts WHERE @id = id
  WHENEVER ERROR STOP
  CASE
    WHEN SQLCA.SQLCODE = 0
      EXIT CASE
    WHEN SQLCA.SQLCODE = NOTFOUND

LET myError.message = SFMT("Could not find account id :%1",id)
CALL com.WebServiceEngine.SetRestError(400,myError)
OTHERWISE
  CALL com.WebServiceEngine.SetRestError(505,NULL)
END CASE
RETURN thisAccount.*
END FUNCTION

Related concepts

Example: handling expected application errors on page 3400
Error handling is supported with the WSThrows and WSError attributes.

Handling security

You handle security in GWS high-level REST via scopes.

Security is configured based on the recommendations of the OpenAPI specification via scopes implemented by OAuth, and Bearer token authentication. OAuth allows the user to access the protected content from the resource server without sharing their credentials. Instead access is granted by access tokens forwarded to the GWS REST service.

To implement security the Genero Application Server delegate service must be running. For more information, see the How to implement delegation and Genero Identity Provider (GIP) pages in the Genero Application Server User Guide.

If there is no access token, or there is no delegate service providing the verification, the security function is not executed. Security is executed if, and only if, the request contains a scope definition, and its value matches what is in the WSScope attribute.

Note: Testing your services with WSScope:

- When testing your service in standalone mode without a GAS, the WSScope is not checked. However, when behind a GAS, the appropriate scope is required and you will need to deploy and secure the service with the Genero Identity Provider (GIP).
- Alternatively, if you need to integrate Genero REST services security into your own environment system, you can also write your own delegate service to validate any kind of token, extract the scope from it, and forward it to the REST service.

Handling security errors

Access token errors are automatically handled by the GWS engine. You do not need to do anything in your code. If the client request does not have the correct access token, the service will return HTTP 403.

Publishing scopes

When you generate the service description, the scope you declared in the WSScope attribute, the user or group member detail, is published in the "security" section for that function in the OpenAPI specification file. If you declare scopes with a modular variable with the WSInfo and WSScope attributes, the "security" tag at the end of the specification file contains the security details.

Related concepts

Generate service description on demand on page 3406
You can access a description of your GWS REST Web service through a query string on the URL of the Web service.

OAuthAPI overview on page 3504
The OAuthAPI library supports the OAuth protocol that authenticates user access and issues access tokens.

Example: set security with WSScope
You can set security using the WSScope attribute either at the function level or at the service level.

**Example 1 setting security at function level**

```plaintext
IMPORT com

TYPE profileType RECORD ...

PUBLIC FUNCTION FetchMyUserProfile( p_user_id INTEGER ATTRIBUTE(WSQuery) )
ATRIBUTES(
    WSGet,
    WSPath="/users",
    WSDescription="Returns a user profile, requires authentication",
    WSThrows="404:user not found",
    **WSScope**="profile, profile.me")
RETURNS profileType ATTRIBUTES(WSName="data",
    WSMedia="application/json, application/xml")

DEFINE p profileType
WHENEVER ERROR CONTINUE
SELECT * INTO p.* FROM users
    WHERE user_id == p_user_id
WHENEVER ERROR STOP
IF SQLCA.SQLCODE NOTFOUND THEN
    INITIALIZE p TO NULL
    CALL com.WebServiceEngine.SetRestError(404,NULL)
END IF
RETURN p.*
END FUNCTION
```

In this example the **WSScope** on page 3486 required for the REST operation to be executed is "profile" or "profile.me".

**Example 2 setting security at service level via WSInfo**

```plaintext
PUBLIC DEFINE serviceInfo
RECORD ATTRIBUTE(WSInfo,
    **WSScope**="users.fourjs")

    title STRING,
    version STRING,
    contact STRING
END RECORD
```

In this example the scope is set in the service information record of the module. The attributes set are **WSInfo** and **WSScope**. The scope required for all REST functions in the module to be executed is "users.fourjs".

**Related concepts**

- **WSInfo** on page 3488
  - Specify information about the Web service such as title, version, contact details, scopes, etc.

- **Register service with Identity Provider** on page 3403

  Register service with Identity Provider

  If you deploy a Web service on the Genero Application Server (GAS) that requires user authentication, you must first register the service with an Identity Provider that allows access to it.

  You can use the Genero Identity Provider (GIP) or a third party IdP, such as Google or Microsoft, etc., to do this. For more information, see the How to implement Single sign-on (SSO) and Genero Identity Provider (GIP) pages in the Genero Application Server User Guide.

  **Related concepts**

  - **Access to secure Web service** on page 3414
To access a secure REST Web service, the client application must have a valid access token.

**Provide service information**

Provide information about the service, such as title, version, contact details, etc., that is generated in the OpenAPI specification file.

In your service module:

1. Define a PUBLIC record with a `WSInfo` attribute. This record allows you to provide metadata about your REST service, which can be generated according to the OpenAPI standard and used by clients. This example record follows that OpenAPI specification:

```
# service module
PUBLIC DEFINE serviceInfo RECORD ATTRIBUTE(WSInfo)
   title STRING,
   description STRING,
   termOfService STRING,
   contact RECORD
      name STRING,
      url STRING,
      email STRING
END RECORD,
version STRING
END RECORD = (  
title: "my service",
version: "1.0",
contact: ( email:"helpdesk@mysite.com") )
```

**Note:** If you do not set a service information record, the default value as the name of the service module is used for the service title.

2. Add record fields as required. The GWS does not check the record structure used with `WSInfo` attribute. You are therefore allowed to add the fields you want to be available in the OpenAPI specification.

3. Add values for the record fields.

The information provided is then generated in the `openapi.json` or `openapi.yaml` specification file.

**Related concepts**

*Generate service description on demand* on page 3406

You can access a description of your GWS REST Web service through a query string on the URL of the Web service.

*Publish a REST service* on page 3404

In publishing the service you provide your service to users who can access it on the net.

**Publish a REST service**

In publishing the service you provide your service to users who can access it on the net.

Functions of your service you want to publish, must be defined as PUBLIC. When you publish a service, only the functions that are public are generated in the OpenAPI specification file. Functions that are private, are not available as operations of the service and are therefore not generated.

To publish a service, for example, in the service's main module, you:

1. Import the service module.
2. Call on the `com.WebServiceEngine.RegisterRestService()` method to register the service.

**Web service main module**

This is an example of a module that runs a Web service. The IMPORT FGL statement imports the service module "serviceModule", allowing its public-defined variables and functions to be referenced.
In the call to the `RegisterRestService()` method, the Web service module "serviceModule.4gl" is registered with the GWS server as a REST service. The name of the service is set to "MyService". This is the public name given to the REST service, used in the URI endpoints to access its resources.

The call to `ProcessServices()` is made to start the service process within a `WHILE` loop. Within the loop requests for services are processed and various Web service engine errors are handled. The Web service runs until interrupted.

```plaintext
IMPORT com
IMPORT FGL serviceModule

MAIN
    DEFINE ret INTEGER
    CALL com.WebServiceEngine.RegisterRestService("serviceModule", "MyService")
    DISPLAY "Server started"
    CALL com.WebServiceEngine.Start()
    WHILE TRUE
        LET ret = com.WebServiceEngine.ProcessServices(-1)
        CASE ret
            WHEN 0
                DISPLAY "Request processed."
            WHEN -1
                DISPLAY "Timeout reached."
            WHEN -2
                DISPLAY "Disconnected from application server."
                EXIT PROGRAM  # The Application server has closed the connection
            WHEN -3
                DISPLAY "Client Connection lost."
            WHEN -4
                DISPLAY "Server interrupted with Ctrl-C."
            WHEN -9
                DISPLAY "Unsupported operation."
            WHEN -10
                DISPLAY "Internal server error."
            WHEN -23
                DISPLAY "Deserialization error."
            WHEN -35
                DISPLAY "No such REST operation found."
            WHEN -36
                DISPLAY "Missing REST parameter."
            OTHERWISE
                DISPLAY "Unexpected server error " || ret || "."
            END CASE
        IF int_flag<>0 THEN
            LET int_flag=0
        END IF
    END WHILE
    DISPLAY "Server stopped"
END MAIN
```

**Related concepts**

Generate service description on demand on page 3406
You can access a description of your GWS REST Web service through a query string on the URL of the Web service.

**Generate service description on demand**

You can access a description of your GWS REST Web service through a query string on the URL of the Web service. The Genero REST Web service allows for the functions defined in your Web service to be generated according to the OpenAPI specification. You can choose to access the description by choosing to output it in either the JSON or YAML format:

http://host:port/gas/ws/r/group/xcf/resource?openapi.json

Or

http://host:port/gas/ws/r/group/xcf/resource?openapi.yaml

**Note:** The GWS service must be running on the specified port in order to provide the Web service information. The GWS server must be behind a GAS in production to handle the load.

The OpenAPI file allows you to see your entire Web service, including:

- Available resources (paths) and functions, for example, /add, /sub/{f}, /mult/{coef}/hello, etc., and operations (GET, POST, etc.)
- Parameters, input and output for each function
Figure 133: Parameters represented in the OpenAPI specification file

- Responses, such as error handling specifications, error codes and messages
Figure 134: Response codes represented in the OpenAPI specification file
- Service information, such as title, version, contact, etc.

```json
{
    "openapi": "3.0.0",
    "info": {
        "title": "my service",
        "version": "1.0",
        "contact": "fg@4js.com"
    },
    "servers": {
        "url": "http://Localhost:8090/Myservice"
    }
}
```

Figure 135: Sample of service information in OpenAPI specification file

### Generating the stub file

You can use the `fgrestful` on page 2085 tool provided by the Genero Web Services package to generate a stub file from the `openapi.json` file. The stub file is a Genero BDL file with code that will manage calls made by your client application to the Web service. For details, see the Get the RESTful Web service information on page 3410 page.
Testing the GWS service in stand-alone mode

If you are on a local server, you can start the GWS server as shown in Getting started with REST high-level on page 3375

Related concepts

Provide service information on page 3404
Provide information about the service, such as title, version, contact details, etc., that is generated in the OpenAPI specification file.

Get the RESTful Web service information on page 3410
To access a remote Web service, you must get the information from the service provider.

Steps to writing a REST client application on page 3410
Create, configure and deploy a Genero Web Services client generated from the high level REST framework.

Error codes of com.WebServicesEngine on page 2843
Error codes returned by com.WebServicesEngine methods.

Related tasks

Get service information from secure Web service on page 3411
To access a secure Web service, including its specification, you need an access token issued on behalf of an authorized user.

REST API versioning

Options for versioning the API can be used when changes to API functions impact existing client access.

Changes to the API may affect functionality clients already depend on. For example, a change to the format of the response data or type (changing an integer to a float), will impact existing users. The options for managing this change will require versioning the API so that clients are not disrupted.

A new version of an API can be implemented by versioning either the resource URI, or specifying the version in a custom header. There are advantages and disadvantage in using either method and the choice depends on your preference.

URI Versioning

Putting the version in the URI is the recommended approach.

For example, a resource at:

```
http://host:port/gas/ws/r/group/xcf/resource
```

Is now at:

```
http://host:port/gas/ws/r/group/xcf/version/resource/
```

Even though this in principle breaks the constraint that the URI should refer to a unique resource, it is the preferred way. In effect it means that you have two versions of the API and are able to generate a different openapi.json specification per version based on the path of your REST API.

Versioning using a custom header

Alternatively, the version can be included in a custom header.

```
api-version:v2.0
```

The version must be specified by the client. Clients need to know which headers to specify before requesting a resource.
The API function retrieves the value of the header. If the version matches one specified, then that is the method that will be invoked. Based on this, the API can decide if it is capable of fulfilling the request and respond to the client appropriately.

**Version syntax**

The version specified need not be numeric, nor specified using the “\(v[x]\)” syntax. Alternatives such as dates, project names, or other identifiers that you find meaningful and flexible enough to change as the versions change can be used.

*Example: set version with WSP\(\text{Path}\)*

You can set version using the WSP\(\text{Path}\) attribute.

**Example setting version in the path**

```plaintext
IMPORT com

TYPE profileType2 RECORD ... END RECORD

PUBLIC FUNCTION FetchMyUser2Profile(p_user_id INTEGER ATTRIBUTE(WSP\(\text{Param}\)) )
  ATTRIBUTES(
    WSGet,
    WSP\(\text{Path}\)="/users/v2/(p_user_id)",
    WSDescription="Returns a user profile for API v2 ",
    WSThrows="404:user not found"
  )
  RETURNS profileType2 ATTRIBUTES(WSName="data",
    WSMedia="application/json,application/xml")

DEFINE p profileType2
WHENEVER ERROR CONTINUE
SELECT * INTO p.* FROM users
  WHERE user_id == p_user_id
WHENEVER ERROR STOP
IF SQLCA.SQLCODE==NOTFOUND THEN
  INITIALIZE p TO NULL
  CALL com.WebServiceEngine.SetRestError(404,NULL)
END IF
RETURN p.*
END FUNCTION
```

In this example there are changes to the profile record type (profileType2) that are not compatible with the existing version of the API. The WSP\(\text{Path}\) on page 3469 for the new function contains a version number (v2) that provides the path to the resource. An example of the resource URL is:

```
http://host:port/gas/ws/r/group/xcf/Accounts/users/v2/1
```

**Related concepts**

- **WS\(\text{Info}\)** on page 3488
  Specify information about the Web service such as title, version, contact details, scopes, etc.

- **Steps to writing a REST client application**
  Create, configure and deploy a Genero Web Services client generated from the high level REST framework.

- **Get the RESTful Web service information**
  To access a remote Web service, you must get the information from the service provider.

  If the server supports the OpenAPI specification, you can use the fglrestful on page 2085 tool provided by the Genero Web Services package to obtain the necessary information (like the SOAP WSDL). For secure Web services you need to get an access token.

**Related concepts**

- **Calling a web service function** on page 3413
A GWS REST service is called through a client application.

Generate service description on demand on page 3406
You can access a description of your GWS REST Web service through a query string on the URL of the Web service.

Get service information from Web service
Using the fglrestful tool, you can generate the client stub directly from a REST Web service URL or an OpenAPI specification file.

The following steps describe the options.

1. Use the fglrestful tool to generate the stub.
   For example, from the Web service URL
   ```
   fglrestful -o clientStub http://host:port/gas/ws/r/group/xcf/resource?
   openapi.json
   ```
   For example, from an OpenApi service specification file
   ```
   fglrestful -o clientStub service-spec-file
   ```
   Where:
   a. clientStub specifies the file name of the stub in the output (-o) option.
   b. The URL of the Web service is specified with the query string ?openapi.json to get the specification file.
   Note: The "MyService" GWS REST service must be running on the specified port in order to provide the service information.

   The stub file can also be generated from a file containing the specification as shown in the second example.

   The stub file can be generated as described in the second example in step 1 on page 3411.

   2. Alternatively, to view the specification in the browser, you can use the Web service URL with the ?
   openapi.json query string.
   ```
   https://host:port/gas/ws/r/group/xcf/resource?openapi.json
   ```
   The OpenAPI specification is displayed formatted in JSON. You can copy and paste the output and save to a file. The stub file can be generated as described in the second example in step 1 on page 3411.

   Note: If you can not access the service because it is protected, you must get the access token using the
   getToken tool as described in Get service information from secure Web service on page 3411. For further
   information, see the getToken page in the Genero Application Server User Guide.

Related concepts
Access to secure Web service on page 3414
To access a secure REST Web service, the client application must have a valid access token.

Generate service description on demand on page 3406
You can access a description of your GWS REST Web service through a query string on the URL of the Web service.

Troubleshooting stub file creation on page 3413
Learn how to troubleshoot stub file creation.

Get service information from secure Web service
To access a secure Web service, including its specification, you need an access token issued on behalf of an authorized user.

You can get the token using the gettoken tool and provide it in the URL query string to get the OpenAPI specification. The gettoken tool is located in the FGLDIR/web_utilities/services/gip/bin/gettoken directory.
The following steps describe the process.

1. Use the gettoken tool to get the access token.
   Line breaks have been added to the command example to improve readability.

   ```bash
   fglrun GetToken.42r password -u user -p mypw
   --idp https://host:port/gas/ws/r/services/GeneroIdentityProvider
   --savetofile mytoken.json myWSScope
   ```

   Where:
   a. The `password` command is used to get an access token on behalf of an authenticated user.
   b. The `user` name and `password` provided (`-u user -p mypw`) is that of a registered user in the IdP.
   c. The URL of the Genero Identity Provider (GIP) is specified in the `--idp` option.
   d. `mytoken.json` is the file name where the access token is saved.
   e. `myWSScope` is the scope of the service required in the token.

   **Note:** The access token is valid for 10 minutes.

   For further information, see the GetToken page in the Genero Application Server User Guide.

2. Add the token in the Web service URL query string to get the specification.

   ```bash
   https://host:port/gas/ws/r/group/xcf/resource?
   openapi.json&access_token=token
   ```

   The OpenAPI specification is displayed in the browser formatted in JSON. You can copy and paste the output and save to a file. The stub file can be generated as described in Get service information from Web service on page 3411.

3. Or alternatively, you can use the token with the `fglrestful` on page 2085 tool to generate the stub.

   ```bash
   fglrestful -o clientStub http://host:port/gas/ws/r/group/xcf/resource?
   openapi.json -t token
   ```

   Where:
   a. `clientStub` specifies the file name of the stub in the output (`-o`) option.
   b. The URL of the Web service is specified with the query string `?openapi.json` to get the specification file.

   **Note:** The "MyService" REST service must be running on the specified port in order to provide the service information.
   c. The token is specified in the token (`-t`) option.

   The `clientStub.4gl` generated from the specification contains the complete code to manage calls made by client applications to the Web service resources. You need to compile and link the module into your Genero REST client application. For details, see Call the stub functions on page 3415.

**Related concepts**

- Access to secure Web service on page 3414
- Generate service description on demand on page 3406
- Troubleshooting stub file creation on page 3413
Learn how to troubleshoot stub file creation.

**Troubleshooting stub file creation**

Learn how to troubleshoot stub file creation.

The stub file generated from the specification file must be error free or otherwise the stub file does not get generated. A variety of issues may cause errors. The most common errors are those caused by incompatible types, missing variables, or other issues.

The `fglrestful` tool writes errors to the standard output. You can see these if errors are reported when you generate the stub file. Errors, such as the following, will stop the generation of the stub file.

```
Error in /paths/Property/UpdateProperties/{applicationID}/post/requestbody
Reason : ComplexType='UpdatePropertiesRequestBodyType' not supported on binary operation
```

The error message relates to incompatible types. Messages contain the path to where in the `openapi.json` file errors occurred, and what the reason was. You can refer to the specification file to try to isolate the problem and determine if it is an issue that you can resolve or one that needs the assistance of your local Four Js support center.

**Display warnings**

Warnings, unlike errors, will not stop the stub file creation but displaying them may help you debug. You can opt to display warnings by running the `fglrestful` command with the warning option set to `yes`:

```
fglrestful -W yes -o stub_filename http://localhost:8090/MyService?openapi.json
```

For example, this warning describes a function with a duplicate id.

```
Warning in /paths/Liability/GetReasonTypes/{subscriberId}
Reason : OperationId=GetReasonTypes has already been defined, append _1
```

**Related concepts**

`fglrestful` on page 2085

The `fglrestful` tool produces REST web services stub files for client programs using an OpenApi specification.

**Calling a web service function**

A GWS REST service is called through a client application.

*Write the MAIN program code block*

Import the client stub, and call its functions.

To call a REST service function, in your client's main module, you:

1. Import the client stub module
2. If you are accessing a secure service, code within the `MAIN` program block to get the access token before calling any functions. See Main program code for access to secure service on page 3414.
3. Call the functions within the `MAIN` program block

```
IMPORT FGL clientStub
MAIN
  DEFINE wsstatus INTEGER
  DEFINE rets STRING
  DEFINE opt1 INTEGER
  LET opt1=22

  CALL clientStub.getUsersName(opt1) RETURNING wsstatus, rets

  IF wsstatus = clientStub.C_SUCCESS THEN
    DISPLAY "User name is ":rets
  ELSE
    DISPLAY "Error in getting user details"
  END IF
```
This example is a very basic version of the code. See also Call the stub functions on page 3415.

Related tasks
Main program code for access to secure service on page 3414
Code to get the access token for a secure Web service.

Access to secure Web service
To access a secure REST Web service, the client application must have a valid access token.

The access token got from the Identity Provider must be registered with the GWS engine so that each HTTP request to the service can use it, otherwise an authorization error is returned. For more information on secured Web services, see Handling security on page 3402.

The stub file generated by fglrestful on page 2085 contains all the definitions and the functions that your GWS client application needs to retrieve the access token. There is a sample included in Main program code for access to secure service on page 3414 so you can see how to configure your application and code for a secure Web service.

Related concepts
Generate service description on demand on page 3406
You can access a description of your GWS REST Web service through a query string on the URL of the Web service.

OAuthAPI library on page 3504
The OAuthAPI library.

Main program code for access to secure service
Code to get the access token for a secure Web service.

To access a secure Web service, the user needs to get an access token that is provided by the Identity Provider securing the Web service. The OAuthAPI.init on page 3510 function can be used for this purpose if needed.

1. Set your app to use delegation in its application configuration file.
   This step is mandatory.

   `<APPLICATION Parent="defaultwa" ...>
   <EXECUTION>
   ...
   <DELEGATE service="services/OpenIDConnectServiceProvider">
   <IDP>IdP_URL</IDP>
   <CLIENT_PUBLIC_ID>XXXXXXXX</CLIENT_PUBLIC_ID>
   <CLIENT_SECRET_ID>XXXXXX-XXXXXX</CLIENT_SECRET_ID>
   </DELEGATE>
   ...
   </EXECUTION>
   </APPLICATION>`

   Where:
   - The OpenIDConnectServiceProvider is the delegation REST Web service in the $FGLDIR.
   - The IdP_URL can have an entry of localhost when everything runs on the same Genero Application Server. Otherwise, you must provide the Genero Identity Provider (GIP) URL. For example: http://othermachine.com:6394/ws/r/services/GeneroIdentityProvider
   - The OAuth access tokens for CLIENT_PUBLIC_ID and CLIENT_SECRET_ID are those you get from the IdP.

   For further information, see the Configure delegation for application or service page in the Genero Application Server User Guide.

2. In the MAIN/END MAIN clause of your client app, call the OAuthAPI.init function to get the OAuth access tokens at runtime. This must be done before calling any other service functions.
For example:

```
IMPORT FGL OAuthAPI

DEFINE my_user_id INTEGER

MAIN
  # ...

  # Init OAuthAPI
  IF NOT OAuthAPI.init(5, "AF350CBC-8801-4DFB-9A78-A95B25BB32AF", "8JEq3HBfXrmj/8vMP66iaRQnGrWVyjqr") THEN
    DISPLAY "Error: unable to initialize OAuth"
    EXIT PROGRAM 1
  ELSE
    LET my_user_id = OAuthAPI.getIDSubject
  END IF

  # ...

END MAIN
```

You can get user information coming from the IdP from variables with the prefix `OIDC_`. For example,

```
LET userEmail = fgl_getenv("OIDC_EMAIL")
```

For an example of OAuthAPI calls, see the `consoleApp` source in `FGLDIR\web_utilities\services\gip\src\console`.

When using a third-party IdP, if it supports OpenID Connect, then the OAuthAPI can be used the same as for Genero Identity Provider. For further information, see the OpenID Connect SSO pages in the Genero Application Server User Guide.

**Related concepts**

- [OAuthAPI library](#) on page 3504
  - The OAuthAPI library.
- [Access to secure Web service](#) on page 3414
  - To access a secure REST Web service, the client application must have a valid access token.
- [Write the MAIN program code block](#) on page 3413
  - Import the client stub, and call its functions.

**Call the stub functions**

Functions in the stub file allows you to write code in your own `.4gl` module that calls these functions as part of your client application.

The service information obtained from the REST Web Service provides information about using the service.

**Using parameters and return values**

To call a function in your client application, you must define variables for its input parameters and return values. These, for example, may be simple types or user-defined records.

Details for these variables can be obtained from the OpenAPI specification file (see [Generate service description on demand](#) on page 3406) or from the generated function in the stub file.

**Record type defined in client stub**

```
PUBLIC TYPE addUserRequestBodyType RECORD
  users_name STRING,
  users_address STRING,
  users_country STRING
```
Calling a stub function in the client app

In this example the this variable is defined as a record type based on a user-defined record in the client stub, clientStub.addUserRequestBodyType.

Values are passed to the variable before calling the function.

Variables are defined for the return code (wsstatus) and for error details returned by the call (res). See Handle GWS REST server errors on page 3416.

```
IMPORT FGL clientStub
FUNCTION myWScall()
    DEFINE this clientStub.addUserRequestBodyType
    DEFINE wsstatus INTEGER
    DEFINE res STRING
    LET this.user_name= "Mike Pantock"
    # ...
    CALL clientStub.addUser(this.*)RETURNING wsstatus, res
    CASE wsstatus
        WHEN clientStub.C_SUCCESS
            DISPLAY "Success adding new user"
            # ... function code
        OTHERWISE
            DISPLAY "Unexpected error :", wsstatus, res
    END CASE
END FUNCTION
```

Handle GWS REST server errors

When a call to a Genero Web Services REST function returns a status, you can check for server errors.

The error codes described in the "responses" section of the OpenAPI specification file are generated as constants on the client stub. You can reference these to trap errors in your code.

Error codes generated for expected errors in client stub by fglrestful

If it is an expected error (other than success), a value above 1000 is returned. In this example constants are generated in the client stub from the OpenAPI specification.

```
# Error codes
PUBLIC CONSTANT C_SUCCESS = 0
PUBLIC CONSTANT C_INVALID = 1001
PUBLIC CONSTANT C_NOTAVAILABLE = 1002
PUBLIC CONSTANT C_MYERROR = 1003
PUBLIC CONSTANT C_INTERNAL_SERVER_ERROR = 1004
```

If a record type for error handling is generated in the client stub, it can be referenced in your client application.

```
# generated myErrorErrorType
PUBLIC TYPE myErrorErrorType RECORD
    code INTEGER,
    reason STRING
END RECORD
```

```
# My error
PUBLIC DEFINE myError myErrorErrorType
```
Trapping server errors on the client side

A variable (wsstatus in the example) is defined to trap the status code in the return.

Details of expected errors in the response from the server can be displayed in the reference to the variable defined in the client stub, clientStub.myError in the example.

It is recommended to always handle unexpected errors (CASE OTHERWISE in the code example), as these may arise due to different transport layers. For an unexpected HTTP error, the HTTP status code (200-599) is returned, otherwise -1 is returned.

```
IMPORT FGL clientStub

# ... function performs call to the client stub function
FUNCTION myWSRESTcall()
    DEFINE rets INTEGER
    DEFINE wsstatus INTEGER

    DEFINE p_body clientStub.updateUsersRequestBodyType
    LET p_body.users_id = "4"
    LET p_body.users_name = "new name to be decided"
    # ...

    CALL clientStub.updateUsers(p_body.*) RETURNING wsstatus, rets
    CASE wsstatus
        WHEN clientStub.C_SUCCESS
            DISPLAY "Updated the user:"
        WHEN clientStub.C_INVALID
            DISPLAY "error code   ":", clientStub.myError.code
            DISPLAY "error reason ":", clientStub.myError.reason
        WHEN clientStub.C_NOTAVAILABLE
            DISPLAY "error code   ":", clientStub.myError.code
            DISPLAY "error reason ":", clientStub.myError.reason
        OTHERWISE
            DISPLAY "error code   ":", clientStub.myError.code
            DISPLAY "error reason ":", clientStub.myError.reason
            DISPLAY "wsstatus is : ", wsstatus
            DISPLAY "Unexpected error", rets
    END CASE
END FUNCTION
```

Change REST client behavior at runtime

The global endpoint record allows you to change the client behavior at runtime.

The high-level REST Genero Web Services generates a global record called tGlobalEndpointType that allows you to change the client behavior at runtime without the need for modifying any generated client stub.

**Global Endpoint type definition**

Bindings defined for the REST Web service can be changed at runtime.

The global endpoint type is generated when the fglrestful tool generates the stub file.

At runtime you can use endpoint to change the client behavior, such as the Web server address the client is bound to, or to change other bindings exposed on the Web service.

**REST global Endpoint type**

```
TYPE tGlobalEndpointType RECORD # REST Endpoint
    Address RECORD
        Uri STRING
    END RECORD,
    Binding RECORD
        Version STRING, # HTTP Version (1.0 or 1.1)
        ConnectionTimeout INTEGER,
        ReadWriteTimeout INTEGER,
        CompressRequest STRING
```
Description of variables:

- **Address.Uri**: Represents the location of the server.
- **Binding.Version**: Represents the HTTP version to use for communication (only 1.0 or 1.1 allowed, default is 1.1).
- **Binding.ConnectionTimeout**: Represents the maximum time in seconds to wait for the establishment of the connection to the server.
- **Binding.ReadWriteTimeout**: Represents the maximum time in seconds to wait for a connection read or write operation before breaking the connection.
- **Binding.CompressRequest**: Represents the compression option specified in the header Content-Encoding, gzip or deflate.

**Public endpoint variable**

In the stub file, an `Endpoint` variable of type `tGlobalEndpointType` is defined as public. The address of the Web service is set when the stub is generated by the Genero Web Services engine.

```plaintext
PUBLIC DEFINE Endpoint
tGlobalEndpointType
  = (Address: (Uri: "http://localhost:8090/MyService"))
```

**Set server endpoint**

Set the REST Web service location from client app at runtime.

To change the server location at runtime, set the record `Uri` member with a valid URL of another service. If you leave the variable unset, the client will connect to the server URL defined in the stub file at code generation time.

**Set server address**

In the stub file, an `Endpoint` variable of type `tGlobalEndpointType` is defined as public. You can set the address of the Web service via the client application with for example:

```plaintext
```

**RESTful Web services with low-level APIs**

With Genero Business Development Language, you can write a RESTful Web Services client or server application.

**The Calculator tutorial**

There are several ways to implement REpresentational State Transfer (REST) Web services. This documentation uses a tutorial based on a calculator Web services application, using the low-level APIs with URIs and verbs.

This tutorial is provided from both the client and the server perspective. In **Writing a Web services client application** on page 3418, you learn how to create a client application that accesses the Add function in a calculator Genero Web services server application, with the option to perform the data exchange in either JSON or XML. In **Writing a Web Services server application** on page 3430, you learn how to implement the Web service on the server side.

**Writing a Web services client application**

Create, configure and deploy a RESTful Genero Web Services (GWS) client.

In this section, you will learn about the basic steps of creating a client application to access the "Add" function in the calculator RESTful GWS Web Service that is detailed in the **Calculator RESTFul Web services server application**. The topics explain:

1. What functions are available on the Server, and which query parameters/arguments to use with the URIS (resource). See **Step 1: Obtain information about Web service resources** on page 3419.
2. What GWS packages you need, see Step 2: Import extension packages (com, xml, util) on page 3419.
3. What Genero Business Development Language records you need to define, see Step 3: Define the records on page 3420.
4. How to request the Web service functions. Step 4: Build the HTTP request on page 3421
5. How to get the result and handle errors. Step 5: Process the HTTP response on page 3422.

In discussing the above five steps, the Calculator RESTFul Web services client application is used to provide code examples.

**Step 1: Obtain information about Web service resources**
To access a remote RESTful Web services server, you must first get information about the services offered by a server from the service provider.

In REST, the name of the resources or functions and what query parameters or arguments to use with them are generally all that you require, as this information contains everything the client needs to interact with the required resource.

**Add Function**
In the code example for the Calculator RESTFul Web services server application, we know that the calculator demo service performs various calculator functions. Our client application simply wants to access the function to add values.

```plaintext
# Calculator Web Service add function

FUNCTION add()
  LET add_out.r = add_in.a + add_in.b
  LET add_out.status.code = 0
  LET add_out.status.desc = "OK"
END FUNCTION
```

The HTTP request must therefore use an URI with a resource name and the required parameters that matches the calculator function defined in the calculator Web service server side. Therefore, Add becomes the resource name part of the URI we need to include in our code.

```plaintext
# Sample request to add two numbers

GET http://localhost:8090/add?a=1&b=2
```

In the next step we start to code the client application, Step 2: Import extension packages (com, xml, util) on page 3419

**Step 2: Import extension packages (com, xml, util)**
The functions you need to create a REST Web Service client application are contained in the classes that make up the com package of the Genero Web Services (GWS). Use the IMPORT statement to include the required packages.

Since this example application also uses util.JSON and XML class data types for the data exchange, you need to include the following lines at the top of your module as instructions to import the required packages:

```plaintext
IMPORT com
IMPORT xml
IMPORT util
```

Inside your module's "MAIN" code block:

- Declare variables of the com class to handle the HTTP request and response.
- Declare xml.DomDocument and xml.DomNode objects of the XML class to handle the XML option for data exchange.

```plaintext
MAIN
  DEFINE req com.HttpRequest
```
In the next step we define some records **Step 3: Define the records** on page 3420

**Related concepts**
The util package on page 2726
These topics cover the classes for the util package.

**Step 3: Define the records**
In this step you define the records you need for the HTTP Request and Response and the processing of the data.

Declare a record to use for HTTP Request/Response status. The calculator demo application creates a user-defined `TYPE(TYP_status)` to reference when defining variables.

```vcl
TYPE TYP_status RECORD
  code INTEGER,   # The HTTP status code
  desc STRING    # The HTTP status description
END RECORD
```

Declare a public record called `info` to hold the pieces of information that make up the HTTP request you send out and that comes back in the response from the Web service. You will use this when processing the data and outputting it for display.

```vcl
PUBLIC DEFINE info RECORD
  url STRING,       # URI of resource on the server
  verb STRING,      # HTTP method (POST, PUT, GET, or DELETE)
  reqtype STRING,   # Request type identifier (Content-Type)
  request STRING,   # Request data format (application/json, or application/xml)
  status STRING,    # For HTTP Response
  resptype STRING,  # Response type identifier (Accept)
  response STRING,  # Response data format (application/json, or application/xml)
  result RECORD     # Record for runtime execution errors
    code INT,
    desc STRING
END RECORD
```

Declare two records, one (`add_in`) to hold operands for the add calculation and another (`add_out`) to hold the result of the calculation and response status. Recall that the type `TYP_status` was defined at the top of the module.

```vcl
DEFINE add_in RECORD
  a INTEGER,
  b INTEGER
END RECORD,

DEFINE add_out RECORD
  status TYP_status,
  r INTEGER
END RECORD
```

At this stage we are ready to code the HTTP Request, **Step 4: Build the HTTP request** on page 3421
Step 4: Build the HTTP request

In this step you code the HTTP request and perform the request to the server to add two numbers.

Include code inside a TRY/CATCH block to trap runtime exceptions. Assign two values to fields in the add_in record as operands for the calculation. For details about the add_in record we use in this topic, see Step 3: Define the records on page 3420.

```
TRY
  LET add_in.a = 1
  LET add_in.b = 2
CATCH
  ...
  ...
END TRY
```

Recall we defined the variable req as an object of the com.HTTPRequest class earlier, see Step 2: Import extension packages (com, xml, util) on page 3419. We instantiate this now by invoking the com.HttpRequest.Create class method to create an URI for our HTTP request; adding to the URI string:

- The resource name or name of the server function, "add" in our case.
- A question mark ("?"), to indicate a query string.
- A query string that contains two key/value pairs in the format "field=value" separated by an ampersand ("&"), for example ("a=1&b=2"). They make up the query parameters or arguments required by the server's "add" function which will be extracted for processing by the Web service. We assign the values for the calculation from the "add_in" record.

```
LET req = com.HttpRequest.Create("http://localhost:8090/add?a=" || add_in.a || ";b=" || add_in.b)
```

**Note:** In our example, we anticipate the calculator Web server is on the localhost. For a production environment you need to take care of URI parsing: making sure that the URI matches what your service expects.

Set the HTTP method with the "GET" verb to request the service from the calculator Web resource. We do this by calling the com.HTTPRequest object's setMethod function referenced by the req variable.

```
CALL req.setMethod("GET")
```

Next we add the Accept and Content-Type headers to the com.HTTPRequest object referenced by the req variable by calling the setHeader function. These headers specify to the Web service how we intend to receive and deliver content in the message body's media type, JSON or XML. Our preference is for JSON.

```
CALL req.setHeader("Content-Type", "application/json")
CALL req.setHeader("Accept", "application/json")
```

Finally, we are ready to perform the HTTP request and call the com.HTTPRequest object's doRequest function.

```
CALL req.doRequest()
```

In the next step we handle the response from the Web service, Step 5: Process the HTTP response on page 3422.
**Step 5: Process the HTTP response**

In the final step you process the HTTP response; check for errors, and process the data.

At this stage a response to our request from the Web service has been received and is stored in the variable, `resp`. We defined the variable `resp` as an object of the `com.HttpResponse` class earlier, see Step 2: Import extension packages (com, xml, util) on page 3419. Now it is time to process the response.

```plaintext
LET resp = req.getResponse()
```

Start checking for errors. First get the HTTP status code returned in the response by calling the `com.HttpResponse` object's `getStatusCode` function referenced by the `resp` variable. Assign the status code to the "status" field of the `info` record (for details about this record see Step 3: Define the records on page 3420).

```plaintext
LET info.status = resp.getStatusCode()
```

The standard HTTP code of 200 indicates a successful operation, so we check for this before looking at the data.

```plaintext
IF info.status = 200 THEN
```

If the HTTP response was good, we save the value for Content-type response header to the "resptype" field of the `info` record. The Content-type header defines the type associated with the message body's byte sequence. REST APIs commonly use "application/json" or "application/xml" to reveal the format of the message body.

```plaintext
LET info.resptype = resp.getHeader("Content-Type")
```

**Process for XML**

```plaintext
IF info.resptype.getIndexOf("/xml",1) THEN
    LET doc = resp.getXmlResponse()
    LET node = doc.getDocumentElement()
    CALL xml.Serializer.DomToVariable(node, add_out)
    LET info.response = node.toString()
ELSE # JSON response processing
    [...] END IF
```

If the Content-type indicates XML format, we store the HTTP response as data in the `xml.DomDocument` object referenced by the `doc` variable using the `getXmlResponse` function. We then get a reference to the root node of the `doc` variable object by instantiating the `node` variable defined as an object of the `xml.DomNode` class using the `getDocumentElement` function.

Recall the `doc` and `node` variables were defined earlier, see Step 2: Import extension packages (com, xml, util) on page 3419.

To serialize the XML data into the `add_out` variable, we pass a reference to the `node` object variable in a call to the `xml.Serializer.DomToVariable` function. To convert the data referenced in the `node` object variable to string, we use the `xml.DomNode.toString` function to store it in the "response" field of the `info` record.

**Process for JSON**

If the Content-type does not indicate XML format, then we process the response for JSON, as we anticipate this as the only other option REST APIs use to format the message body.

We store the HTTP response as a string in the "response" field of the `info` record with a reference to the `resp` variable, object of the `com.HttpResponse` class, using the `getTextResponse` function. Then we pass a reference to the `info` record in a call to the `util.JSON.parse()` function to convert the HTTP response to JSON format into the `add_out` record.

```plaintext
IF info.resptype.getIndexOf("/xml",1) THEN
```
# XML response processing

ELSE # JSON response processing
    LET info.response = resp.getTextResponse()
    CALL util.JSON.parse(info.response, add_out)
END IF

## Outputting to the display

Finally, we output the HTTP response to the display depending on the status code.

IF add_out.status.code = 0 THEN
    DISPLAY add_in.a, "+", add_in.b, "+", add_out.r # Display the response, status and result
ELSE
    DISPLAY "]", add_out.status.code, "]", add_out.r
END IF

## Handling Errors

If the HTTP request failed, we handle the error by formatting the HTTP response status code and description into the "response" field of the info record. Other program errors are caught in the CATCH block and stored in "result" fields of the info record.

TRY
    [...] IF info.status = 200 THEN [...] ELSE
    LET info.response = SFMT("[%1] %2", resp.getStatusCode(), resp.getStatusDescription())
END IF
CATCH
    # Catch other runtime execution errors from the SQLCA diagnostic record
    LET info.result.code = status
    LET info.result.desc = sqlca.sqlerrm
END TRY

This completes the tutorial for the calculator demo client application. We have seen the basic steps that a RESTful client application must perform in order to carry out a simple request to a Web service. To review the complete source code, see The RESTful calculator demo source on page 3436.

Each Web services client application will be different in the type of processing that is done. For more examples of Web service applications, see the code samples provided with the package in demo/WebServices.

### Configure a WS client to access an HTTPS server

Configuration steps to access a server in HTTPS.

To configure access to an HTTPS server, you need a client certificate.

**Important:** On GMI mobile devices the iOS SSL/TLS layer is used for HTTPS, and the device Keychain typically holds the server certificate authority list. Therefore, the FGLPROFILE file security settings are ignored with the exception of the following: security.global.ca, security.global.ca.lookuppath, and security.global.systemca.

Before you begin, there are options to consider depending on how you wish to use the client certificate:

- If you do not have the certificate information in your FGLPROFILE file, Genero Web Services creates a certificate for you. This is an implicit or temporary certificate that is valid for a session only. For more information see HTTPS configuration on page 3305.
  
  **Note:** For the implicit certificate, no configuration is required.
• Alternatively, for stronger security, you generate a client certificate of your own, configure your application to use the client certificate generated, and add the configuration details to the FGLPROFILE file. Follow the steps outlined in this section.

**Note:** In a production environment, some servers provide a client certificate and you use the certificate as provided, and add the configuration details to the FGLPROFILE file.

The **openssl command line tool** can be used to create your own certificates for the configuration of secured communication. The following steps outline the configuration process:

**Related concepts**

- **Configuring the Apache Web server for HTTPS** on page 3460
  Configuration steps to secure a Web service for Apache server in HTTPS.

- **Certificates in practice** on page 3313
  Procedures and tools for creating, importing, and viewing certificates and keys.

- **Encryption and authentication** on page 3299
  A scenario involving a person (Georges) and his bank guides you through the concepts of secured communication, certificates, and certificate authorities.

- **Accessing secured services** on page 3303
  Security and authentication are important. Genero Web Services provides various communications options for a client to connect to a Web service.

**Create the client certificate**

Generate a root certificate authority that signs a client certificate.

In this task you create your own root certificate authority and client certificate using the OpenSSL command line tool.

1. Create the root certificate authority.
   a) Create the root certificate authority serial file.
      ```
      $ echo 01 > MyCompanyCA.srl
      ```
      This command creates a serial file with an initial HEX value 01. OpenSSL uses this file to track the serial numbers of certificates it creates. The serial file is typically given the same name as the CA with the extension .srl.
   
   b) Create the Certificate Signing Request (CSR) and private key.
      ```
      $ openssl req -new -out MyCompanyCA.csr -keyout MyCompanyCA.pem
      ```
      Follow the instructions to create the CSR.
   
   c) Create the Root Certificate Authority for a period of validity of 2 years.
      ```
      $ openssl x509 -trustout -in MyCompanyCA.csr -out MyCompanyCA.crt -req -signkey MyCompanyCA.pem -days 730
      ```
      **Note:** The private key file (MyCompanyCA.pem) of a Root Certificate Authority must be handled with care. This file is responsible for the validity of all other certificates it has signed. As a result, it must not be accessible by other users.

2. Create the client's X.509 certificate and private key.
   a) Create the client's Certificate Signing Request and private key.
      ```
      $ openssl req -new -out MyClient.csr
      ```
      **Note:** By default, openssl outputs the private key in the privkey.pem file. If you want to specify a different file name, or if your openssl version does not output the private key by default, add the -keyout <myprivkey>.pem to the command.
   
   b) Remove the password from the RSA private key.
\$ openssl rsa -in privkey.pem -out MyClient.pem

**Note:** The unprotected private key is output in MyClient.pem.

c) Create the client's certificate (self-signed X.509 certificate valid for a period of 1 year) trusted by the Root Certificate Authority created in step 1.

(line breaks added for document readability)

\$ openssl x509 -in MyClient.csr -out MyClient.crt -req -signkey MyClient.pem -CA MyCompanyCA.crt -CAkey MyCompanyCA.pem -days 365

**Note:** Most servers do not check the identity of the clients. For these servers, the client's certificate does not necessarily need to be trusted; it is only used for data encryption purpose. If, however, the server performs client identification, you must trust a Certificate Authority in which it has total confidence concerning the validity of the client's certificates.

**Note:** The purpose of the client's certificate is to identify the client to any server; therefore the subject of the certificate must correspond to the client's identity as it is known by the servers.

**Note:** To import the certificate in a keystore you can create a pkcs12 certificate. See Import a certificate and its private key into the Windows key store on page 3315.

**What to do next**

Configure your FGLPROFILE file for the client certificate. See Configure for the client certificate on page 3307.

**Related concepts**

Encryption and authentication on page 3299

A scenario involving a person (Georges) and his bank guides you through the concepts of secured communication, certificates, and certificate authorities.

The OpenSSL (openssl) tool on page 3313

The openssl command line tool creates certificates for the configuration of secured communications.

**Related tasks**

View a certificate on page 3316

This procedure allows you to view a certificate using the openssl command.

Configure for the client certificate

Configure your application to use the client certificate generated. For production systems, you add the configuration details to your FGLPROFILE file.

You have generated a client certificate as described in Create the client certificate on page 3306.

In this task you add the certificate information to your FGLPROFILE file.

Add the client's security configuration to your FGLPROFILE file.

The client security entry defines the certificate and the associated private key used by the Genero Web Services client during an HTTPS communication. The security entry must be defined with an unique identifier (**id1** in this example).

```plaintext
security.id1.certificate = "MyClient.crt"
security.id1.privatekey = "MyClient.pem"
```

**Note:** If the private key is protected with a password, you must remove it or create a script that returns the password on demand. See Create the client certificate.

Your applications are configured to use the client certificate.

**What to do next**
Create the client's certificate authority list and configure the global certificate authority list (security.global.ca) in your FGLPROFILE file. See Configure for the certificate authority list on page 3308.

**Related concepts**

**FGLPROFILE entries for Web Services** on page 3489

The FGLPROFILE entries relating to Genero Web Services are divided between five categories: security, basic or digest HTTP authentication, proxy configuration, server configuration, and XML cryptography.

**FGLPROFILE: HTTP(S) Proxy Authentication** on page 3496

FGLPROFILE entries can be used to define a connection to an HTTPS server via a proxy, and with HTTP and Proxy Authentication.

**Encryption and authentication** on page 3299

A scenario involving a person (Georges) and his bank guides you through the concepts of secured communication, certificates, and certificate authorities.

**Related tasks**

**View a certificate** on page 3316

This procedure allows you to view a certificate using the openssl command.

**Configure for the certificate authority list**

The client needs to check to see if the server certificate is trusted. This is done using a certificate authority list. In this task you create the client's certificate authority list using the OpenSSL command line tool, and set the global certificate authority list in your FGLPROFILE file.

**Important:** On GMI mobile devices the iOS SSL/TLS layer is used for HTTPS, and the device Keychain typically holds the server certificate authority list. Therefore, the FGLPROFILE file security settings are ignored with the exception of the following: security.global.ca, security.global.ca.lookuppath, and security.global.systemca.

1. Create the client's certificate authority list.
   a) Save the certificates of the HTTPS server to disk.
      Type the server's URL in your browser. When prompted, save all the certificates from the Certificate Hierarchy. For more information see, Selecting the certificate to add on page 3317 and Missing certificates on page 3311.
   b) Create the client's Certificate Authority List by running the following command for each of the certificates that you saved to disk.

      ```bash
      $ openssl x509 -in ServerCertificate.crt -text >> ClientCAList.pem
      ```

      **Note:** All trusted certificate authorities are listed. These are checked following a chain of child to parent certificates until a certificate is reached which is trusted. All other certificates that were trusted by the Root Certificate Authority will also be considered as trusted by the client. For more information see Certificate authorities on page 3301.

2. Set the global certificate authority list in your FGLPROFILE file.

   The global certificate authority list entry defines the file containing the certificate authority list used by the Genero Web Services client to validate all certificates coming from the different servers it will connect to.

   ```
   security.global.ca = "ClientCAList.pem"
   ```

   **Note:** The file is located based on the current execution directory. If you use Genero Studio, for instance, fglrun may not be executed in the same directory as when you use the command line. The recommended practice therefore is to specify an absolute path for the .pem, for example:

   ```
   security.global.ca = "/opt/usr/certs/ClientCAList.pem"
   ```

   If security.global.ca is not defined, Genero Web Services will look to see whether the operating system has a keystore, otherwise security.global.ca.lookuppath will be used. For further information, see Certificate authorities in HTTPS configuration on page 3305.
The client application is configured to use the appropriate certificate authority list to validate a server's certificate.

**What to do next**

In your FGLPROFILE file ensure there are configuration entries (ws.*) for the HTTPS server URL and for the HTTP authentication when accessing the HTTPS server. See *Add configuration entries in your FGLPROFILE file* on page 3309.

**Related concepts**

[Error: Peer certificate is issued by a company not in our CA list](#) on page 3310

When a client connects to a server using HTTPS, the client needs to trust the server it is in communication with. So the client needs to add the server's CAs (certificate authorities lists) to its trusted CAs.

[Enable OCSP](#) on page 3310

To enable OCSP (Online Certificate Status Protocol), set the `security.global.ocsp.enable` and `security.global.ocsp.url` entries in FGLPROFILE.

[FGLPROFILE entries for Web Services](#) on page 3489

The FGLPROFILE entries relating to Genero Web Services are divided between five categories: security, basic or digest HTTP authentication, proxy configuration, server configuration, and XML cryptography.

[FGLPROFILE: HTTP(S) Proxy Authentication](#) on page 3496

FGLPROFILE entries can be used to define a connection to an HTTPS server via a proxy, and with HTTP and Proxy Authentication.

[Accessing secured services](#) on page 3303

Security and authentication are important. Genero Web Services provides various communications options for a client to connect to a Web service.

**Add configuration entries in your FGLPROFILE file**

Add a set of configuration entries that specify the security configuration for GWS clients accessing an HTTPS server.

In this task you add configuration entries (ws.*) in your FGLPROFILE file for the HTTPS server URL and for HTTP authentication when accessing the HTTPS server. For an example, see *FGLPROFILE: HTTP(S) Proxy Authentication* on page 3496.

**Important:** On GMI mobile devices the iOS SSL/TLS layer is used for HTTPS, and the device Keychain typically holds the server certificate authority list. Therefore, the FGLPROFILE file security settings are ignored with the exception of the following: `security.global.ca`, `security.global.ca.lookuppath`, and `security.global.systemca`.

Add configuration entries for the server.

The following entries must be defined with an unique identifier (such as "myserver"):

   
   (Line breaks added for document readability)

b) `ws.myserver.security = "id1"

   Where the value (id1 in this example) must match the unique identifier defined by the client security entry created in *Configure for the client certificate* on page 3307.

**Tip:** The unique identifier "myserver" can be used in the BDL client code in place of the actual URL.

**Related concepts**

[FGLPROFILE entries for Web Services](#) on page 3489

The FGLPROFILE entries relating to Genero Web Services are divided between five categories: security, basic or digest HTTP authentication, proxy configuration, server configuration, and XML cryptography.

[Accessing secured services](#) on page 3303

Security and authentication are important. Genero Web Services provides various communications options for a client to connect to a Web service.

[Troubleshoot certificate issues](#) on page 3310
You may encounter known (and common) issues when completing the Genero Web Services tutorials or when adding Web services of your own. These issues and their solutions are presented in the following topics.

**Configure a WS client to connect via an HTTP Proxy**

Configuration steps to connect via a HTTP proxy.

**Important:** On GMI mobile devices, FGLPROFILE settings are ignored: The device configuration for proxy will always be used.

1. Add the location of the proxy to your FGLPROFILE file with the `proxy.http.location` entry.
   
   Add the entry `proxy.http.location` to your fglprofile. For the value, provide the IP address of the HTTP proxy and the port number where the HTTP proxy is listening, separated by a colon. For example, to have a client connect via a HTTP proxy located at the IP address "10.0.0.170" and listening on port number "8080", add this entry to your fglprofile:
   
   ```
   proxy.http.location = "10.0.0.170:8080"
   ```
   
   **Note:** To configure the client to connect via an HTTPS proxy, replace `http` with `https`.

2. Define the list of host names the client will not have to connect to via a proxy with the `proxy.http.list` entry.
   
   Add the entry `proxy.http.list` to your FGLPROFILE file. For the value, provide a semi-colon separated list of clients. For example, to exclude all hosts beginning with "www.mycompany.com" or "www.google." from connecting via a HTTP proxy, add this entry to your fglprofile:
   
   ```
   proxy.http.list = "www.mycompany.com;www.google."
   ```

**Related reference**

[Proxy configuration](#) on page 3492

**Configure a WS client to use IPv6**

Configuration steps to customize IPv6 for a WS client.

A Web Services client program can access a WS server using IPv6.

URLs that map to IPv6 addresses will be automatically handled by the Web Services library. It is also possible to specify an IPv6 address directly as URL in your BDL code by enclosing the address in [] square brackets, for example:

```
LET myURL = "http://[fe80::20c:29ff:fe05:9ca3]:80/index.html"
```

By default, the WS library will automatically use IPv6 addresses if available, and fallback to IPv4 if not. To overcome the default behavior, you can explicitly specify the IP version.

The platform where WS client programs execute must support IPv6.

1. If needed, force the IP version with the `ip.global.version` entry in your fglprofile file, by specifying "4" for IPv4 or "6" for IPv6.
   
   For example, to force IPv4 (when IPv6 is available):
   
   ```
   ip.global.version = "4"
   ```

2. When using IPv6 for link-local addresses, if several network interfaces exist on the machine, you can explicitly specify which interface must be used with the `ip.global.v6.interface.name` or `ip.global.v6.interface.id` entry in fglprofile.

   In order to specify the IPv6 network interface by name, use:
   
   ```
   ip.global.v6.interface.name = "eth0"
   ```

   **Important:** The `ip.global.v6.interface.name` entry is not supported on Microsoft™ Windows® platforms.

   In order to specify the IPv6 network interface by id, use:
Authenticate the WS client to a server (HTTP basic authentication)

Configuration steps to authenticate the client to a server.

See Basic or digest HTTP authentication on page 3491. For an example, see FGLPROFILE: HTTP(S) Proxy Authentication on page 3496.

Important: On GMI mobile devices, FGLPROFILE settings are ignored: Use the com.HTTPRequest.setAuthentication on page 2883 API instead.

1. Add HTTP authenticate entries to your FGLPROFILE file.

   To connect to a server with HTTP Authentication, define the client login and password with the same values as registered on the server side. These entries must be defined with an unique identifier (httpauth in this example) to define a HTTP Authentication with "mylogin" as login and "mypassword" as password:

   ```
   authenticate.httpauth.login = "mylogin"
   authenticate.httpauth.password = "mypassword"
   ```

   See [RFC2617] for more details.

2. Encrypt the password.

   Due to security leaks, it is recommended that you NOT have a password in clear text. The Genero Web Services package provides the tool fglpass, which encrypts a password with a certificate that is readable only with the associated private key. To encrypt the HTTP authentication password:

   a) Encrypt the clear text password with fglpass using the client certificate.

   ```
   $ fglpass -e -c MyClient.crt
   Enter password :mypassword
   ```

   Note: fglpass outputs the encrypted password on the console but can be redirected to a file.

   b) Modify the HTTP authentication password entry by specifying the security configuration to use to decrypt it ("id1" in our example)

   ```
   authenticate.httpauth.password.id1="HWTFu8QE2t3e5D4joy7js8mB95oOGTzLmcAor9j5DS+C
   loi1iGCwzV9eWpfmIWSON9IwoiJheYxfnu20uaGGmmiUGiHxT6341ePXNSsicu32Nt1Vp9t6RcSO
   wN/p9a6D4XtiD9iHW7iQvXhqC9uamd3gI9Q3GhWxOMM1Y/c8Y="
   ```

   Note: Hard returns have been added to the code sample above, for the purpose of printing and viewing within this document. The value for authenticate.httpauth.password.id1 is a single string with no spaces.

   Note: The size of the encrypted password depends on the size of the public key, and can change based on the certificate used to encrypt it.

3. Configure the client to authenticate to a server.

   As a client is able to connect to different servers that do not know the client with the same login and password, it is necessary to specify the login and password that corresponds to each server. To authenticate the client known as "myclient" with the password passphrase by the server myserver, add the following entry:

   ```
   ws.myserver.authenticate = "httpauth"
   ```

Related tasks

Configure a WS client to access an HTTPS server on page 3305
Configuration steps to access a server in HTTPS.

**Authenticate the WS client to a proxy**
Configuration steps to authenticate the client to a proxy (proxy authentication).

For an example, see FGLPROFILE: HTTP(S) Proxy Authentication on page 3496.

**Important:** On GMI mobile devices, FGLPROFILE settings are ignored: The device configuration for proxy will always be used.

1. Add an HTTP authenticate entry to your FGLPROFILE file.
   To connect via a proxy with HTTP Proxy Authentication, it is necessary to define the client login and password as registered on the HTTP proxy.
   
   The following two entries must be defined with an unique identifier (proxyauth for our example) to define a HTTP Proxy Authentication with `myapplication` as login and `mypassword` as password:
   ```plaintext
   authenticate.proxyauth.login = "myapplication"
   authenticate.proxyauth.password = "mypassword"
   ```
   
   See [RFC2617](https://www.ietf.org/rfc/rfc2617.txt) for more details.

2. For proxy authentication, an entry must be made to the HTTP proxy configuration in order to authenticate a client.
   
   To authenticate a client known as `myapplication` with `mypassword` as password by the HTTP Proxy, add the following entry to the HTTP proxy configuration:
   ```plaintext
   proxy.http.authenticate = "proxyauth"
   ```
   
   **Note:** To authenticate the client to a HTTPS proxy, replace `http` with `https`.

**Related reference**
Proxy configuration on page 3492

**Writing a Web Services server application**
To create a RESTful Genero Web services server application, there are a minimum of five steps that the server application must handle.

In discussing the five steps, the Calculator RESTful Web services server application is used to provide code examples.

**Step 1: Import extension packages**
Extension packages include classes and functions necessary for your Web services server application. Use the `IMPORT` statement to include these packages.

Include the following lines at the top of your module as instructions to import the required libraries:

```plaintext
IMPORT com
IMPORT util
```

Methods to create a REST Web Service server application are contained in the classes that make up the com library of the Genero Web Services (GWS).

The calculator demo application also uses methods from the util.JSON class for data exchange.

In the next step we start the server and listen for incoming requests, Step 2: Listen for requests on page 3430

**Step 2: Listen for requests**
Listen for an incoming request.

Define an object variable using the com.HTTPServiceRequest class identifier. This variable will reference the incoming HTTP service request. In addition, define an INTEGER variable to hold an error status, if needed.

```plaintext
DEFINE ret INTEGER
DEFINE req com.HTTPServiceRequest
```
Start the server engine with the method `com.WebServiceEngine.Start()`.

```plaintext
DEFER INTERRUPT

# Start the server
DISPLAY "Starting server..."

# Starts the server on the port number specified by the FGLAPPSERVER environment variable
# (EX: FGLAPPSERVER=8090)
CALL com.WebServiceEngine.Start()
DISPLAY "The server is listening."
```

Listen for an incoming request using `com.WebServiceEngine.getHTTPServiceRequest`. When a request is received, the variable `req` references the HTTP service request object. You can use the same object to send a response.

Include a `CATCH` block to trap runtime exceptions. For a RESTful Web service, include a check for error -15565, which is returned when an incoming request cannot be returned.

```plaintext
... # create the server
WHILE TRUE
    TRY
        LET req = com.WebServiceEngine.getHTTPServiceRequest(-1)
        ... parse and process the request ...
    CATCH
        LET ret = STATUS
        CASE ret
            WHEN -15565
                DISPLAY "Disconnected from application server."
                EXIT WHILE
            OTHERWISE
                DISPLAY "[ERROR] " || ret
                EXIT WHILE
        END CASE
    END TRY
END WHILE
```

For more information see `com.WebServiceEngine methods` on page 2829.

In the next step we extract the details of the incoming request. **Step 3: Parse the request** on page 3431

**Step 3: Parse the request**

Using methods of the `com.HTTPServiceRequest` class (among others), parse the details of the request.

An HTTP service request has been received and stored in an instance of a `com.HTTPServiceRequest` object.

To use the details of the request, the next step is to define a record to hold the pieces of information that come in with a request and to parse the request into the record. The demo creates a user-defined type `reqInfoTyp` to reference when defining the record.

**Note:** This topic details how the calculator demo server application parses the instance of the `HTTPServiceRequest` object. While your specific HTTP service requests may differ in the details, the parsing needs will be similar.

```plaintext
TYPE reqInfoTyp RECORD
    method STRING,
    ctype STRING,  # check the Content-Type
    informat STRING,  # short word for Content Type
    caccept STRING,  # check which format the client accepts
```
Define a variable based on the type.

```
DEFINE m_reqInfo reqInfoTyp
```

Call a function that takes the variable referencing the instance of the com.HTTPServiceRequest object as input. The purpose of the function will be to parse the HTTP service request into its components.

```
CALL getReqInfo(req)
```

**The getReqInfo function**

This function uses both the HTTPServiceRequest and StringTokenizer classes and methods to parse the request into the reqInfoTyp record.

```
FUNCTION getReqInfo(req)
```

Define the variables needed. The variable `req` references the instance of the HTTP service request object. The variables `str`, `val`, `token`, and `i` are used when parsing out details with the StringTokenizer classes and methods.

```
DEFINE req com.HTTPServiceRequest
DEFINE str, val base.StringTokenizer
DEFINE token STRING
DEFINE i INT
```

For the remainder of this topic, you are retrieving data from the instance of the HTTP service request object (referenced by the variable `req`) and populating the record variable designed to hold this data (`m_reqInfo`).

Initialize `m_reqInfo` to NULL. This ensures that there are no pre-existing values in the record variable.

**Note:** The variable `m_reqInfo` record was defined at the top of the module, and therefore can be used in this function.

```
INITIALIZE m_reqInfo TO NULL
```

Retrieve the value of the Content-Type request header. The Content-Type request header defines the format of the incoming message body. REST APIs commonly use "application/json" or "application/xml" to reveal the format of the message body. The server application anticipates the Content-Type to be either JSON or XML. The custom `getHeaderByName` function retrieves the value of the Content-Type header, and based on the return value, populates the input format variable (`m_reqInfo.informat`) with either "XML" or "JSON".

```
LET m_reqInfo.ctype = getHeaderByName(req,"Content-Type")
IF m_reqInfo.ctype.getIndexOf("/xml",1) THEN
  LET m_reqInfo.informat = "XML"
ELSE
  LET m_reqInfo.informat = "JSON"
END IF
```

Retrieve the value for Accept request header. The Accept request header is where the client application conveys its response preference. The server application expects the Accept request header to be either JSON or XML. The
custom `getHeaderByName` function retrieves the value of the Accept header, and based on the returned value, populates the output format variable `m_reqInfo.outformat` with either "XML" or "JSON".

```plaintext
LET m_reqInfo.caccept = getHeaderByName(req,"Accept")
IF m_reqInfo.caccept.indexOf("/xml",1) THEN
  LET m_reqInfo.outformat = "XML"
ELSE
  LET m_reqInfo.outformat = "JSON"
END IF
```

Parse out the HTTP method (verb) of the HTTP service request with the function `com.HTTPServiceRequest.getMethod()`, and populate the method variable `m_reqInfo.method` with the result.

```plaintext
LET m_reqInfo.method = req.getMethod()
```

While possible values could include GET, POST, PUT, HEAD, and DELETE, our demo server application expects GET, and the processing part of the server application code does not test or code for any verbs aside from GET.

Parse out the path and the query. Retrieve the URL resource using the function `com.HTTPServiceRequest.getUrl()`. To parse out the path and the query from the URL, call the custom `parseUrl` function. The path is placed in the path variable `m_reqInfo.path` and the query string is placed in the query variable `m_reqInfo.query`.

```plaintext
CALL parseUrl(req.getUrl()) RETURNING m_reqInfo.path, m_reqInfo.query
```

Take the query variable `m_reqInfo.query` and parse out the value pairs using the `StringTokenizer` class and methods. The value pairs of variable name and value are stored in the `m_reqInfo.items` dynamic array.

```plaintext
LET str = base.StringTokenizer.create(m_reqInfo.query,"&")
LET i=1
CALL m_reqInfo.items.clear()
WHILE str.hasMoreTokens()
  LET token = str.nextToken()
  LET val = base.StringTokenizer.create(token,"=")
  IF val.hasMoreTokens() THEN LET m_reqInfo.items[i].name = val.nextToken() END IF
  IF val.hasMoreTokens() THEN LET m_reqInfo.items[i].value = val.nextToken() END IF
  LET i=i+1
END WHILE
```

By the end of the custom `getReqInfo()` function, the HTTP service request object is parsed into the variables that comprise a `reqInfoType` record. The application can now access the values it needs to process the request.

**getHeaderByName function**

The purpose of this function is to take as input two things:

- The HTTP service request object in a `com.HTTPServiceRequest` variable.
- The name of the header type whose value you want to return.

In the demo application, this function is used to parse out the value for the Content-Type header request and the Accept header request.

To do this, the function uses the API methods `getRequestHeaderCount`, `getRequestHeaderName`, and `getRequestHeaderValue` against the `com.HTTPServiceRequest` object. The function first counts the number of header requests, and then cycles through them until it finds the specific header request that is being asked for. When the match happens, the value for that header is returned; if no match is found, then NULL is returned.

```plaintext
FUNCTION getHeaderByName(areq,hname)
```
DEFINE areq com.HTTPServiceRequest
DEFINE hname STRING

DEFINE aname STRING
DEFINE iname STRING
DEFINE i INT
DEFINE n INT

LET aname = hname.toLowerCase()
LET n = areq.getRequestHeaderCount()
FOR i=1 TO n
    LET iname = areq.getRequestHeaderName(i)
    IF aname.equals(iname.toLowerCase()) THEN
        RETURN areq.getRequestHeaderValue(i)
    END IF
END FOR

RETURN NULL
END FUNCTION

parseUrl function

For this function, the URL is passed in. This function divides the URL into two parts, the path (up to and including
the question mark) and the query. The path and the query are then returned to the calling function.

FUNCTION parseUrl(url)
    DEFINE url STRING
    DEFINE i INT

    LET i = url.getIndexOf("?",1)
    IF i = 0 THEN
        RETURN url, NULL
    ELSE
        RETURN url.substring(1,i), url.substring(i+1,url.getLength())
    END IF
END FUNCTION

For example, the URL being passed in by the demo client application is "http://localhost:8090/add?a=1&b=2". The
function then returns the path ("http://localhost:8090/add?") and the query ("a=1&b=2").

For more information see com.WebServiceEngine methods on page 2829

In the next step we process the incoming request, Step 4: Process the request on page 3434

Step 4: Process the request

Having parsed the HTTPServiceRequest object into its parts, you can now process the request.

In the previous step, the m_reqInfo record was populated with the values of the HTTPServiceRequest object.
The application now needs to work with these values.

Each Web services server application will be different in the type of processing that is done. For completeness,
this topic will highlight some of the processing that is happening in the calculator demo server application.

Despite having variables in the m_reqInfo record, four additional STRING variables are defined to hold the
method, the URL, the query, and the output format. In addition, an INTEGER variable is defined to hold an index
value, used by some of the custom processing functions.

DEFINE method STRING
DEFINE url STRING
DEFINE qry STRING
DEFINE acc STRING
DEFINE idx INT
With the record populated, the server application then populates the additional variables.

```plaintext
# Get the type of method
LET method = m_reqInfo.method
# Get the request path
LET url = m_reqInfo.path
# Get the query string
LET qry = m_reqInfo.query
# Get the output format
LET acc = m_reqInfo.outformat
```

In the demo server application, the `CASE` statement only tests for the GET method.

```plaintext
CASE method
  WHEN "GET"
    ... processing instructions
  OTHERWISE
    CALL setError("Unknown request:
      || url || \
      "
      || method)
    LET err.code = -3
    LET err.desc = ERR_METHOD
    CALL req.sendTextResponse(200,"OK",util.JSON.stringify(err))
END CASE
```

For the remainder of the processing code, the demo server application:

- Determines whether the request is for addition, subtraction, multiplication, or division
- Retrieves the values for the variables a and b
- Handles errors appropriately

For example, the source code that would process an addition request:

```plaintext
IF url.indexOf("/add?",1) > 0 THEN
  LET idx = getParameterIndex("a")
  IF idx = 0 THEN
    LET add_out.status.code = -1
    LET add_out.status.desc = ERR_PARAM_A
    CALL req.setResponseHeader("Content-Type","application/json")
    CALL req.sendTextResponse(200, "OK", util.JSON.stringify(add_out))
    EXIT CASE
  ELSE
    LET add_in.a = getParameterValue(idx)
  END IF
  LET idx = getParameterIndex("b")
  IF idx = 0 THEN
    LET add_out.status.code = -1
    LET add_out.status.desc = ERR_PARAM_B
    CALL req.setResponseHeader("Content-Type","application/json")
    CALL req.sendTextResponse(200, "OK", util.JSON.stringify(add_out))
    EXIT CASE
  ELSE
    LET add_in.b = getParameterValue(idx)
  END IF
  CALL add()
  ... send response
END IF
```

In the next step the server sends the response back to the client, **Step 5: Send response** on page 3435.

**Step 5: Send response**

Having completed the processing, the server sends the response back to the client.

The server application has the value or values that must be returned to the client. It now needs to format and send the response.
In the demo server application, the `setResponseHeader()` method sets the Content-Type to JSON.

```java
CALL req.setResponseHeader("Content-Type","application/json")
```

The `sendTextResponse()` sends three things: a code, a description, and the response data. In this case, the data is formatted as JSON using the `util.JSON.stringify()` method, which transforms a record variable into a flat JSON formatted string.

```java
CALL req.sendTextResponse(200, "OK", util.JSON.stringify(add_out))
```

**Related concepts**

**Step 6: Provide information about your service**

You must provide information about the services offered by your application to the developers of the Web services client applications that will interact with your server.

**Step 6: Provide information about your service**

You must provide information about the services offered by your application to the developers of the Web services client applications that will interact with your server.

Information that the client application developer will need to know:

- The names of the resources or functions, and the query parameters and arguments to use with them.
- What media type the server expects for receiving, and then delivering, the message body (such as "JSON" or "XML").
- The HTTP method the server is expecting (such as "GET")

See also [Step 1: Obtain information about Web service resources](#) on page 3419.

**The RESTful calculator demo source**

This calculator demo provides a sample RESTful Web services, demonstrating how to create a server and client application using low-level APIs.

These topics provide the full code of the calculator demo application, and compares how the server and client functionality relates.

**Calculator server source**

The source code for the server-side application included in the RESTful Web services calculator demo.

```java
IMPORT com
IMPORT util
IMPORT FGL WSHelper

TYPE TYP_status RECORD
    code INTEGER,
    desc STRING
END RECORD

GLOBALS
DEFINE
    add_in RECORD
        a INTEGER,
        b INTEGER
    END RECORD,
    add_out RECORD
        status TYP_status,
        r INTEGER
    END RECORD

DEFINE
    substract_in RECORD
        a INTEGER,
        b INTEGER
```

| Web services | 3436 |
END RECORD,
  substract_out RECORD
    status TYP_status,
    r INTEGER
END RECORD

DEFINE
  multiply_in RECORD
    a INTEGER,
    b INTEGER
END RECORD,
  multiply_out RECORD
    status TYP_status,
    r INTEGER
END RECORD

DEFINE
  divide_in RECORD
    a INTEGER,
    b INTEGER
END RECORD,
  divide_out RECORD
    status TYP_status,
    quotient INTEGER,
    remainder INTEGER
END RECORD

DEFINE
  err TYP_status
END GLOBALS

TYPE reqInfoTyp RECORD
  method STRING,
  ctype STRING,  # check the Content-Type
  informat STRING,  # short word for Content Type
  caccept STRING,  # check which format the client accepts
  outformat STRING,  # short word for Accept
  path STRING,
  items WSHelper.WSQueryType
END RECORD

DEFINE m_reqInfo reqInfoTyp

CONSTANT ERR_PARAM_A = "Operand 'a' not found"
CONSTANT ERR_PARAM_B = "Operand 'b' not found"
CONSTANT ERR_OPERATION = "Operation not found"
CONSTANT ERR_METHOD = "Method not supported"

MAIN
  DEFINE ret INTEGER
  DEFINE req com.HTTPServiceRequest
  DEFINE method STRING
  DEFINE url STRING
  DEFINE acc STRING
  DEFINE idx INT
  
  DEFER INTERRUPT
  
  # Start the server
  # DISPLAY "Starting server..."
  #
# Starts the server on the port number specified by the FGLAPPSERVER environment variable
#  (EX: FGLAPPSERVER=8090)

CALL com.WebServiceEngine.Start()
DISPLAY "The server is listening."

# create the server
WHILE TRUE
TRY
LET req = com.WebServiceEngine.getHTTPServiceRequest(-1)
CALL getReqInfo(req)
# Get the type of method
LET method = m_reqInfo.method
# Get the request path
LET url = m_reqInfo.path
# Get the output format
LET acc = m_reqInfo.outformat

DISPLAY "Processing request... ", method, " ", url

# parse the url, retrieve the operation and the operand
CASE method
WHEN "GET"
IF url.indexOf("/add",1) > 0 THEN
LET idx = getParameterIndex("a")
IF idx = 0 THEN
LET add_out.status.code = -1
LET add_out.status.desc = ERR_PARAM_A
CALL req.setResponseHeader("Content-Type","application/json")
CALL req.sendTextResponse(200, "OK",
util.JSON.stringify(add_out))
EXIT CASE
ELSE
LET add_in.a = getParameterValue(idx)
END IF
LET idx = getParameterIndex("b")
IF idx = 0 THEN
LET add_out.status.code = -1
LET add_out.status.desc = ERR_PARAM_B
CALL req.setResponseHeader("Content-Type","application/json")
CALL req.sendTextResponse(200, "OK",
util.JSON.stringify(add_out))
EXIT CASE
ELSE
LET add_in.b = getParameterValue(idx)
END IF
CALL add()
CALL req.setResponseHeader("Content-Type","application/json")
CALL req.sendTextResponse(200, "OK",
util.JSON.stringify(add_out))
ELSE IF url.indexOf("/subtract",1) > 0 THEN
LET idx = getParameterIndex("a")
IF idx = 0 THEN
LET add_out.status.code = -1
LET add_out.status.desc = ERR_PARAM_A
CALL req.setResponseHeader("Content-Type","application/json")
CALL req.sendTextResponse(200, "OK",
util.JSON.stringify(subtract_out))
ELSE
LET idx = getParameterIndex("b")
IF idx = 0 THEN
LET add_out.status.code = -1
LET add_out.status.desc = ERR_PARAM_B
CALL req.setResponseHeader("Content-Type","application/json")
CALL req.sendTextResponse(200, "OK",
util.JSON.stringify(subtract_out))
END IF
END IF

END CASE
END TRY
EXIT CASE
ELSE
    LET substract_in.a = getParameterValue(idx)
END IF
    LET idx = getParameterIndex("b")
    IF idx = 0 THEN
        LET add_out.status.code = -1
        LET add_out.status.desc = ERR_PARAM_B
        CALL req.setResponseHeader("Content-Type","application/json")
        CALL req.sendTextResponse(200, "OK",
        util.JSON.stringify(substract_out))
        EXIT CASE
    ELSE
        LET substract_in.b = getParameterValue(idx)
    END IF
CALL substract()
CALL req.setResponseHeader("Content-Type","application/json")
CALL req.sendTextResponse(200, "OK",
    util.JSON.stringify(substract_out))
ELSE IF url.getIndexOf("/multiply",1) > 0 THEN
    LET idx = getParameterIndex("a")
    IF idx = 0 THEN
        LET add_out.status.code = -1
        LET add_out.status.desc = ERR_PARAM_A
        CALL req.setResponseHeader("Content-Type","application/json")
        CALL req.sendTextResponse(200, "OK",
        util.JSON.stringify(multiply_out))
        EXIT CASE
    ELSE
        LET multiply_in.a = getParameterValue(idx)
    END IF
    LET idx = getParameterIndex("b")
    IF idx = 0 THEN
        LET add_out.status.code = -1
        LET add_out.status.desc = ERR_PARAM_B
        CALL req.setResponseHeader("Content-Type","application/json")
        CALL req.sendTextResponse(200, "OK",
        util.JSON.stringify(multiply_out))
        EXIT CASE
    ELSE
        LET multiply_in.b = getParameterValue(idx)
    END IF
CALL multiply()
CALL req.setResponseHeader("Content-Type","application/json")
CALL req.sendTextResponse(200, "OK",
    util.JSON.stringify(multiply_out))
ELSE IF url.getIndexOf("/divide",1) > 0 THEN
    LET idx = getParameterIndex("a")
    IF idx = 0 THEN
        LET add_out.status.code = -1
        LET add_out.status.desc = ERR_PARAM_A
        CALL req.setResponseHeader("Content-Type","application/json")
        CALL req.sendTextResponse(200, "OK",
        util.JSON.stringify(divide_out))
        EXIT CASE
    ELSE
        LET multiply_in.a = getParameterValue(idx)
    END IF
CALL multiply()
CALL req.setResponseHeader("Content-Type","application/json")
CALL req.sendTextResponse(200, "OK",
    util.JSON.stringify(multiply_out))
ELSE
LET divide_in.a = getParameterValue(idx)
END IF
LET idx = getParameterIndex("b")
IF idx = 0 THEN
  LET add_out.status.code = -1
  LET add_out.status.desc = ERR_PARAM_B
  CALL req.setResponseHeader("Content-Type","application/json")
  CALL req.sendTextResponse(200, "OK",
  util.JSON.stringify(divide_out))
ELSE
  LET divide_in.b = getParameterValue(idx)
END IF
CALL divide()
CALL req.setResponseHeader("Content-Type","application/json")
CALL req.sendTextResponse(200, "OK",
util.JSON.stringify(divide_out))
ELSE
  CALL setError("Unknown request:\n" || url || "\n" || method)
  LET err.code = -2
  LET err.desc = ERR_OPERATION
  CALL req.sendTextResponse(200, "OK", util.JSON.stringify(err))
END IF
END IF
END IF
END IF
OTHERWISE
  CALL setError("Unknown request:\n" || url || "\n" || method)
  LET err.code = -3
  LET err.desc = ERR_METHOD
  CALL req.sendTextResponse(200, "OK", util.JSON.stringify(err))
END CASE
IF int_flag<>0 THEN
  LET int_flag=0
  EXIT WHILE
END IF
CATCH
  LET ret = STATUS
  CASE ret
  WHEN -15565
    DISPLAY "Disconnected from application server."
    EXIT WHILE
  OTHERWISE
    DISPLAY "[ERROR] " || ret
    EXIT WHILE
  END CASE
END TRY
END WHILE
END MAIN
FUNCTION add()
  LET add_out.r = add_in.a + add_in.b
  LET add_out.status.code = 0
  LET add_out.status.desc = "OK"
END FUNCTION
FUNCTION substract()
  LET substract_out.r = substract_in.a - substract_in.b
  LET substract_out.status.code = 0
LET substract_out.status.desc = "OK"
END FUNCTION

FUNCTION multiply()
  LET multiply_out.r = multiply_in.a * multiply_in.b
END FUNCTION

FUNCTION divide()
  IF divide_in.b != 0 THEN
    LET divide_out.quotient = divide_in.a / divide_in.b
    LET divide_out.remainder = divide_in.a MOD divide_in.b
    LET divide_out.status.code = 0
    LET divide_out.status.desc = "OK"
  ELSE
    LET divide_out.status.code = 0
    LET divide_out.status.desc = "Cannot divide by 0"
  END IF
END FUNCTION

FUNCTION getHeaderByName(areq,hname)
  DEFINE areq com.HTTPServiceRequest
  DEFINE hname STRING

  DEFINE aname STRING
  DEFINE iname STRING
  DEFINE i INT
  DEFINE n INT

  LET aname = hname.toLowerCase()
  LET n = areq.getRequestHeaderCount()
  FOR i=1 TO n
    LET iname = areq.getRequestHeaderName(i)
    IF aname.equals(iname.toLowerCase()) THEN
      RETURN areq.getRequestHeaderValue(i)
    END IF
  END FOR

  RETURN NULL
END FUNCTION

FUNCTION getReqInfo(req)
  DEFINE req com.HTTPServiceRequest

  INITIALIZE m_reqInfo TO NULL

  LET m_reqInfo.ctype = getHeaderByName(req,"Content-Type")
  IF m_reqInfo.ctype.getIndexOf("/xml",1) THEN
    LET m_reqInfo.informat = "XML"
  ELSE
    LET m_reqInfo.informat = "JSON"
  END IF

  LET m_reqInfo.caccept = getHeaderByName(req,"Accept")
  IF m_reqInfo.caccept.getIndexOf("/xml",1) THEN
    LET m_reqInfo.outformat = "XML"
  ELSE
    LET m_reqInfo.outformat = "JSON"
  END IF

  LET m_reqInfo.method = req.getMethod()

  LET m_reqInfo.path = req.getURLPath()

  CALL req.getURLQuery(m_reqInfo.items)
FUNCTION setError(s)
    DEFINE s STRING
    DISPLAY s
END FUNCTION

# returns 0 if element not found
FUNCTION getParameterIndex(s)
    DEFINE s STRING
    DEFINE i INT

    FOR i=1 TO m_reqInfo.items.getLength()
        IF s.equals(m_reqInfo.items[i].name) THEN
            RETURN i
        END IF
    END FOR

    RETURN 0
END FUNCTION

FUNCTION getParameterValue(i)
    DEFINE i INT

    RETURN m_reqInfo.items[i].value
END FUNCTION

Calculator client source
The source code for the client-side application included in the RESTful Web services calculator demo.

IMPORT com
IMPORT xml
IMPORT util

TYPE TYP_status RECORD
    code INTEGER,
    desc STRING
END RECORD

PUBLIC DEFINE info RECORD
    url STRING,
    verb STRING,
    reqtype STRING,
    request STRING,
    status STRING,
    resptype STRING,
    response STRING,
    result RECORD
        code INT,
        desc STRING
    END RECORD
END RECORD

DEFINE
    add_in RECORD
        a INTEGER,
        b INTEGER
    END RECORD,
    add_out RECORD
        status TYP_status,
        r INTEGER
    END RECORD
MAIN
  DEFINE req com.HttpRequest
  DEFINE resp com.HttpResponse
  DEFINE doc xml.DomDocument
  DEFINE node xml.DomNode
  TRY
    LET add_in.a = 1
    LET add_in.b = 2
    add_in.a || "&b=" || add_in.b)
    CALL req.setMethod("GET")
    CALL req.setHeader("Content-Type", "application/json")
    CALL req.setHeader("Accept", "application/json")
    CALL req.doRequest()
    LET resp = req.getResponse()
    LET info.status = resp.getStatusCode()
    IF info.status = 200 THEN
      LET info.resptype = resp.getHeader("Content-Type")
      IF info.resptype.getIndexOf("/xml",1) THEN
        LET doc = resp.getXmlResponse()
        LET node = doc.getDocumentElement()
        CALL xml.Serializer.DomToVariable(node, add_out)
        LET info.response = node.toString()
      ELSE
        LET info.response = resp.getTextResponse()
        CALL util.JSON.parse(info.response, add_out)
      END IF
    ELSE
      LET info.response = SFMT("[%1] %2", resp.getStatusCode(),
      resp.getStatusDescription())
    END IF
    CATCH
      LET info.result.code = status
      LET info.result.desc = sqlca.sqlerrm
  END TRY
END MAIN

Comparing the client to the server
Comparing the code between the RESTful calculator client and server.

The purpose of this side-by-side comparison is to provide a glimpse of how the client code and the server code relate. In some cases, only the initial line of server code has been provided; you can look at the full source for either the client or the server for the complete code.

• Calculator client source on page 3442
• Calculator server source on page 3436
Table 719: Client versus Server Code Comparison

<table>
<thead>
<tr>
<th>Calculator client</th>
<th>Calculator server</th>
</tr>
</thead>
<tbody>
<tr>
<td>LET req = com.HttpRequest.Create(&quot;<a href="http://localhost:8090/add?a=">http://localhost:8090/add?a=</a>&quot;</td>
<td></td>
</tr>
<tr>
<td>The client creates the HTTP request, which is a URL consisting of a path and parameters.</td>
<td>The server parses out the path and the query from the HTTP service request.</td>
</tr>
<tr>
<td>CALL req.setMethod(&quot;GET&quot;)</td>
<td>LET m_reqInfo.method = req.getMethod()</td>
</tr>
<tr>
<td>The client sets the method with the verb &quot;GET&quot;.</td>
<td>The server parses out the method verb from the HTTP service request.</td>
</tr>
<tr>
<td>CALL req.setHeader(&quot;Content-Type&quot;, &quot;application/json&quot;)</td>
<td>LET m_reqInfo.ctype = getHeaderByName(req,&quot;Content-Type&quot;) IF m_reqInfo.ctype.getIndexOf(&quot;/xml&quot;,1) THEN LET m_reqInfo.informat = &quot;XML&quot; ELSE LET m_reqInfo.informat = &quot;JSON&quot; END IF</td>
</tr>
<tr>
<td>The client sets the Content-Type header request.</td>
<td>The server parses out the value of the Content-Type header request.</td>
</tr>
<tr>
<td>CALL req.setHeader(&quot;Accept&quot;, &quot;application/json&quot;)</td>
<td>LET m_reqInfo.caccept = getHeaderByName(req,&quot;Accept&quot;) IF m_reqInfo.caccept.getIndexOf(&quot;/xml&quot;,1) THEN LET m_reqInfo.outformat = &quot;XML&quot; ELSE LET m_reqInfo.outformat = &quot;JSON&quot; END IF</td>
</tr>
<tr>
<td>The client sets the Accept header request.</td>
<td>The server parses out the value of the Accept header request.</td>
</tr>
<tr>
<td>CALL req.doRequest()</td>
<td>LET req = com.WebServiceEngine.getHTTPServiceRequest(-1)</td>
</tr>
<tr>
<td>The client submits the request.</td>
<td>The server receives the HTTP service request.</td>
</tr>
<tr>
<td>LET resp = req.getResponse() LET info.response = resp.getTextResponse() CALL util.JSON.parse(info.response, add_out)</td>
<td>CALL req.setResponseHeader(&quot;Content-Type&quot;,&quot;application/json&quot;) CALL req.sendTextResponse(200, &quot;OK&quot;, util.JSON.stringify(output_variable))</td>
</tr>
<tr>
<td>The client: • retrieves the HTTP response using a method of the com.HttpRequest class, • returns it as a string using methods of the com.HTTPResponse class, and • parses out the desired value using methods of the util.JSON class.</td>
<td>The server returns a response to the client.</td>
</tr>
</tbody>
</table>
How Do I ... ?

These topics provide you with the information needed to perform specific tasks related to Genero Web Services (SOAP or RESTful).

Tip: See here for SOAP-specific "How Do I … ?” topics.

Call Java APIs from Genero in a SOA environment

How to call a Java library from Genero in a SOA environment, using Genero and Java Web services.

This can easily be done using the Java JAX-WS framework on a server, and a Genero application for the client part.

Note: There is no strong linkage between Genero and a java JVM. You can, for instance, upgrade the Java service without changing anything in your Genero code.

These topics are presented as a tutorial that uses a Java barcode creation library to build a picture from a code.

Related concepts

Call .NET APIs from Genero in a SOA environment on page 3450

How to call a .NET library from Genero in a SOA environment, using Genero and Web services, and IIS and Visual Studio .NET.

Recommendation

The usage of Genero Web Services to call a Java service is recommended in a SOA environment. It enables several Genero applications to connect to a centralized Java service without the need to start a new JVM for each running Genero application.

It also provides more flexibility because there is no strong linkage between Genero and the Java virtual machine. You can, for instance, upgrade the Java service without changing anything in your Genero code.

However, due to the XML serialization process and the HTTP transport protocol in Web Services, there can be some performance issues. So if your main concern is performance, it is recommended to use the Genero Java bridge.

Prerequisites

• A JRE 1.5 or above
• The Java barcode library (available here)
  • You must add these JARs to the Java CLASSPATH: barcode.jar and BarcodeReader.jar
  • The trial version has some functions partially implemented.
• Download the JAX-WS framework from the Sun® metro project here; add this JAR to the java CLASSPATH: webservices-tools.jar

Using the barcode library

The barcode library is composed of two libraries:

• A library for building a barcode image from a numeric code
• A library for reading a barcode image to return the numeric code

This section is based on the library you want to use in Genero.

In our tutorial, we create two functions called buildImage and readImage.

This is the Java implementation:

```java
buildImage( type : String, code : String) : byte[ ]
```

```java
try {
    Barcode builder=new Barcode();
    builder.setType(GetBarcodeBuilderType(type));
    builder.setData(data);
```
Web services

```java
builder.setAddCheckSum(true);
ByteArrayOutputStream out=new ByteArrayOutputStream();
if (builder.createBarcodeImage(out)) {
    byte[] ret = out.toByteArray();
    return ret;
} else {
    return null;
}
} catch (Exception e) {
    return null;
}
```

```java
readImage( type : String, img : byte[] ) : String
try {
    File f=new File("tmp.jpg");
    FileOutputStream stream=new FileOutputStream(f);
    stream.write(img);
    stream.close();
    String[] datas =
        BarcodeReader.read(f, GetBarcodeReaderType(type));
    if (datas==null) {
        return null;
    } else {
        String ret = datas[0];
        return ret;
    }
} catch (Exception e) {
    return null;
}
```

The following two functions convert the barcode type to the type expected by the library:

```java
private int GetBarcodeBuilderType(String str) {
    if (str.equals("CODABAR")) {
        return Barcode.CODABAR;
    } else if (str.equals("CODE11")) {
        return Barcode.CODE11;
    } else if (str.equals("CODE128")) {
        return Barcode.CODE128;
    } else if (str.equals("CODE128A")) {
        return Barcode.CODE128A;
    } else if (str.equals("CODE128B")) {
        return Barcode.CODE128B;
    } else if (str.equals("CODE128C")) {
        return Barcode.CODE128C;
    } else if (str.equals("CODE2OF5")) {
        return Barcode.CODE2OF5;
    } else if (str.equals("CODE39")) {
        return Barcode.CODE39;
    } else if (str.equals("CODE39EX")) {
        return Barcode.CODE39EX;
    } else if (str.equals("CODE93")) {
        return Barcode.CODE93;
    } else if (str.equals("CODE93EX")) {
        return Barcode.CODE93EX;
    } else if (str.equals("EAN13")) {
        return Barcode.EAN13;
    } else if (str.equals("EAN13_2")) {
        return Barcode.EAN13_2;
    } else if (str.equals("EAN13_5")) {
        return Barcode.EAN13_5;
    } else if (str.equals("EAN8")) {
        return Barcode.EAN8;
    }
```
} else if (str.equals("EAN8_2")) {
    return Barcode.EAN8_2;
} else if (str.equals("EAN8_5")) {
    return Barcode.EAN8_5;
} else if (str.equals("INTERLEAVED25")) {
    return Barcode.INTERLEAVED25;
} else if (str.equals("ITF14")) {
    return Barcode.ITF14;
} else if (str.equals("ONECODE")) {
    return Barcode.ONECODE;
} else if (str.equals("PLANET")) {
    return Barcode.PLANET;
} else if (str.equals("POSTNET")) {
    return Barcode.POSTNET;
} else if (str.equals("RM4SCC")) {
    return Barcode.RM4SCC;
} else if (str.equals("UPCA")) {
    return Barcode.UPCA;
} else if (str.equals("UPCE")) {
    return Barcode.UPCE;
} else {
    return -1;
}

private int GetBarcodeReaderType(String str) {
    if (str.equals("CODABAR")) {
        return BarcodeReader.CODABAR;
    } else if (str.equals("CODE11")) {
        return BarcodeReader.CODE11;
    } else if (str.equals("CODE128")) {
        return BarcodeReader.CODE128;
    } else if (str.equals("CODE39")) {
        return BarcodeReader.CODE39;
    } else if (str.equals("CODE39EX")) {
        return BarcodeReader.CODE39EX;
    } else if (str.equals("CODE93")) {
        return BarcodeReader.CODE93;
    } else if (str.equals("DATAMATRIX")) {
        return BarcodeReader.DATAMATRIX;
    } else if (str.equals("EAN13")) {
        return BarcodeReader.EAN13;
    } else if (str.equals("EAN8")) {
        return BarcodeReader.EAN8;
    } else if (str.equals("INTERLEAVED25")) {
        return BarcodeReader.INTERLEAVED25;
    } else if (str.equals("ITF14")) {
        return BarcodeReader.ITF14;
    } else if (str.equals("ONECODE")) {
        return BarcodeReader.ONECODE;
    } else if (str.equals("PLANET")) {
        return BarcodeReader.PLANET;
    } else if (str.equals("POSTNET")) {
        return BarcodeReader.POSTNET;
    } else if (str.equals("QRCODE")) {
        return BarcodeReader.QRCODE;
    } else if (str.equals("RM4SCC")) {
        return BarcodeReader.RM4SCC;
    } else if (str.equals("RSS14")) {
        return BarcodeReader.RSS14;
    } else if (str.equals("RSSLIMITED")) {
        return BarcodeReader.RSSLIMITED;
    } else if (str.equals("UPCA")) {
        return BarcodeReader.UPCA;
Calling Java from Genero

The integration of one or several Java libraries with multiple methods in a Genero application can be performed, as described in the following topics.

**Step 1: Write a new java class**

Instead of writing the functions in 4GL, you simply need to write them in a Java class with the methods you want to use in 4GL. In our example, the two functions are `buildImage` and `readImage`. And of course, don't forget to import the necessary Java import instructions.

```java
import com.barcodelib.barcodereader.BarcodeReader;
import com.barcodelib.barcode.Barcode;
import java.io.*;
import javax.jws.*;
import javax.jws.soap.SOAPBinding;
import javax.xml.ws.Endpoint;

public class BarcodeService {
    public byte[] buildImage(String type, String data) {
        /*BUILDIMAGE IMPLEMENTATION CODE DESCRIBED ABOVE*/
    }
    public String readImage(String type, byte[] img) {
        /*READIMAGE IMPLEMENTATION CODE DESCRIBED ABOVE*/
    }
}
```

If you want the service to run standalone, you must also add the following main method to tell the system the port number on which the service will run:

```java
public static void main(String[] args) {
    String endpointUri = "http://localhost:9090/";
    Endpoint.publish(endpointUri, new BarcodeService());
    System.out.println("BarcodeService started at " + endpointUri);
}
```

In the next step we transform the Java class in a Web Service, **Step 2: Transform the Java class in a Web service** on page 3448.

**Step 2: Transform the Java class in a Web service**

To transform the Java class created in **Step 1: Write a new java class** on page 3448 into a Web Service, simply add a `WebService` annotation.

```java
@WebService(targetNamespace = http://www.mycompany.com/barcode",
    name="Barcode",
    serviceName="BarcodeService")
public class BarcodeService{
    ...
}
```
This defines all public and non static methods of the class as operations of the **BarcodeService** Web Service.

In the next step we start the Web Service, **Step 3: Start the service** on page 3449.

**Step 3: Start the service**

Compile the Java class created in **Step 2: Transform the Java class in a Web service** on page 3448, and run it.

Commands to compile and execute the service in standalone mode:

```
$ javac BarcodeService.java
$ java BarcodeService
```

Once the service is started, it is ready to accept requests and you can also retrieve its WSDL at the following URL:

http://localhost/9090/BarcodeService?WSDL

**Note:** If you want the service to be started on a Web server, you must deploy it first using Eclipse or the Web Server deployment tools.

In the next step we generate the client stub to access the BarcodeService, **Step 4: Generate BDL stub to access the Java library** on page 3449.

**Related concepts**
**Deploy a Web Service** on page 3459
Genero Web services need to be deployed on a Genero Application Server that plugs into a Web server, such as Apache. Understand what you need to implement for configuration and security.

**Step 4: Generate BDL stub to access the Java library**

Use the `fglwsdl` tool to generate the client stub to access the BarcodeService:

```
$ fglwsdl http://localhost:9090/BarcodeService?WSDL
```

This will create two `.4gl` files that must be compiled and linked into your BDL application in order to call the Java barcode library functions. These files contain the BDL interface to access the Java library where you will find the two functions, `readImage` and `buildImage`, defined in BDL.

In the final step we modify the existing application, **Step 5: Modify your BDL application** on page 3449.

**Related concepts**
**WS client stubs and handlers** on page 3328
To access a remote Web Service, you first must get the WSDL information from the service provider.

**Step 5: Modify your BDL application**

The last step is to modify the existing application where you want to use the Java library, by calling the BDL functions generated in the stub. Then compile your application and the previously generated stubs, and link everything together.

Your application is now ready to use the different features of your Java library.

**Related concepts**
**Example program** on page 3449

**Conclusion** on page 3450

**Example program**

This program calls the `buildImage` function of the Barcode Java library.

```
GLOBALS "BarcodeService_BarcodePort.inc"

MAIN
```
DEFINE wsstatus INTEGER

IF num_args() != 3 THEN
    CALL ExitHelp()
END IF

LET ns1buildImage.arg0 = arg_val(1)
LET ns1buildImage.arg1 = arg_val(2)
LOCATE ns1buildImageResponse.return IN MEMORY

LET wsstatus = buildImage_g()
IF wsstatus <> 0 THEN
    DISPLAY "Error ("||wsError.code||") : ",wsError.description
ELSE IF
    IF ns1buildImageResponse.return IS NULL THEN
        DISPLAY "Encoding failed"
    ELSE
        CALL ns1buildImageResponse.return.writeFile(arg_val(3))
    END IF
END IF
FREE ns1buildImageResponse.return
END MAIN

FUNCTION ExitHelp()
    DISPLAY arg_val(0)||" <type> <data> <filename>"
    DISPLAY "type : barcode type such as EAN8 or CODE128"
    DISPLAY "data : data to be encoded with a barcode [0-9A-D]"
    DISPLAY "filename : resulting image filename"
    DISPLAY "example : createImage EAN8 12358723A mybarcode.jpg"
EXIT PROGRAM (-1)
END FUNCTION

Conclusion
In a SOA environment, you can call any Java library from Genero using Web Services, and without a strong
dependency to a JVM. This follows SOA principles - it allows you to reuse the Java library in another BDL
application without any new development, you can update the Java part without recompiling any .4gl source, and
integrate any function available from a SOA platform.

Call .NET APIs from Genero in a SOA environment
How to call a .NET library from Genero in a SOA environment, using Genero and Web services, and IIS and Visual
Studio .NET.
There is no strong linkage between Genero and .NET. You can call a .NET library from a non-Windows® Genero
platform.
These topics are presented as a tutorial that uses a .NET barcode creation library to build a picture from a numeric
code, and C# as the development language. This works with any .NET language.

Related concepts
Call Java APIs from Genero in a SOA environment on page 3445
How to call a Java library from Genero in a SOA environment, using Genero and Java Web services.

Prerequisites
• IIS (Internet Information Services) Web server
• Visual Studio Professional Edition C#
  • Visual Studio is only needed for development. Once the service is built, you can deploy on any IIS Web
    Server.
- The .NET barcode library (available here)
  - The trial version has some functions partially implemented.
  - The .NET library is called BarcodeLib.Barcode.dll, and must be added to the Visual Studio Project.

**Using the barcode library**

This section is based on using the library in Genero. In our tutorial, we create one function called `buildImage`. This is the C# implementation:

```csharp
buildImage( type : String, code : String ) : byte[]
```

```csharp
Linear barcode = new Linear();
barcode.Data = code;
barcode.Type = GetBarcodeBuilderType(type);
barcode.AddChecksum = true;
// save barcode image into your system
barcode.ShowText = true;
byte[] ret = barcode.drawBarcodeAsBytes();
if (ret != null) return ret;
else return null;
```

You will also need to convert the barcode type to the type expected by the library. Therefore, you will need this function.

```csharp
private BarcodeType GetBarcodeBuilderType(String str)
{
    if (str.Equals("CODABAR")) {
        return BarcodeType.CODABAR;
    } else if (str.Equals("CODE11")) {
        return BarcodeType.CODE11;
    } else if (str.Equals("CODE128")) {
        return BarcodeType.CODE128;
    } else if (str.Equals("CODE128A")) {
        return BarcodeType.CODE128A;
    } else if (str.Equals("CODE128B")) {
        return BarcodeType.CODE128B;
    } else if (str.Equals("CODE128C")) {
        return BarcodeType.CODE128C;
    } else if (str.Equals("CODE2OF5")) {
        return BarcodeType.CODE2OF5;
    } else if (str.Equals("CODE39")) {
        return BarcodeType.CODE39;
    } else if (str.Equals("CODE39EX")) {
        return BarcodeType.CODE39EX;
    } else if (str.Equals("CODE93")) {
        return BarcodeType.CODE93;
    } else if (str.Equals("EAN13")) {
        return BarcodeType.EAN13;
    } else if (str.Equals("EAN13_2")) {
        return BarcodeType.EAN13_2;
    } else if (str.Equals("EAN13_5")) {
        return BarcodeType.EAN13_5;
    } else if (str.Equals("EAN8")) {
        return BarcodeType.EAN8;
    } else if (str.Equals("EAN8_2")) {
        return BarcodeType.EAN8_2;
    } else if (str.Equals("EAN8_5")) {
        return BarcodeType.EAN8_5;
    } else if (str.Equals("INTERLEAVED25")) {
        return BarcodeType.INTERLEAVED25;
    } else if (str.Equals("ITF14")) {
        return BarcodeType.ITF14;
    }
    return BarcodeType.OTHER;
}
```
Calling .NET from Genero

Step 1: Create an ASP.NET Web Service Application

Start Visual Studio, and create a new web project with the name "BarCodeService".

Figure 136: Visual Studio New Project; ASP.NET Web Service Application selected

In the next step we rename the generated files, Step 2: Rename the generated files on page 3452.

Step 2: Rename the generated files

Rename the generated class called Service1 with an appropriate name such as "BarCode", and rename the file Service1.asmx to BarCodeService.asmx, for instance.

The .asmx file is the file that is accessible from the IIS web server once the application is deployed. The .asmx file also contains a reference to the default generated class, Service1, which must also be renamed to the new name (BarCode in our tutorial), in case Visual Studio didn't make the change automatically.

The class view after renaming the class:
In the next step we add the barcode library as a reference, Step 3: Add the barcode library as a reference on page 3453.

**Step 3: Add the barcode library as a reference**

Right-click on the Solution Explorer, select Add Reference and use the Browse panel to enter the location of the barcode library called BarcodeLib.Barcode.dll:
Add the buildImage method

Step 4: Add the buildImage method

Remove the default generated HelloWorld method, and create the buildImage method.

Add the three using instructions to import the barcode library, and the instruction to declare buildImage as a WebMethod. Use the GetBarcodeBuilderType() method to convert a string to a code as expected by the barcode library.

```csharp
using BarcodeLib;
using BarcodeLib.Barcode;
using BarcodeLib.Barcode.Linear;

namespace BarCodeService
{
    /// <summary>
    /// Summary description for Service1
    /// </summary>
    [WebService(Namespace = "http://tempuri.org")]
    [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
    [ToolboxItem(false)]
    {
        [WebMethod]
        public byte[] buildImage(String type, String code)
        {
            try
            {
                Linear barcode = new Linear();
                barcode.Data = code;
                barcode.Type = GetBarcodeBuilderType(type);
                barcode.AddCheckSum = true;
                // save barcode image into your system
                barcode.ShowText = true;
            }
            catch (ParseException e)
            {
```

Note: By default, the barcode library will be copied to the right place when deploying on the IIS web server.
byte[] ret = barcode.drawBarcodeAsBytes();
    if (ret != null) return ret;
    else return null;
}
catch (Exception e)
{
    return null;
}
}

namespace BarCodeService
{
    // Summary description for Service
    // WebService(NameSpace = "http://tempuri.org/")
    // WebServiceBinding(ConformsTo = UniProfiles.BasicProfile1_1)
    // ToolboxItem(Icaise)
    // To allow this Web Service to be called from script, using ASP.NET
    {
        [WebMethod]
        public byte[] buildImage(String type, String code)
        {
            try
            {
                Linear barcode = new Linear();
                barcode.Data = code;
                barcode.Type = GetBarcodeBuilderType(type);
                barcode.AddCheckSum = true;

                // save barcode image into your system
                barcode.ShowText = true;
                byte[] ret = barcode.drawBarcodeAsBytes();
                if (ret != null) return ret;
                else return null;
            }
            catch (Exception e)
            {
                return null;
            }
        }
    }
}

Figure 139: BarCodeService.BarCode

In the next step we publish the service to your IIS Web server, Step 5: Publish the service on page 3456.
Step 5: Publish the service

Build the entire application, right-click on the solution, and select the publish operation. This will copy all necessary files to your IIS web server and make your application available at an URL, depending on where you deploy it on your IIS web server.

In our tutorial, the service will be located at the root of the server. In other words, it will be available at http://localhost/BarCodeService.asmx and the WSDL at URL http://localhost/BarCodeService.asmx?WSDL

Figure 140: Publish Web dialog

In the next step we generate the stub file, Step 6: Generate .4gl stub to access the .NET library on page 3456.

Step 6: Generate .4gl stub to access the .NET library

Use the fglwsdl tool to generate the client stub to access the BarcodeService, as follows:

$ fglwsdl http://localhost/BarCodeService.asmx?WSDL

This will create two .4gl files, which must be compiled and linked into your BDL application in order to call the .NET barcode library functions. These files contain the BDL interface to access the .NET library where you will find the function buildImage, defined in BDL.

In the final step we modify the existing application, Step 7: Modify your BDL application on page 3456.

Related concepts

WS client stubs and handlers on page 3328
To access a remote Web Service, you first must get the WSDL information from the service provider.

Step 7: Modify your BDL application

Modify your existing application, where you want to use the .NET library, by calling the BDL functions generated in the stub. Then compile your application and the previously generated stubs, and link everything together.

Your application is now ready to use the different features of your .NET library.

Related concepts

Example BDL program on page 3457
Conclusion on page 3457
**Example BDL program**

This program calls the `buildImage` function of the Barcode.NET library.

```bdl
GLOBALS "BarCode_BarCodeSoap.inc"
MAIN
  DEFINE wsstatus INTEGER
  IF num_args() != 3 THEN
    CALL ExitHelp()
  END IF

  LET buildImage.type = arg_val(1)
  LET buildImage.code = arg_val(2)
  LOCATE buildImageResponse.buildImageResult IN MEMORY

  LET wsstatus = buildImage_g()
  IF wsstatus <> 0 THEN
    DISPLAY "Error ("||wsError.code||") : ",wsError.description
  ELSE
    IF buildImageResponse.buildImageResult IS NULL THEN
      DISPLAY "Encoding failed"
    ELSE
      CALL buildImageResponse.buildImageResult.writeFile(arg_val(3))
    END IF
  END IF

  FREE buildImageResponse.buildImageResult
END MAIN

FUNCTION ExitHelp()
  DISPLAY arg_val(0)||" <type> <data> <filename>"
  DISPLAY "type : barcode type such as EAN8 or CODE128"
  DISPLAY "data : data to be encoded with a barcode [0-9A-D]"
  DISPLAY "filename : resulting image filename"
  DISPLAY "example : createImage EAN8 12358723A mybarcode.jpg"
  EXIT PROGRAM (-1)
END FUNCTION
```

**Conclusion**

It is quite easy to interact with a .NET library from Genero using .NET Visual Studio and the web services. Of course you also need an IIS Web server installed on your Windows® system. This means that you can, in the same Genero application, interact with .NET and Java libraries without any strong linkage between Genero and the third party libraries you want to use. This meets the SOA principles that provide more flexibility to your entire BDL application.

You can integrate any new library from another vendor, without the risk of conflicts between different libraries that can happen if you had to link everything together in C or Java.

You can upgrade a third party library without recompiling the BDL application, which will still work.

You can use all these third party libraries in other BDL or other applications.

**Compute a hash value from a BDL string**

How to compute a hash value of a BDL string using the `security.Digest` API.

**Program Example**

This retrieves the hash value from the signature and returns it. The computed hash value is encoded in Base64, so you may have additional conversion to do.
Program example ComputeHash.4gl:

IMPORT SECURITY

MAIN

DEFINE result STRING

IF num_args() != 2 THEN
    DISPLAY "Usage: ComputeHash <string> <hashcode">
    DISPLAY " string: the string to digest"
    DISPLAY " hashcode: SHA1, SHA512, SHA384, SHA256, SHA224, MD5"
ELSE
    LET result = ComputeHash(arg_val(1), arg_val(2))
    IF result IS NOT NULL THEN
        DISPLAY "Hash value is: ", result
    ELSE
        DISPLAY "Error"
    END IF
END IF

END MAIN

FUNCTION ComputeHash(toDigest, algo)

DEFINE toDigest, algo, result STRING
DEFINE dgst security.Digest

TRY
    LET dgst = security.Digest.CreateDigest(algo)
    CALL dgst.AddStringData(toDigest)
    LET result = dgst.DoBase64Digest()
CATCH
    DISPLAY "ERROR : ", STATUS, " - ", SQLCA.SQLERRM
    EXIT PROGRAM(-1)
END TRY

RETURN result

END FUNCTION

Example execution:

$ fglrun ComputeHash "Hello World" SHA1
Hash value is: Ck1VqNd45QIvq3AZd8XYQLvEhtA=

Related concepts
The security package on page 3127
The Genero Web Services security package provides classes and methods to support basic cryptographic features.

**Deploy a Web Service**

Genero Web services need to be deployed on a Genero Application Server that plugs into a Web server, such as Apache. Understand what you need to implement for configuration and security.

**Web services server program deployment**

The Genero Application Server (GAS) manages Web services. You must consider GAS configuration when deploying your Web service in a production environment.

**Introduction**

In a production environment, Genero Web Services becomes a part of a global application architecture handled by the Genero Application Server (GAS). The GWS DVMs are managed by the GAS.

This architecture takes care of:

- Security issues
- Scalability
- Load management
- Balancing of the Web service requests amongst the available virtual machines
- Runtime monitoring

**GAS configuration**

For deployment, the GWS Server application must be added to the GAS configuration. See *Configure applications for Web service* in the *Genero Application Server User Guide*.

The web services application can be added to the GAS in different ways:

- **GWS Server application implementing a single Web service.**
  
  This application can be deployed on various physical machines. A Genero Web Services VMProxy (GWSProxy) is started on each machine where the GWS Server application is executed, to manage the requests for a service and manage the DVMs that handle the requests. A single VMProxy can communicate with multiple GWS DVMs, and manage the load balancing.

- **GWS Server application implementing multiple Web Services.**
  
  The GWSProxy would manage the client requests, dispatching the request to the appropriate DVM and the appropriate web service.

**Note:** A Web Service Server must be stateless; several instances of the same Service can be created to support load balancing.

The basic deployment strategy can be implemented in varying permutations, depending on your business needs and the volume of requests.
### Figure 141: Deployment strategy

- Using the World Wide Web, a Web Service client requests WSDL information for a particular Web Service from the Web Server.
- The Web Service client uses this information to make a Web Service request from the Web Server.
- The Web server passes the request to the GAS dispatcher.
- The GAS dispatcher starts a GWSProxy, which will be in charge of the pool of DVMs that will serve the web service application.
- The GWSProxy will start the number of DVMs specified by the START element defined for the web service application.

For a more detailed explanation of the Services Pool for web services, refer to the GAS Architecture topic in the Genero Application Server User Guide.

### Access the web services server from a client application

To reach the web service from the internet, client applications must use the following URL form:

```
http://host_name/ws/r/app_id
```

1. `host_name` defines the web server host name where the GAS is running.
2. `app_id` is the XCF file of the GAS web services application.

### Related concepts

Introducing the GAS and JGAS on page 2183

The Genero Application Server (GAS) is an engine that plugs in to a Web server for the purpose of delivering Genero Web applications and services. The Genero Application Server for Java (JGAS) is designed to run your applications on the Java EE servlet. A general knowledge of how they operate can be helpful in testing and deploying Web applications.

### Configuring the Apache Web server for HTTPS

Configuration steps to secure a Web service for Apache server in HTTPS.

The first three steps are for the creation of all X.509 certificates. Certificates are created using the OpenSSL command line tool. The next three steps are for server configuration.

### Related concepts

Encryption and authentication on page 3299

A scenario involving a person (Georges) and his bank guides you through the concepts of secured communication, certificates, and certificate authorities.

The OpenSSL (openssl) tool on page 3313

The openssl command line tool creates certificates for the configuration of secured communications.

### Related tasks

View a certificate on page 3316
This procedure allows you to view a certificate using the `openssl` command.

**Step 1: Create the root certificate authority**
Generate a root certificate authority that signs a client certificate.

- Create the root certificate authority serial file:
  
  ```
  $ echo 01 > MyCompanyCA.srl
  ```

- Create the Root Authority's Certificate Signing Request and private key:
  
  ```
  $ openssl req -new -out MyCompanyCA.csr -keyout MyCompanyCA.pem
  ```

- Create the Root Certificate Authority for a period of validity of 2 years:
  
  ```
  $ openssl x509 -trustout -in MyCompanyCA.csr -out MyCompanyCA.crt
  -req -signkey MyCompanyCA.pem -days 730
  ```

**Important:** The private key file (`MyCompanyCA.pem`) of a Root Certificate Authority must be handled with care. This file is responsible for the validity of all other certificates it has signed. As a result, it must not be accessible by other users.

In the next step we create the server's certificate and private key, **Step 2: Create the server's certificate and private key** on page 3461.

**Step 2: Create the server's certificate and private key**
The server's certificate identifies the server as trusted by any client that connects to it.

- Create the server's serial file:
  
  ```
  $ echo 01 > MyServer.srl
  ```

- Create the server's Certificate Signing Request and private key:
  
  ```
  $ openssl req -new -out MyServer.csr
  ```

  **Note:** By default, `openssl` outputs the private key in the `privkey.pem` file.

- Remove the password from the private key:
  
  ```
  $ openssl rsa -in privkey.pem -out MyServer.pem
  ```

  **Note:** The key is also renamed in `MyServer.pem`.

- Create the server's Certificate trusted by the Root Certificate Authority:
  
  ```
  $ openssl x509 -in MyServer.csr -out MyServer.crt
  -req -signkey MyServer.pem -CA MyCompanyCA.crt -CAkey MyCompanyCA.pem
  ```

  **Note:** The purpose of the server's Certificate is to identify the server to any client that connects to it. Therefore, **the subject of that server's certificate must match the host name of the server as it is known on the network**; otherwise the client will not trust the server's identity and the communication is stopped. For instance, if the URL of the server is `https://www.MyServer.com/fastcgi/ws/r/MyWebService`, the subject must be `www.MyServer.com`.

In the next step we create the server's certificate authority list, **Step 3: Create the server's certificate authority list** on page 3462.
Step 3: Create the server’s certificate authority list
Create the certificate authority list and add the root certificate authority cert.

- Create the server’s Certificate Authority List:
  
  ```
  $ openssl x509 -in MyCompanyCA.crt -text >> ServerCAList.pem
  ```

  **Note:** As the server trusts only the Root Certificate Authority, the list contains only that one certificate authority; all other certificates that were trusted by the Root Certificate Authority will also be considered as trusted by the server.

In the next step we register the server's certificate authority list, Step 4: Register the server as a Web service in the GAS on page 3462.

Step 4: Register the server as a Web service in the GAS
Web services registered on the GAS are started automatically when the GAS starts.

As the Web server is in charge of the complete HTTPS protocol with all the clients, there is no additional GAS configuration needed to add security. Simply register the BDL server to the list of Web Services of the GAS. For more information, refer to the Genero Application Server User Guide.

For more details, see Web services server program deployment on page 3459.

In the next step we configure Apache for HTTPS, Step 5: Configure Apache for HTTPS on page 3462.

Step 5: Configure Apache for HTTPS
Add the locations of your certs to the Apache configuration file.

You must configure Apache to support HTTPS by adding the required modules. Refer to the Apache Web server documentation for more information.

- For the Apache 1.3 manual, go to http://httpd.apache.org/docs/1.3.
- For the Apache 2.0 manual, go to http://httpd.apache.org/docs/2.0/.

Once the Apache Web server supports HTTPS, you must change or add the following directives to the Apache configuration file:

- Set the Apache Web server Certificate Authority List directive created in Step 3:
  ```
  SSLCACertificateFile D:/Apache-Server/conf/ssl/ServerCAList.pem
  ```

- Set the Apache Web server Certificate and associated private key directives created in Step 2:
  ```
  SSLCertificateFile D:/Apache-Server/conf/ssl/MyServer.crt
  SSLCertificateKeyFile D:/Apache-Server/conf/ssl/MyServer.pem
  ```

- Require the Apache Web server to verify the validity of all client certificates:
  ```
  SSLSVerifyClient require
  ```

  **Note:** The Apache Web server must be started on a machine where the host is the same as the one defined in the subject of the server's certificate (www.MyServer.com in our example).

In the next step we configure Apache for HTTP basic authentication, Step 6: Configure Apache for HTTP basic authentication on page 3462.

Step 6: Configure Apache for HTTP basic authentication
Create login details for an authenticated user for the Apache web server, and add the location of your authentication file to the Apache configuration file.

You must configure Apache to support HTTP basic authentication by adding the required modules. Refer to the Apache Web server documentation for more information.

- For the Apache 1.3 manual, go to http://httpd.apache.org/docs/1.3.
- For the Apache 2.0 manual, go to http://httpd.apache.org/docs/2.0/.
Once the Apache Web server supports HTTP basic authentication, you must:

1. Add an user to the Apache Web server basic authentication file with the same login and password as defined for the client.

   Apache provides the tool htpasswd that you can use to create the file and add the user. To add the user `mylogin` with the password `mypassword` to a new file called `myusers`:

   ```
   $ htpasswd -c myusers mylogin mypassword
   ```

   **Note:** To add additional users, remove the option `-c`.

2. Add an Apache Web server location directive that enables you to group several directives for one URL. (In the example, the URL is `/fastcgi/ws/r/MyWebService`).

   The following example (based on Apache 2.0) defines the HTTP authentication type (Basic), with a user file (`user-basic`) containing the login and password of those who are allowed to access the service.

   ```
   <Location /fastcgi/ws/r/MyWebService>
   AllowOverride None
   Order allow,deny
   Allow from all
   #
   # Basic HTTP authenticate configuration
   #
   AuthName "Top secret"
   AuthType Basic
   AuthUserFile "D:/Apache-Server/conf/authenticate/myusers"
   Require valid-user # Means any user in the password file
   </Location>
   ```

   For more information about Apache Web server directives, refer to the Apache Web Server manual.

**Related tasks**

- [Configure a WS client to access an HTTPS server](#) on page 3305
- Configuration steps to access a server in HTTPS.

**Reference**

These topics are the reference guides for Genero Web Services.

**High-level RESTful Web service attributes**

**Table 720: Resource operation attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>WSDelete</strong></td>
<td>In order to remove an existing resource, you set the WSDelete attribute.</td>
</tr>
<tr>
<td><strong>WSGet</strong></td>
<td>In order to retrieve a resource, you set the WSGet attribute.</td>
</tr>
<tr>
<td><strong>WSPost</strong></td>
<td>In order to create a new resource, you set the WSPost attribute.</td>
</tr>
<tr>
<td><strong>WSPut</strong></td>
<td>Update an existing resource with the WSPut attribute.</td>
</tr>
</tbody>
</table>
### Table 721: Path templating attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSParam</td>
<td>Specifies a value for a template path to a REST Web service resource.</td>
</tr>
<tr>
<td>WSPATH = &quot;/ \path-\element</td>
<td>value-\template [/...]&quot;</td>
</tr>
<tr>
<td></td>
<td>Specifies a path to a REST Web service resource that identifies its function and allows parameters to be passed in the URL.</td>
</tr>
</tbody>
</table>

### Table 722: Query, cookie, header attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSCookie</td>
<td>Allows cookies to be passed to Web service REST function.</td>
</tr>
<tr>
<td>WSHeader</td>
<td>Defines custom headers in the request and response of Web service functions.</td>
</tr>
<tr>
<td>WSQuery</td>
<td>Specifies parameters to be passed at the end of the request URL after a question mark (?).</td>
</tr>
</tbody>
</table>

### Table 723: Name, description, optional attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSDescription</td>
<td>Describes REST function parameters.</td>
</tr>
<tr>
<td>WSName</td>
<td>Specify alternative names for parameters in a REST function.</td>
</tr>
<tr>
<td>WSOptOptional</td>
<td>Qualifies a parameter as optional in a REST function.</td>
</tr>
</tbody>
</table>

### Table 724: File handling attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSAAttachment</td>
<td>Defines file attachments in the REST message.</td>
</tr>
<tr>
<td>WSMedia = &quot;MIME-type [...,]&quot;</td>
<td>Defines the data format of the REST message.</td>
</tr>
</tbody>
</table>
Table 725: HTTP code attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSRetCode= &quot;{</td>
<td>code ↓ code:description ↓&quot;</td>
</tr>
</tbody>
</table>

Table 726: Context attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSContext</td>
<td>Defines an injection variable to retrieve REST operation context values at the service level.</td>
</tr>
</tbody>
</table>

Table 727: Error handling attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSError= &quot;description&quot;</td>
<td>Specifies a description of the status-code that is returned in the HTTP response.</td>
</tr>
<tr>
<td>WSThrows = &quot;{</td>
<td>code ↓ code:description ↓</td>
</tr>
</tbody>
</table>

Table 728: Security handling attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSInfo</td>
<td>Specify information about the Web service such as title, version, contact details, scopes, etc.</td>
</tr>
<tr>
<td>WSScope = &quot;{</td>
<td>scope ↓ ❱,❙,❍,❱&quot;</td>
</tr>
</tbody>
</table>

Resource operation attributes
Attributes that map an HTTP operation in a function to a REST resource.

Important: One HTTP verb action such as WSGet, WSPost, WSPut, or WSDelete is a mandatory attribute in a REST Web service function.

Related concepts
Define your resource operations on page 3377
Operations are the HTTP methods used to manipulate the resources of your Web service.

HTTP verbs and attributes
HTTP verbs are defined by the high-level RESTful attributes. Some verbs have requirements for request or response body and others do not.

Table 729: HTTP verbs and attributes on page 3466 provides a list of HTTP verbs and their corresponding attributes. In the Request body and Response body columns, you find information about each verb's requirements for message body. In summary:

- POST, and PUT requests are the only verbs that require an input message body. Failing to specify a message body in a request will cause the error-9106.
When you compile your Web service application, the fglcomp tool checks that attributes comply with these requirements.

**Table 729: HTTP verbs and attributes**

<table>
<thead>
<tr>
<th>Verbs</th>
<th>Attributes</th>
<th>Request body</th>
<th>Response body</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>WSGet</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>POST</td>
<td>WSPost</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>PUT</td>
<td>WSPut</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>DELETE</td>
<td>WSDelete</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Related concepts**

Set a request body on page 3383

Functions that create or update a resource need to set a request body for the incoming payload. You specify the request body in an input parameter.

Set a response body and header on page 3384

Functions that create, update, delete, or get a resource need to set a response body. You specify this in a return parameter without an attribute. Other return values can be sent in headers, using the WSHeader attribute.

**WSDelete**

In order to remove an existing resource, you set the WSDelete attribute.

**Syntax**

```
WSDelete
```

**Usage**

You use this attribute to specify the action of the HTTP verb DELETE to delete a resource. You set the WSDelete attribute in the ATTRIBUTES() clause of the function.

**Example using WSDelete to delete a resource**

```public function DeleteUserProfile(
    id STRING ATTRIBUTE(WSParam) )
attributes (WSDelete,
    WSPath="/users/{id}",
    WSDescription='Deletes a user profile',
    WSThrows='404:user not found')
returns STRING
    define ret STRING
    # ... function code ...
    let ret = sfmt("Deleted user with ID: %1",id)
    return ret
end function```

In this example the resource specified in the URL is deleted.

```
http://host:port/gas/ws/r/xcf/users/xxx
```

`xxx` is the user id replaced in the Genero BDL function.
**WSGet**
In order to retrieve a resource, you set the WSGet attribute.

**Syntax**

```
WSGet
```

**Usage**
You use this attribute to specify the action of the HTTP verb GET to return data from a resource. You set the `WSGet` attribute in the `ATTRIBUTES()` clause of the function.

**Example using WSGet to return a user record**

```
TYPE profileType RECORD ... END RECORD

PUBLIC FUNCTION GetUserProfile(
    id STRING ATTRIBUTE(WSParam)
) ATTRIBUTES (WSGet,
    WSPath="/users/{id}",
    WSDescription="Returns a user profile",
    WSThrows="404:user not found"
) RETURNS (profileType ATTRIBUTE(WSName="User_data", WSMedia="application/json")
)
    DEFINE data profileType
    # ... function code ...
    RETURN data.*
END FUNCTION
```

In this example the REST function gets the specified user record. The resource URL is:

```
http://host:port/gas/ws/r/xcf/users/xxx
```

`xxx` is the user id replaced in the Genero BDL function.

**Related concepts**

- Example: Get operation with WSGet on page 3377
- Example of methods you can use to get data from a resource with the WSGet attribute.

**WSPost**
In order to create a new resource, you set the WSPost attribute.

**Syntax**

```
WSPost
```

**Usage**
You use this attribute to specify the action of the HTTP verb POST to create a new resource. You set the `WSPost` attribute in the `ATTRIBUTES()` clause of your function.

**Example WSPost to create a new user record**

```
TYPE profileType RECORD ... END RECORD

PUBLIC FUNCTION CreateUserProfile( newUser profileType )
    ATTRIBUTES (WSPost,
        WSPath="/users",
        WSThrows="404:user not found")
```

**Example: Get operation with WSGet on page 3377**

Example of methods you can use to get data from a resource with the WSGet attribute.
returns string
define ret string
# ... function code ...
let ret = sfmt("Created user with ID: %1",id)
return ret
end function

In this example a new resource is created. newUser is data passed in the message body in either JSON or XML format. An example of the resource URL is:

http://host:port/gas/ws/r/xcf/users

WSPut
Update an existing resource with the WSPut attribute.

Syntax

WSPut

Usage
You use this attribute to specify the action of the HTTP verb PUT to update an existing resource. You set the WSPut attribute in the ATTRIBUTES() clause of the function.

Example WSPut to update a user record

TYPE profileType RECORD ...
END RECORD

PUBLIC FUNCTION UpdateUserProfile(
    id STRING ATTRIBUTE(WSParam),
    thisUser profileType )
ATTRIBUTES( WSPut,
    WSPath="/users/{id}",
    WSDescription="Update user with the given id",
    WSThrows='404:user not found')
returns string
define ret string
# ... function code ...
let ret = sfmt("Updated user with ID: %1",id)
return ret
end function

In this example, a resource is updated. thisUser is data passed in the message body in either JSON or XML format. An example of the resource URL shows xxx as the user id replaced in the Genero BDL function.

http://host:port/gas/ws/r/xcf/users/xxx

Path templating attributes
Attributes that define the format of the URL of the REST functions.

Related concepts
Set resource path with WSParam and WSPath on page 3380
Path parameters allow you to specify variables in the resource URL.

**WSPParam**
Specifies a value for a template path to a REST Web service resource.

**Syntax**

```
WSPParam
```

**Usage**

You use this attribute for path templating to specify values that are provided at runtime in function parameters. You set the WSP Param attribute in the ATTRIBUTES() clause of the function.

For each Genero BDL function parameter with a WSPParam attribute, there must be a matching template path in the WSPPath on page 3469. fglcomp checks to ensure that there is one template value per parameter, otherwise compilation error-9111 is thrown.

WSPParam is an optional attribute.

**Example path templating with WSPParam**

```plaintext
TYPE accountType RECORD ... END RECORD

PUBLIC FUNCTION getAccountById(id INTEGER ATTRIBUTE(WSPParam))

    ATTRIBUTES(WSGet,
        WSPPath="/accounts/{id}",
        WSDescription="Returns an account record",
        WSThrows="404:not found"
    )
    RETURNS accountType ATTRIBUTES(WSName="Account",WSMedia="application/xml")
    DEFINE p_accountRec accountType
    # ... function code ...
    RETURN p_accountRec.*
END FUNCTION
```

The client application needs to provide an appropriate parameter value when making a call to the function. For example, the variable part of the path (id) is replaced by the integer value /4 in the URL:

```
http://host:port/gas/ws/r/MyService/accounts/4
```

**Related concepts**

- Set resource path with WSPParam and WSPPath on page 3380
  Path parameters allow you to specify variables in the resource URL.
- Set query, header, or cookie parameters (if needed) on page 3382
  You define these parameters in your function if the resource needs data to be passed as query, cookie, or header.

**WSPPath**

Specifies a path to a REST Web service resource that identifies its function and allows parameters to be passed in the URL.

**Syntax**

```
WSPPath = "// path-element value-template /..."
```
where *path-element* is an identifier:

```
identifier
```

and where *value-template* is an identifier enclosed in curly brackets:

```
{identifier}
```

1. *path-element* is a slash-separated list of path elements
2. *value-template* elements are place holders for a slash-separated list of variable values.

**Usage**

You use this attribute to set the URL to a resource. A URL to, for example, the "users" resource is specified as follows:

```
WSPath="/users"
```

**Note:** The **WSPath** string begins with a slash (/), and there is a slash between each path element and/or template value.

If you need to specify a resource within a collection, such as a specific user identified by ID, and when a pattern could match many similar resources, it is specified with a path template.

```
WSPath=/users/{id}
```

The path template is an identifier enclosed in curly brackets { }. Zero, one, or several path templates can be specified. Values are substituted when a GWS client makes a call to the resource.

Path template values are provided at runtime in function parameters which have **WSParam** on page 3469 attributes. As many template values as path templates must be specified. In other words there must be a function parameter with a **WSParam** attribute to match each template path. **fglcomp** checks to ensure that there is one template value per parameter, otherwise compilation error-9111 is thrown.

You set the **WSPath** attribute in the **ATTRIBUTES()** clause of the function. **WSPath** is an optional attribute. If there is no **WSPath** specified, the function name (case sensitive) is the path to the resource in the URL.

**Example path templating with **WSParam**

```
TYPE accountType RECORD ... END RECORD

PUBLIC FUNCTION getAccountById(id INTEGER ATTRIBUTE(WSParam))

   ATTRIBUTES(WGet,
      WPath="/accounts/{id}",
      WDescription="Returns an account record",
      WThrows="404: not found"
   )
   RETURNS accountType ATTRIBUTES(WSName="Account",WMedia="application/xml")
   DEFINE p_accountRec accountType
   # ... function code ...
   RETURN p_accountRec.*
END FUNCTION
```

The client application needs to provide an appropriate parameter value when making a call to the function. For example, the variable part of the path *(id)* is replaced by the integer value /4 in the URL:

```
http://host:port/gas/ws/r/MyService/accounts/4
```
Related concepts

Set resource path with WParam and WPath on page 3380

Path parameters allow you to specify variables in the resource URL.

Set query, header, or cookie parameters (if needed) on page 3382

You define these parameters in your function if the resource needs data to be passed as query, cookie, or header.

Query, cookie, header attributes

Standard REST attributes defining parameters handling queries, cookies, and headers in REST functions.

WSQuery

Specifies parameters to be passed at the end of the request URL after a question mark (‘?’).

Syntax

```
WSQuery
```

This attribute supports a query in the URL. Zero, one, or several parameters can be specified.

WSQuery is an optional attribute.

Usage

You set the WSQuery attribute in an ATTRIBUTE() clause of an input parameter (only) of the function. Its use is suited to filtering on resource collections. The client application needs to provide appropriate parameter values in the URL when making a call to the function, such as /users?fname=john.

Example WSQuery

```
TYPE accountType RECORD
   id VARCHAR(10),
   fname VARCHAR(25),
   lname VARCHAR(25)
# ...
END RECORD

PUBLIC FUNCTION getAccountRecords(
   resourceId VARCHAR(10) ATTRIBUTES(WSQuery, WOptional, WSName = "id"),
   firstName VARCHAR(25) ATTRIBUTES(WSQuery, WOptional, WSName = "fname"),
   lastName VARCHAR(25) ATTRIBUTES(WSQuery, WOptional, WSName = "lname"))
   ATTRIBUTES (WGet,
      WPath = "/accounts",
      WDescription = "Fetches the accounts resource with the optional filter value(s) applied.",
      WScope = "officestore.user",
      WThrows = "404:Not Found,500:Internal Server Error"
   )
   RETURNS (DYNAMIC ARRAY ATTRIBUTE(WSName = "accounts") OF
      accountType ATTRIBUTES(XMLName = "account", WMedia = "application/json, application/xml"))
   
   DEFINE recList DYNAMIC ARRAY OF accountType
   # ... function code ...
   RETURN recList

END FUNCTION
```

This function has WSQuery parameters which are optional. This allows the function to be used to query any of the parameter values if provided in the query string of the URL, for example:

```
http://host:port/gas/ws/r/MyService/accounts?fname=value&lname=value
```
WSHeader

Defines custom headers in the request and response of Web service functions.

Syntax

```plaintext
WSHeader
```

Where:

WSHeader is defined in an `ATTRIBUTE()` clause in an input parameter or return value of the function.

Usage

You use the `WSHeader` attribute to define a custom header. It is valid to set this in input parameters and in return values. A typical use of `WSHeader` is to pass data related to the service in the form of HTTP header.

Example WSHeader in input parameter

```plaintext
IMPORT com

PUBLIC DEFINE myerror RECORD ATTRIBUTE(WSError="My error")
    code INTEGER,
    reason STRING
END RECORD

PUBLIC FUNCTION getRemoteAddress (ip_addr STRING
    ATTRIBUTE(WSHeader, WSOptional, WSName="X-FourJs-Environment-Variable-REMOTE_ADDR")
) 
    ATTRIBUTES (WSGet,
        WSPath="/users/ip",
        WSDescription="Get remote address of the client",
        WSThrows="400:Invalid,404:NotAvailable"
    )
    RETURNS ( INTEGER ATTRIBUTE(WSHeader), STRING)
    DEFINE ip STRING
    TRY
        LET ip=ip_addr
        IF ip IS NULL THEN
            LET ip="Got no remote address."
        ELSE
            LET ip = SFMT("Hello there, you're at %1",ip )
        END if
    CATCH
        LET ip="Error getting remote address."
        LET myerror.reason = SFMT("Error during execution: %1 [%2]", SQLCA.SQLCODE, SQLERRMESSAGE )
        CALL com.WebServiceEngine.SetRestError(400,myerror)
    END TRY
    RETURN 3, ip
END FUNCTION
```

In the call to the function, a header named "X-FourJs-Environment-Variable-REMOTE_ADDR" provides the IP address of the remote client. The input is optional as defined by the attribute `WSOptional`.

Tip: The `WSContext` on page 3481 attribute, if set, can provide your service with access to environment variables set by the GAS.
Example WSHeader in return value

PUBLIC FUNCTION help()
    ATTRIBUTES (WSGet)
    RETURNS (INTEGER ATTRIBUTE(WSHeader), STRING)
    # ... function code ... 
    RETURN 3, "hello world"
END FUNCTION

In the response, you define a custom header in the RETURNS clause of your function.

In the HTTP output the integer ("3") is returned in the header and the string ("hello world") is returned in the response body.

Unnamed parameters and return values are given default names, "rv0", "rv1", etc., at runtime. You can change default header naming via the WSName on page 3475 attribute, for example with: ATTRIBUTE(WSHeader, WSName="MyHeader")

WSCookie
Allows cookies to be passed to Web service REST function.

Syntax

WSCookie

Usage

You set this attribute to indicate that a parameter value will transport a cookie. The client application needs to provide the appropriate header and value when making a call to the function.

WSCookie is defined in an ATTRIBUTE() clause of an input parameter of the function.

WSCookie is an optional attribute.

Example WSCookie

IMPORT  com

TYPE profileType RECORD
    id INT,
    name VARCHAR(50),
    email VARCHAR(100),
    ccode VARCHAR(10)
    # ...
END RECORD

DEFINE myError RECORD ATTRIBUTE(WSError="My error")
    code INTEGER,
    reason STRING
END RECORD

PUBLIC FUNCTION getUsersByMyCountry(
    p_country STRING ATTRIBUTE(WSCookie, WSOptional, WSName = "Preferred_country") )
    ATTRIBUTES (WSGet,
        WSPath="/users",
        WSDescription="Get users with cookie (for country) if sent from client.",
        WSThrows="400:Invalid,404:NotAvailable"
    )
    RETURNS (
        DYNAMIC ARRAY ATTRIBUTE(WSName="Users",WSMedia="application/xml,application/json") OF profileType
In the call to the function, a header named "Preferred_country" provides the cookie value.

**Name, description, optional attributes**
Standard REST attributes define names for, and descriptions of parameters. Optional parameters are set using the optional attribute in REST functions.

**WSDescription**
Describes REST function parameters.

**Syntax**

```plaintext
WSDescription
```

**Usage**
You use this attribute to provide useful description for parameters in the function. You can set `WSDescription` on:

- Any input and output parameter of a REST operation.
- The attribute clause of the function.

Information you specify with this attribute, is generated for the function in the OpenAPI specification file. You can control generation of descriptions in the stub file with the `--comment` option of the `fglrestful` on page 2085 tool.

**Example using WSDescription**

```plaintext
TYPE orderType RECORD ... END RECORD

PUBLIC FUNCTION getOrders(  
category STRING ATTRIBUTE(WSQuery, WSOptional,  
WSName="cat",WSDescription="Product category"),  
sortfield STRING ATTRIBUTE(WSHeader, WSOptional,  
WSName="sortfld",WSDescription="Sort field"),  
popbrand STRING ATTRIBUTE(WSCookie, WSOptional, WSNamem =  
"popbrand",WSDescription="Most popular brand"))  
ATTRIBUTES (WSGet,
```
WSPath = "/orders",
    WSDescription = "Fetches orders from the resource with the optional filter value(s) applied."
    WSScope = "officestore.user",
    WSThrows = "404:Not Found,500:Internal Server Error"
) 
RETURNS (DYNAMIC ARRAY ATTRIBUTE(WSName = "Orders") OF orderType 
    ATTRIBUTES(XMLName = "order", 
        WSMedia = "application/xml") ) 

    DEFINE orderList DYNAMIC ARRAY OF orderType 

        # ... function code to get orders ...
        RETURN orderList 
    END FUNCTION

Related concepts

Set a request body on page 3383
Functions that create or update a resource need to set a request body for the incoming payload. You specify the request body in an input parameter.

Set a response body and header on page 3384
Functions that create, update, delete, or get a resource need to set a response body. You specify this in a return parameter without an attribute. Other return values can be sent in headers, using the WSHeader attribute.

WSName
Specify alternative names for parameters in a REST function.

Syntax

WSName

Usage

Use this attribute to specify names for parameters in a REST function. In general, you do not need to use WSName. Consider using it to improve the readability of input parameters or to formats the data returned in the response.

WSName is an optional attribute.

Example 1: Providing an alternative name

You need to use WSName to provide the required name, if the input parameter name cannot be used as a variable name in your REST function.

For example, if you need to pass parameters for the Genero Application Server environment, you will need to use WSName. These variables (for example, X-FourJs-XXX) have hyphens, which are not allowed in Genero BDL variables. WSName allows you to use a parameter name that is BDL friendly but provide the required name of the variable in the header.

PUBLIC FUNCTION getRemoteAddress (remote_ip_addr STRING 
    ATTRIBUTE(WSHeader, WSOptional, WSName="X-FourJs-Environment-Variable-REMOTE_ADDR") )

In the HTTP request, there will be a header named "X-FourJs-Environment-Variable-REMOTE_ADDR", not "remote_ip_addr". In your function, you reference the variable using the "remote_ip_addr" variable name.

Example 2: Providing a user-friendly parameter name

You would consider setting the WSName attribute clause of a parameter, if the parameter name is not obvious or user-friendly.
For example, if the parameter references a field name in a database table that is abbreviated, or could be ambiguous in your function, WSName provides you with an option to provide a clearer name.

```plaintext
PUBLIC FUNCTION getUsers(reg STRING ATTRIBUTE(WSQuery, WSOptional, 
                          WSName="region"), 
                          ccode STRING ATTRIBUTE(WSHeader, WSName="country") ) 
```

In the HTTP request:

- There will be a header named "country", not "ccode".
- In the URL to call the REST function, you must use the value specified by WSName (not the parameter name) for the query:

  ```plaintext
  http://host:port/gas/ws/r/MyService/users?region=alsace
  ```

**Example 3: Formatting output data**

You can use WSName to format the output data. It allows you to override the default naming convention ("rv0", "rv1", etc., ) that is otherwise output at runtime. By specifying a more recognizable name for the data, you improve the readability of the output.

In the example, the attribute names the array "Users". This name will appear as the root path for the output in JSON, or the root tag for the output in XML.

```plaintext
TYPE userType RECORD ... END RECORD 

PUBLIC FUNCTION getUsers()
  ATTRIBUTES (WSGet, 
              WSPath="/users", 
              WSDescription="Get users resource ", 
              WSThrows="400:Invalid,404:NotAvailable" ) 
  RETURNS (DYNAMIC ARRAY ATTRIBUTE(WSName = "Users") OF 
            userType ATTRIBUTES(WSMedia = "application/json, application/xml"))

  DEFINE regionList DYNAMIC ARRAY OF userType 
  # ... function code ... 
  RETURN regionList 
END FUNCTION
```

**Related concepts**

- Customize XML serialization with WSName and XMLName on page 3394
- Use the WSName and XMLName attributes to customize serialization at runtime and improve the readability of the XML output.

**WSOptional**

Qualifies a parameter as optional in a REST function.

**Syntax**

```plaintext
WSOptional
```

**Usage**

You use this attribute to specify a parameter in a REST function as optional. You set the WSOptional attribute in the ATTRIBUTE() clause of the parameter.
WSOptional can be set only with the following attributes:

- **WSQuery** on page 3471
- **WSHeader** on page 3472 (input, output)
- **WSCookie** on page 3473

**Example optional cookie**

```plaintext
IMPORT  com

TYPE profileType RECORD
   id INT,
   name VARCHAR(50),
   email VARCHAR(100),
   ccode VARCHAR(10)
   # ...
END RECORD

DEFINE myError RECORD ATTRIBUTE(WSError="My error")
   code    INTEGER,
   reason  STRING
END RECORD

PUBLIC FUNCTION getUsersByCountry(
   p_country STRING ATTRIBUTE(WSCookie, WSOptional, WSName = "Preferred_country")
) ATTRIBUTES (WSGet,
   WSPath="/users",
   WSDescription="Get users by country if the cookie is sent by client.",
   WSThrows="400:Invalid,404:NotAvailable"
) RETURNS (DYNAMIC ARRAY ATTRIBUTE(WSName="Users",WSMedia="application/xml,application/json") OF profileType)

   DEFINE countryList DYNAMIC ARRAY OF profileType
   IF p_country IS NULL THEN
      SELECT * FROM users
   ELSE
      SELECT * FROM users WHERE @ccode=p_country
   END IF
   #... function code ...
RETURN countryList
END FUNCTION
```

**File handling attributes**
Attributes that define file attachment handling in REST services and functions.

**WSAttachment**
Defines file attachments in the REST message.

**Syntax**

```plaintext
WSAttachment
```

**Usage**
You use this attribute to define attachment files. Files may be attached in the message response or request.
In order to send a file as an attachment, you can have only one input parameter in the request. You specify the absolute path to the file in this parameter.

The file attachment MIME type can be set by the WSMedia attribute. If WSMedia is not specified, the engine accepts it as the OpenAPI specification for all file types, "/*".

For instance, if a text file is to be returned as an attachment, you handle this by adding the attribute WSAattachment to a STRING parameter and by setting the WSMedia attribute to MIME type "text/plain". You must also provide the absolute path to the file.

**Example WSAattachment with WSMedia**

```plaintext
PUBLIC FUNCTION Get()
    ATTRIBUTES(WSGet)
    RETURNS (STRING ATTRIBUTE(WSAttachment, WSMedia="text/plain") )
        # ... function code ...
        RETURN "/usr/tmp/myfile.txt"
END FUNCTION
```

The REST engine copies the file to the temporary directory defined by the TMP environment variable.

**Note:** The file is removed from the temporary directory at the end of the REST operation to avoid a build-up of files on your disk.

If you are attaching an image, it is defined in WSMedia with a wildcard to allow for all image types:

```plaintext
input STRING ATTRIBUTE (WSAttachment, WSMedia="image/*")
```

If the file can be any type, WSMedia is not specified.

```plaintext
input STRING ATTRIBUTE (WSAttachment)
```

The payload format is chosen according to that specified in the Content-Type header received from the client using the Web service. You code in your function to replace the image/* placeholder in the Content-Type header with the actual value. This can be done, for instance, using the WSCContext attribute, to set the header.

**Related concepts**

- **Handling file attachments with REST** on page 3386
  The Genero REST high-level framework provides two mechanisms for handling attachments.

- **Multipart requests or responses** on page 3394
  In GWS REST there is support for the standard multiple part message, in which more than one different sets of data are combined in a single body.

**WSMedia**

Defines the data format of the REST message.

**Syntax**

```
WSMedia = " MIME-type [, ...]"
```

Where WSMedia is a comma-separated list of MIME types:

1. **MIME-type** is a supported data format type.

**Usage**

You use this attribute to define the supported data format of the message. You set the WSMedia attribute in the ATTRIBUTE() clause of variables and/or on input and output function parameters.
Example WSMedia with record variable

In this example a record is defined with a list of MIME types with WSMedia to allow the function handle different formats. The payload format is chosen according to that specified in the Accept header received from the client using the Web service.

```plaintext
PUBLIC DEFINE myRecord RECORD
  ATTRIBUTE (WSMedia=
    "application/json, application/xml, application/x-www-form-urlencoded")
    a INTEGER,
    b FLOAT,
    c STRING
END RECORD
```

Table 730: Supported MIME types on page 3479 outlines the data formats supported via the WSMedia attribute with corresponding default data types where applicable:

<table>
<thead>
<tr>
<th>MIME type</th>
<th>Default data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>application/json</td>
<td>Record and array</td>
</tr>
<tr>
<td>application/xml</td>
<td>Record and array</td>
</tr>
<tr>
<td>application/x-www-urlencoded</td>
<td>Record (only)</td>
</tr>
<tr>
<td>text/plain</td>
<td>Simpletype</td>
</tr>
<tr>
<td>application/octet-stream</td>
<td>BYTE</td>
</tr>
<tr>
<td>image/*</td>
<td>No default type</td>
</tr>
</tbody>
</table>

The actual format is chosen according to that specified in the Accept or Content-Type headers of HTTP requests and responses. See the example in WSContext on page 3481.

There is an order of precedence that defaults to the first MIME type listed in WSMedia when the Accept or Content-Type header is not present in the request. The format chosen can also depend on the data type used, for defaults, see Table 730: Supported MIME types on page 3479.

WSMedia and file attachments

If a file is to be attached, you can handle this through a parameter with a WSAAttachment on page 3477 and a WSMedia attribute. The REST engine is designed to accept the parameter value then as a path to a file to be attached. For instance, if the file is an image, it is defined with a wildcard to allow for all image types:

```plaintext
input STRING ATTRIBUTE (WSAttachment, WSMedia="image/*")
```

WSMedia is an optional attribute.

Related concepts

Handling file attachments with REST on page 3386
The Genero REST high-level framework provides two mechanisms for handling attachments.

Multipart requests or responses on page 3394
In GWS REST there is support for the standard multiple part message, in which more than one different sets of data are combined in a single body.

HTTP code attributes
Attributes that define HTTP code in the response from a REST functions.
WSRetCode
Defines the HTTP status sent in the REST response.

Syntax

```
WSRetCode = "↓ code ↓ code:description ↓ "
```

Where WSRetCode is a string enclosed in quotations and where:

1. `code` is an HTTP code representing successful status in the range 200 to 399.
2. `description` provides a description.

**Important:** If you use the `code` option without a description, the default fault description defined in RFC 2616 is returned to the client. If you use the `code` option with a description, there is no space before or after the colon (`:`).

**Usage**

You use WSRetCode in the ATTRIBUTES() clause of the REST function to specify the HTTP protocol status in the response. The HTTP code and HTTP description you provide is sent in the HTTP protocol response.

**Example using WSRetCode in REST function**

```
PUBLIC FUNCTION Add(
   a INTEGER ATTRIBUTE(WSQuery),
   b INTEGER ATTRIBUTE(WSQuery) )
ATTRIBUTES(WSGet,
   WSRetCode = "215:correct"
) 
RETURNS INTEGER
RETURN a + b
END FUNCTION
```

In this trivial example, instead of "200 OK", the HTTP response is changed to "215 correct".

**Figure 142: HTTP response using WSRetCode**

**Context handling attributes**
Attributes that defines the context for REST services and functions.
**WSContext**
Defines an injection variable to retrieve REST operation context values at the service level.

**Syntax**

```plaintext
WSContext
```

**Usage**
You can use it to retrieve REST operation context such as BaseURL, Media, and Scope, or you can set a default context for the Content-Type header if the WSMedia on page 3478 attribute value contains a wildcard.

You need to define a DICTIONARY variable in your REST module that specifies WSContext in an ATTRIBUTE() clause.

**Example definition for WSContext dictionary**

**Important:** The context variable needs to be a modular variable.

```plaintext
PRIVATE DEFINE Context DICTIONARY ATTRIBUTE(WSContext)OF STRING
```

The GWS engine sets the dictionary key/values before the function is executed. For examples of use in your REST function, see Table 731: Context dictionary on page 3482.
## Table 731: Context dictionary

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Media</strong></td>
<td>Provides one of the media values set in the <code>WSMedia</code> attribute of the input parameter. Or it provides the defaults if there is no attribute, or what the REST engine has chosen when several are possible.</td>
</tr>
<tr>
<td><strong>BaseURL</strong></td>
<td>Provides the base URL of the service when executed. For example, the URL is not the same in standalone as behind a GAS.</td>
</tr>
<tr>
<td><strong>Scope</strong></td>
<td>Provides the valid scope (if there is one) that grants access to the REST function.</td>
</tr>
<tr>
<td><strong>Content-Type</strong></td>
<td>Provides the real MIME-type the REST function returns if the <code>WSMedia</code> attribute value contains &quot;image/*&quot;.</td>
</tr>
</tbody>
</table>

**Example usage**

### Media

**DISPLAY**

```
context["Media"]
```

This may display, for instance, application/xml. If, for instance, `WSMedia` is set with a list of mime types, ("application/json,application/xml"), you can know the exact media type requested in the current execution of the function.

### BaseURL

**DISPLAY**

```
context["BaseURL"]
```

This displays the service's BaseURL, such as: HTTPS:// host:port/gas/ws/r/xcf/Account if the service has been registered as "Account".

This may be useful if your REST function has to query another function, for instance, and when the exact URL is needed.

### Scope

**DISPLAY**

```
context["Scope"]
```

This may display "profile.read".

If, for instance, `WSScope` on page 3486 has been set in the function with a list of scopes (for example, "profile.read, profile.write, profile.mgr"), it may be useful to know which one really applies in the current function execution.

**Important:** This is the only context dictionary value you can set at runtime.

**LET**

```
LET context["Content-Type"]="image/jpeg"
```

For example, this statement will set the returned `Content-Type` header to "image/jpeg" based on the media type requested at runtime.

*WSContext* is an optional attribute.
**WSContext with Content-Type set at runtime**

If the `WSMedia` return parameter attribute has a list of values or a wildcard value, "image/*", you code in a function to set the `Content-Type` header for the actual value returned at runtime in the response. Table 732: HTTP Header on page 3483 outlines options for defining the `Content-Type` header for the image.

**Table 732: HTTP Header**

<table>
<thead>
<tr>
<th>Header Name</th>
<th>Description</th>
<th>Required</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content-Type</td>
<td>The format of the image to upload.</td>
<td>Optional</td>
<td><code>image/jpeg</code>, or <code>image/png</code>, or <code>image/gif</code>, and so on.</td>
</tr>
</tbody>
</table>

**Example WSContext with Content-Type set at runtime**

```plaintext
PRIVATE DEFINE Context DICTIONARY ATTRIBUTE(WSContext) OF STRING
PUBLIC FUNCTION getImage( id INTEGER ATTRIBUTE(WSParam) )
  ATTRIBUTES (WGET,
    WSPartPath="/photos/{id}" )
  RETURNS STRING ATTRIBUTE (WSAttachment, WSMedia="image/*")
  DEFINE image BYTE
  DEFINE mime STRING
  # function code
  CASE mime
    WHEN "png"
      LET Context["Content-Type"]="image/png"
    WHEN "jpg"
      LET Context["Content-Type"]="image/jpg"
  END CASE
  RETURN image
END FUNCTION
```

**Error handling attributes**

Attributes that define error handling in REST services and functions.

**WSError**

Specifies a description of the status-code that is returned in the HTTP response.

**Syntax**

```plaintext
WSError = "description"
```

1. *description* defines a short description as the "Reason-Phrase" for the status-code that is returned in the HTTP response message.

**Usage**

You use this attribute in the management of application level errors in conjunction with `WSThrows` on page 3484. When you use `WSThrows` with the "code:@variable" option to return a description of the status-code instead of the standard HTTP status code description, you must reference a variable that has the `WSError` attribute.

You must set the attribute on a public variable defined at the modular level. You can define it as any Genero BDL simple or complex type depending on your requirements for the detail about the error. For example, if defined as a `RECORD` or `ARRAY`, by default, it will be serialized in `JSON/XML` on the client side. If you define it as a simple type, it will be serialized in `TEXT/PLAIN`.

You code in your REST function to set the variable at runtime with a specific reason for the HTTP status code. The response is sent in a call to the `com.WebServiceEngine.SetRestError` on page 2837 method referencing the variable.
When the client stub is generated from the OpenAPI specification of your Web service, the correct Genero BDL variable corresponding to the HTTP status code is deserialized on the client side.

**Example WSError in record variable**

```gen
PUBLIC DEFINE userError RECORD ATTRIBUTE(WSError="User error")
    message STRING
END RECORD
```

**Example WSError in integer variable**

```gen
PUBLIC DEFINE fatalError INTEGER ATTRIBUTE(WSError="fatal error")
```

WSError is an optional attribute.

**Related concepts**

- **Handling application level errors** on page 3399
- There are many situations in which you need to notify an error to a client using your REST Web service.

**WSThrows**

Defines a list of errors that may be encountered accessing a REST resource.

**Syntax**

```gen
WSThrows = "\{ code \| code: description \| code:@variable \} [...]
```

WSThrows is a comma-separated list of status codes and/or error descriptions where:

1. `code` is an HTTP status code in the range 400 to 599
2. `description` is text describing the error
3. `@variable` is a reference to a variable defined to handle error descriptions. The variable must be defined with a `WSError` attribute. It is returned in the response message body.

**Important:**

- There is no space before or after the colon (\:).
- If you use the `code` option on its own without a description, the default fault description defined in RFC 2616 is returned to the client.

**Usage**

You use this attribute to manage application level errors, such as a database record not found. If your function can potentially return one or more errors, you use `WSThrows` to declare those so that they can be generated in the OpenAPI specification for that resource.

For instance, HTTP status code 404 means resource not found, but using the `WSThrows` attribute you can respond to the client with `404: user id not found` in your function. To return a description of the status code like this, you must use `WSThrows` with the "code:@variable" option of the attribute, and reference a variable with the `WSError` attribute.

If you simply need to return an error code without a description, use `WSThrows` with `code`, or with a description use `code: description`.

For example, these are valid uses of `WSThrows`:

**Important:** You must use valid HTTP status codes. Use of HTTP status codes between 400 to 599 are valid, but use of codes between 200 to 399 representing successful HTTP status, results in `error-9117`.

```gen
WSThrows = "404, 402"
```

In this example, there are no descriptions of the error, just standard HTTP error messages (see RFC 2616) are returned and displayed on the client side.
WSThrows= "404:not found error, 402:hello world"

In this example, the content after the colon (:) is the description of the error displayed on the client side.

WSThrows= "404:@error1, 402:@error2"

Here error descriptions are provided in the WSErrcompanyror attributed variables referenced in @error1 and @error2.

If you need to ensure an error description other than the standard protocol message is returned to the client, you must use WSThrows with the "code:@variable" option. Otherwise the web server may replace your description with the standard HTTP error message.

WSThrows is an optional attribute.

Example function using WSThrows

In the example, the WSThrows attribute is set on the ATTRIBUTES() clause of the function. It has options to respond to a list of errors.

At runtime when errors are trapped, descriptions of the errors are generated. These are set in the myError and fatalError variables, which are defined with the WSErrcompanyror attribute. These will be returned in the message body.

A call is made to the SetRestError() method to return the HTTP status code, referencing the appropriate WSThrows option.

```plaintext
IMPORT com

TYPE profileType RECORD
  id INT,
  name VARCHAR(50),
  email VARCHAR(100)
END RECORD

PUBLIC DEFINE fatalError INTEGER ATTRIBUTE(WSErrcompanyror="fatal error")

PUBLIC DEFINE myError RECORD ATTRIBUTE(WSErrcompanyror="My error")
  code INTEGER,
  reason STRING
END RECORD

PUBLIC FUNCTION updateUser( thisUser profileType )
  ATTRIBUTES (WSPost,
    WSPa<e>sath.AddListener("/users",
    WSDescription ="Update user details",
    WSThrows ="400:@myError,
        412:@fatalError,406:should not happen" )
  RETURNS STRING
  DEFINE ret STRING
  TRY
    UPDATE users SET
      name = thisUser.name,
      email = thisUser.email
    WHERE @id = thisUser.id
    IF SQLCA.SQLCODE = 0 THEN
      LET ret = thisUser.id
    END if
  CATCH
    CASE
      WHEN SQLCA.SQLCODE = NOTFOUND
        LET myError.code = NOTFOUND
        LET myError.reason = SFMT("User with ID: %1 not found",thisUser.id)
  ```
CALL com.WebServiceEngine.SetRestError(400,myError) WHEN SQLCA.SQLCODE < 0
    LET fatalError = SQLCA.SQLCODE
    CALL com.WebServiceEngine.SetRestError(412,fatalError)
  OTHERWISE
    CALL com.WebServiceEngine.SetRestError(406,NULL)
END CASE
END TRY
RETURN ret
END FUNCTION

Related concepts
Handling application level errors on page 3399
There are many situations in which you need to notify an error to a client using your REST Web service.

Security handling attributes
Attributes that define security and information about the REST services and functions.

WSScope
Specify security in the REST service via scopes.

Syntax

   WSScope = " \{ scope \} , ..."  

Where WSScope is a comma-separated list of scopes and where:

1. scope defines access permission for the resource.

Usage
You use this attribute to specify security via scopes forwarded from the Genero Application Server to the GWS REST service.

WSScope is an optional attribute.

Note: Testing your services with WSScope:

- When testing your service in standalone mode without a GAS, the WSScope is not checked. However, when behind a GAS, the appropriate scope is required and you will need to deploy and secure the service with the Genero Identity Provider (GIP).
- Alternatively, if you need to integrate Genero REST services security into your own environment system, you can also write your own delegate service to validate any kind of token, extract the scope from it, and forward it to the REST service.

You can set the WSScope attribute in the service information record of the module at the service level or in the ATTRIBUTES() clause of the REST function.

Example using WSScope in the information record at the service level
When the scope is set in the service information record, all REST functions in the Web service are executed if, and only if, the request contains a scope definition that matches the value in the WSScope attribute.

PUBLIC DEFINE serviceInfo RECORD
   ATTRIBUTE(WSInfo, WSScope="users.myservice")
      title STRING,
      description STRING,
      termOfService STRING,
      contact RECORD
         name STRING,
         url STRING,
         email STRING
END RECORD,
  version STRING
END RECORD = (  
    title: "my service",
    version: "1.0",
    contact: ( email:"helpdesk@mysite.com") )

Note: The main purpose of the service information record (defined with the WSInfo attribute) is to document the REST service. If you are not setting the scope at the modular level here, the record is still needed to provide service information.

Example setting the WSScope in a REST function

If the call to a specific REST function requires user authentication, add the WSScope attribute in its ATTRIBUTES() clause.

For example, this function is only executed if the scope (WSScope) contains "dev.read". The GAS also needs to be started.

IMPORT com

TYPE profileType RECORD
  id INT,
  name VARCHAR(50),
  email VARCHAR(100)
END RECORD

PUBLIC
DEFINE myError RECORD ATTRIBUTE(WSError="My error")
  code INTEGER,
  reason STRING
END RECORD

PUBLIC FUNCTION getUsersList()
  ATTRIBUTES (WSGet,
    WSPath = "/users",
    WSDescription = "Get a list of all users, requires authentication",
    WSThrows = "400:Invalid,404:NotAvailable",
    WSScope = "dev.read")
  RETURNS (  
    DYNAMIC ARRAY ATTRIBUTE(WSName="Users",WSMedia="application/xml") OF profileType
    ATTRIBUTE(XMLName="User") )

DEFINE userList DYNAMIC ARRAY OF profileType
WHENEVER ERROR continue
DECLARE c2 CURSOR FOR SELECT * FROM users
WHENEVER ERROR STOP
CASE
WHEN SQLCA.SQLCODE = 0
  # ... function code ...
OTHERWISE
  INITIALIZE userList TO NULL
  LET myError.reason = SFMT("SQL error:%1 [\%2]",SQLCA.SQLCODE,
  SQLERRMESSAGE)
  CALL com.WebServiceEngine.SetRestError(400,myError)
END CASE
FREE c2
RETURN userList
END FUNCTION
How to determine the scope names

When determining the names for the scopes, it is important to understand the role of scopes. You create a scope in the Identity Provider (IdP) system and assign it to a group or a user, so that the user or group member will get an access token containing the scope name, and be allowed to access an operation that specifies the same scope name using the WSScope attribute.

The names you choose can be a simple name (such as "readonly"), or it can use dot notation (such as "readonly.dev" and "readonly.user") to provide a logical hierarchy. The hierarchy approach is optional; it is purely provided to allow you to organize your scopes in a logical manner that makes sense to you. Your end user would still need to belong to groups that have either "readonly.dev" or "readonly.user" scope assigned to them, the "dev" and "user" extensions do not in themselves have any meaning to the IdP.

For example, you could create a service access list with the server name of "ReadOnly", with two scopes defined: "ReadOnly.dev" and "ReadOnly.user". You would then set the ReadOnly.dev scope to the resources to be accessible to developers, and the ReadOnly.user scope to the resources to be accessible to users only.

When determining scope names, it may also be helpful to think of the overall solution, which can be a complex system with many services all working together. You can evaluate the access needs of the various services and operations, and then identify the list of scopes that would allow you to provide (or restrict) access to your various groups of users.

In summary, there is no restriction on the names you choose for scopes. You can set them as you wish, depending on what you want to achieve.

Related concepts
Handling security on page 3402
You handle security in GWS high-level REST via scopes.
Generate service description on demand on page 3406
You can access a description of your GWS REST Web service through a query string on the URL of the Web service.

WSInfo
Specify information about the Web service such as title, version, contact details, scopes, etc.

Syntax

```plaintext
WSInfo
```

Usage

You use this attribute to specify information that describes the service such as the service title, version, and contact details.

The WSInfo attribute must be set in an ATTRIBUTE() clause on the information record at the module level. This allows you to provide service information that is useful to clients.

WSInfo is an optional attribute.

Example using WSInfo in the service information record

The example service information record follows the OpenAPI standard.

Note: The GWS does not check the record structure used with WSInfo attribute. You are therefore allowed to add the fields you want to be available in the OpenAPI specification.
url STRING,
email STRING,
END RECORD,
version STRING
END RECORD
END RECORD = (title: "Officestore RESTful Services.", version: "3.0",
contact: (email: "helpdesk@mysite.com") )

Related concepts

Provide service information on page 3404

Provide information about the service, such as title, version, contact details, etc., that is generated in the OpenAPI specification file.

Generate service description on demand on page 3406

You can access a description of your GWS REST Web service through a query string on the URL of the Web service.

Web Services FGLPROFILE configuration

The configuration for the Genero Web Services is defined from entries in the FGLPROFILE file.

When using BDL Web Services on server side, it is the Web server that is in charge of the BDL Web Services server security, not the BDL server application itself. You must refer to your Web server manual to secure the server part of the Web Services.

**Note:** This is useful for deployment purposes, as no additional code modification is necessary, even if the location of the different servers change, or if different cryptography keys or X509 certificates are necessary for the same application but intended for several customers with their own needs.

The Genero Web Services secured communication and the support of XML-Security is based on the OpenSSL engine. It allows a BDL Web Services client, or a BDL application using the com or xml API, to communicate with any secured server over HTTP or HTTPS, and to handle encrypted and/or signed XML document in BDL coming from any other application.

FGLPROFILE entries for Web Services

The FGLPROFILE entries relating to Genero Web Services are divided between five categories: security, basic or digest HTTP authentication, proxy configuration, server configuration, and XML cryptography.

**HTTPS and password encryption**

The following table lists the FGLPROFILE file entries specifying the security certificates and algorithms the Web Services client uses for HTTPS and password encryption. These entries specify how an application using the low-level com or xml APIs performs secured communications.

**Important:** On iOS platform the security entries are not available because iOS SSL/TLS layer is used. However, you can configure the following security Web Services FGLPROFILE entries for GMI mobile devices: security.global.ca, security.global.ca.lookuppath, and security.global.systemca.

**Note:** Any entry defining a file on disk, for example security.global.ca, xml.keystore.calist, etc., can be set with a relative or an absolute path. If set as relative, the file is located based on the current execution directory. The recommended practice is to specify an absolute path in case fglrun is not always executed from the same directory.

**Table 733: Security Configuration FGLPROFILE entries**

<table>
<thead>
<tr>
<th>Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>security.global.script</td>
<td>File name of a script executed each time a password of a private key is required by the client. The security script accepts one argument corresponding to the file name of the private key for which the password is required, and must return the correct password or the client stops. For script examples, see <a href="#">Windows® Password Script Example</a> or <a href="#">UNIX™ Password Script Example</a>. This entry cannot be used if security.global.agent is set.</td>
</tr>
</tbody>
</table>
### Entry | Description
--- | ---
security.global.agent | Port number where the fglpass agent is waiting for requests. It returns the password that grants access to a private key when needed by a BDL application. The DVM and the fglpass agent perform authentication and exchange encrypted data over the local host network only. Refer to Using the password agent for details. This entry cannot be used if security.global.script is set.

security.global.agent.gid | Set this entry as true to specify agent authentication based on OS user group. This entry is only for UNIX® platforms. For more details, see Use the password agent on page 3292.

security.global.protocol | The SSL/TLS protocol to use for secured communications. Possible values are:
- TLSv1.2
- TLSv1.1
- TLSv1 (version 1.0)
- SSLv3
- SSLv23 (The default, enabling all supported protocols)

security.global.ca | File name of the Certificate Authority list, with the concatenated PEM-encoded third party X.509 certificates considered as trusted, and in order of preference.

security.global.ca.lookuppath | A list of directories containing certificate authorities. Genero Web Services will load the CA from the directories in this list. The entry is a list of directories separated by a semicolon.

security.global.windowsca | If set to true, build the Certificate Authority list from the Certificate Authorities stored in the Windows® key store. This entry is only valid on Windows® systems where security.global.ca is not set.

security.global.cipher | The list of encryption, digest, and key exchange algorithms the client is allowed to use during a secured communication. If this entry is omitted, all algorithms are supported. For more details about cipher, refer to www.openssl.org.

security.global.certificate | File name of the PEM-encoded client X.509 certificate to be used for any secured connection if not redefined in a specific server configuration.

security.global.privatekey | File name of the PEM-encoded private key associated to the above X.509 certificate and to be used for any secured connection if not redefined in a specific server configuration.

security.global.keysubject | The subject string of a X.509 certificate and its associated private key registered in the Windows® key store to be used for any secured connection if not redefined in a specific server configuration. This entry is valid only on Windows® systems.

security.global.systemca | When set to true, the Certificate Authority is loaded from the key store on Mac® OSX or Windows® systems, and from a predefined directory on Unix/Linux®. If set to false, the Certificate Authority is not loaded from the system-default location. Default is true. This entry is only valid where security.global.ca is not set.
### Entry Description

<table>
<thead>
<tr>
<th>Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>security.global.ocsp.enable</td>
<td>If set to <code>true</code>, once the server has been validated against local certificate authority, an additional request is performed to the certificate issuer's URL to ensure that no certificate has been revoked at time of connection. Default value is <code>false</code> (no additional request is done)</td>
</tr>
<tr>
<td>security.global.ocsp.url</td>
<td>Instead of checking revocation to the URL inside the issuer's certificate, you can specify a fixed URL where all OCSP requests will be sent. For example: security.global.ocsp.url = &quot;http://any_url&quot; By default, this entry is not set and the URL inside the certificate is used.</td>
</tr>
<tr>
<td>security.idsec.certificate</td>
<td>File name of the PEM-encoded client X.509 certificate.</td>
</tr>
<tr>
<td>security.idsec.privatekey</td>
<td>File name of the PEM-encoded private key associated to the above X.509 certificate. If the PEM file is password protected, you need to use the security.idsec.script or security.idsec.agent entries to supply the passphrase to decrypt the private key.</td>
</tr>
<tr>
<td>security.idsec.keysubject</td>
<td>The subject string of a X.509 certificate and its associated private key registered in the Windows® key store. This entry is valid only on Windows® systems.</td>
</tr>
</tbody>
</table>

**Note:**

1. The `idsec` keyword must be replaced with your own identifier, and all necessary entries must be set. See FGLPROFILE setting.
2. If an entry is defined more than once, only the last occurrence is taken into account.

**Related tasks**

Configure a WS client to access an HTTPS server on page 3305
Configuration steps to access a server in HTTPS.

**Basic or digest HTTP authentication**

The following table lists the FGLPROFILE entries that specify the login and password to use in the case of HTTP authentication to a server or a proxy. The entries also specify the login and password to use in an application using the low-level com or xml API.
### Table 734: HTTP basic or digest Authentication FGLPROFILE entries

<table>
<thead>
<tr>
<th>Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>authenticate.idauth.login</td>
<td>The login identifying the client to a server during HTTP Authentication.</td>
</tr>
<tr>
<td>authenticate.idauth.password</td>
<td>The password validating the login of a client to a server during HTTP Authentication. As passwords are never recommended to be in clear text, you must encrypt them with the fglpass tool. For more information, see FGLPROFILE password encryption.</td>
</tr>
<tr>
<td>authenticate.idauth.realm</td>
<td>The string identifying the server to the client during HTTP Authentication. If the string does not match the server's string, authentication fails. This parameter is optional, but it is recommended that you check the server identity, especially if the server's location is suspicious.</td>
</tr>
<tr>
<td>authenticate.idauth.scheme</td>
<td>One of the following strings representing the different HTTP Authentication mechanisms.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Anonymous</strong> (default value) - The client does not know anything about the server, and performs a first request to retrieve the server authentication mechanism. It then uses the login and password to authenticate to the server using the Basic or Digest mechanism, depending on the server's returned value.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Basic</strong> - The client authenticates itself to the server at first request, by sending the login and the password using the Basic authentication mechanism.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Digest</strong> - The client performs a first request without any login and password, to retrieve the server information before authenticating itself to the server in a second request using the Digest mechanism.</td>
</tr>
</tbody>
</table>

**Note:**

1. The `idauth` keyword must be replaced with your own identifier, and all necessary entries must be set. See FGLPROFILE setting.
2. If an entry is defined more than once, only the last occurrence is taken into account.

**Related concepts**

- **Accessing secured services** on page 3303
  Security and authentication are important. Genero Web Services provides various communications options for a client to connect to a Web service.

- **Encryption and authentication** on page 3299
  A scenario involving a person (Georges) and his bank guides you through the concepts of secured communication, certificates, and certificate authorities.

**Proxy configuration**

The following table lists the FGLPROFILE entries that specify how the Web Services client communicates with a proxy. The entries specify the way an application using the low-level `com` or `xml` API communicates with a proxy.
Table 735: Proxy Configuration FGLPROFILE entries

<table>
<thead>
<tr>
<th>Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>proxy.http.location</td>
<td>Location of the HTTP proxy defined as host:port or ip:port. If the port is omitted, the port 80 is used.</td>
</tr>
<tr>
<td>proxy.http.list</td>
<td>The list of beginning host names, separated with semicolons, for which the Web Services client does not go via the HTTP proxy.</td>
</tr>
<tr>
<td>proxy.http.authenticate</td>
<td>The authenticate.idauth the Web Services client uses to authenticate itself to the HTTP proxy.</td>
</tr>
<tr>
<td>proxy.https.location</td>
<td>Location of the HTTPS proxy defined as host:port or ip:port. If the port is omitted, the port 443 is used.</td>
</tr>
<tr>
<td>proxy.https.list</td>
<td>The list of host names, separated with semicolons, for which the Web Services client does not go via this HTTPS proxy.</td>
</tr>
<tr>
<td>proxy.https.authenticate</td>
<td>The authenticate.idauth the Web Services client uses to authenticate itself to the HTTPS proxy.</td>
</tr>
</tbody>
</table>

**Note:** If an entry is defined more than once, only the last occurrence is taken into account.

**IPv6 configuration**

The following table lists the FGLPROFILE entries that specify how the Web Services client uses the IPv6 network protocol.

Table 736: IPv4 and IPv6 FGLPROFILE entries

<table>
<thead>
<tr>
<th>Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ip.global.version</td>
<td>Defines the IP version to be used. Possible values are &quot;4&quot; (IPv4) or &quot;6&quot; (IPv6). By default, when this entry is not defined, the WS library will try to use IPv6 and fallback to IPv4, depending on the operating system.</td>
</tr>
<tr>
<td>ip.global.v6.interface.name</td>
<td>Defines the name of the network interface to be used for IPv6 link-local addresses. For example, this entry can get values such as &quot;eth0&quot;, &quot;en0&quot;, &quot;ethernet_5&quot;.</td>
</tr>
<tr>
<td><strong>Important:</strong> This entry is not supported on Microsoft™ Windows® platforms.</td>
<td></td>
</tr>
<tr>
<td>ip.global.v6.interface.id</td>
<td>Defines the id of the network interface to be used for IPv6 link-local addresses. For example, this entry can get values such as &quot;1&quot;, &quot;2&quot;, &quot;11&quot;.</td>
</tr>
</tbody>
</table>

**Note:** If an entry is defined more than once, only the last occurrence is taken into account.

**Server configuration**

The following table lists the FGLPROFILE file entries that specify the correct way a Web Services client connects to an end point (usually a server).

**Note:** The entries specify also the way an application using the low-level com or xml API connects to an end point.
### Table 737: Server Configuration FGLPROFILE entries

<table>
<thead>
<tr>
<th>Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ws.idws.url</td>
<td>The end point URL of the server. The entry overwrites the global definition.</td>
</tr>
<tr>
<td></td>
<td>See Using * wildcards in server URLs on page 3498.</td>
</tr>
<tr>
<td>ws.idws.regex.url</td>
<td>A regular expression to define all possible URLs that can be used in this server configuration.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> If the <code>ws.idws.url</code> is defined, the <code>regex.url</code> entry is ignored.</td>
</tr>
<tr>
<td></td>
<td>This regex entry follows W3C rules as described in <a href="https://www.w3.org/TR/xmlschema-2/#regexs">https://www.w3.org/TR/xmlschema-2/#regexs</a>.</td>
</tr>
<tr>
<td></td>
<td>See Using regular expressions in server URLs on page 3498.</td>
</tr>
<tr>
<td>ws.idws.cipher</td>
<td>The list of encryption, digest and key exchange algorithms the client is allowed to use during a secured communication to that server. The entry overwrites the global definition.</td>
</tr>
<tr>
<td>ws.idws.verifyserver</td>
<td>If set to <code>true</code>, the client performs a strict server identity validation. If not fulfilled, it stops the communication; otherwise no server identity verification is performed. The default value is <code>true</code>.</td>
</tr>
<tr>
<td>ws.idws.security</td>
<td>The security identifier (<code>security.idsec</code>) the client uses to perform HTTPS communication with the server. See also Add configuration entries in your FGLPROFILE file on page 3309.</td>
</tr>
<tr>
<td>ws.idws.authenticate</td>
<td>The <code>authenticate.idauth</code> the client uses to authenticate itself to the server.</td>
</tr>
</tbody>
</table>

### Note:

1. The `idws` keyword must be replaced with your own identifier. All necessary entries, depending on the remote server's configuration, must be set. See FGLPROFILE setting.
2. You can use the unique identifier in the `.4gl` code instead of the server URL, with the `alias://` prefix. For example, `alias://idws`.
3. If an entry is defined more than once, only the last occurrence is taken into account.

### XML configuration

The following table lists the FGLPROFILE entries that control XML to Genero values conversion, and XML cryptography key or certificate mapping.
Table 738: XML configuration FGLPROFILE entries

<table>
<thead>
<tr>
<th>Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xml.keystore.caclist</code></td>
<td>The list of PEM-encoded third party X.509 certificates, separated with semicolons, of the Certificate Authority considered as trusted, in order of preference.</td>
</tr>
<tr>
<td><code>xml.keystore.x509list</code></td>
<td>The list of PEM-encoded third party X.509 certificates, separated with semicolons, to be used to find out the correct X.509 certificate when getting an incomplete one in a XML signature or an encrypted XML document.</td>
</tr>
<tr>
<td><code>xml.idxml.key</code></td>
<td>The file name of a cryptography key. For instance <code>RSA.pem</code>, <code>DSA.der</code> or <code>HMAC.bin</code>.</td>
</tr>
<tr>
<td><code>xml.idxml.x509</code></td>
<td>The file name of a cryptography x509 certificate. For instance <code>Cert.crt</code>.</td>
</tr>
<tr>
<td><code>xml.serializer.supportEmptyStrings</code></td>
<td>Controls empty string XML nodes conversion to Genero STRING values. The default is false, empty XML tags are converted to NULL. If set to true, an empty XML tag is converted to an empty STRING value. As result, in Genero, the LENGTH() function will return zero and the IS NULL comparison operator will evaluate to FALSE. Note that this entry only works for the STRING data type, and if the tag is not present, the STRING is set to NULL.</td>
</tr>
<tr>
<td>`xml.signature.prefix = { &quot;prefix&quot;</td>
<td>&lt;none&gt; }`</td>
</tr>
<tr>
<td>`xml.encryption.prefix = { &quot;prefix&quot;</td>
<td>&lt;none&gt; }`</td>
</tr>
</tbody>
</table>

**Note:**

1. The `idxml` keyword must be replaced with your own identifier. See FGLPROFILE: XML cryptography on page 3497.
2. You can use the unique identifier in the `.4gl` code instead of the file name.
3. If an entry is defined more that once, only the last occurrence is taken into account.

**HTTP configuration**

The following table lists the FGLPROFILE entries control options for the HTTP protocol.
### Table 739: HTTP Configuration FGLPROFILE entries

<table>
<thead>
<tr>
<th>Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>http.global.request.date</td>
<td>This entry applies to all <code>HTTPRequest</code> objects created in a program. The HTTP RFC 2616 specifies that a Date header should not be sent, if the HTTP request does not contain a body.</td>
</tr>
<tr>
<td></td>
<td>With the <code>http.global.request.date</code> FGLPROFILE entry, you can control this as follows:</td>
</tr>
<tr>
<td></td>
<td>• By default (no entry), or if <code>http.global.request.date = false</code>, no HTTP Date header is sent for GET, HEAD and DELETE requests.</td>
</tr>
<tr>
<td></td>
<td>• When <code>http.global.request.date = true</code>, an HTTP Date header is sent for any GET, HEAD, DELETE requests.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> For PUT and POST requests, the HTTP Date header is always sent.</td>
</tr>
</tbody>
</table>

**Note:** If an entry is defined more than once, only the last occurrence is taken into account.

### Examples

**SSL/TLS configuration examples.**

**Windows® BAT script for private key password**

Windows® BAT script sample returning a password, depending on the `.pem` file passed as parameter.

```batch
@echo off
REM -- Windows password script
IF "%1" == "Cert/MyPrivateKeyA.pem" GOTO KeyA
IF "%1" == "Cert/MyPrivateKeyB.pem" GOTO KeyB
GOTO end
:KeyA
ECHO PasswordA
GOTO end
:KeyB
ECHO PasswordB
GOTO end
:end
GOTO :EOF
```

**UNIX™ shell script for private key password**

UNIX® shell script sample returning a password, depending on the `.pem` file passed as parameter.

```bash
# UNIX password script
if [ "$1" == "Cert/MyPrivateKeyA.pem" ]
then
  echo PasswordA
fi
if [ "$1" == "Cert/MyPrivateKeyB.pem" ]
then
  echo PasswordB
fi
```

**FGLPROFILE: HTTP(S) Proxy Authentication**

FGLPROFILE entries can be used to define a connection to an HTTPS server via a proxy, and with HTTP and Proxy Authentication.

```ini
# Security configuration
security.global.script   = "Cert/password.sh"
security.global.ca       = "Cert/CAList.pem"
```
security.global.cipher = "HIGH" # Use only HIGH encryption ciphers
security.mykey.certificate = "Cert/MyCertificateA.crt"
security.mykey.privatekey = "Cert/MyPrivateKeyA.pem"

# Proxy HTTP Authentication
authenticate.proxyauth.login = "myapplication"
authenticate.proxyauth.password = "mypswd"
authenticate.proxyauth.scheme = "Basic"

# HTTPS Proxy configuration
proxy.https.location = "10.0.0.170"
proxy.https.list = "www.mycompany.com;www.mycompany.com"
proxy.https.authenticate = "proxyauth"

# Server HTTP Authentication
authenticate.serverauth.login = "mylogin"
authenticate.serverauth.password = "password"

# Server configuration
ws.myserver.authenticate = "serverauth"
ws.myserver.security = "mykey"

---

**Related concepts**

**Encryption, BASE64 and password agent with fglpass tool** on page 3290
Genero Web Services supports password encryption with fglpass as password agent.

**Accessing secured services** on page 3303
Security and authentication are important. Genero Web Services provides various communications options for a client to connect to a Web service.

**Related reference**

**Basic or digest HTTP authentication** on page 3491
**Proxy configuration** on page 3492

**FGLPROFILE: XML cryptography**
Use FGLPROFILE file entries to define XML cryptography and use the fglpass agent to get the private key passwords.

---

# Security configuration
security.global.agent = 4444

# Crypto configuration
xml.keystore.calist = "RSARootCertificate.crt;DSARootCertificate.crt"
xml.keystore.x509list = "RSA1024Certificate.crt;DSA1024Certificate.crt"
xml.id1.x509 = "RSA1024Certificate.crt"
xml.id2.x509 = "DSA1024Certificate.crt"
xml.id3.key = "RSA1024Key.pem"
xml.id4.key = "DSA1024Key.der"
xml.id5.key = "HMAC.bin"

---

**Related concepts**

**Encryption, BASE64 and password agent with fglpass tool** on page 3290
Genero Web Services supports password encryption with fglpass as password agent.

**Related reference**

**XML configuration** on page 3494
**FGLPROFILE: Server URL patterns**

FGLPROFILE entries can be used to define multiple server URLs, by using URL patterns.

**Understanding URL patterns**

By using URL patterns, you can create a URL base that applies to multiple server applications.

URLs matching the pattern can share server configuration (such as authentication and HTTPS).

FGLPROFILE entries using URL patterns apply to the following APIs:

- `com.HTTPRequest.Create()`
- `xml.DomDocument.load()`
- `xml.DomDocument.save()`
- `xml.StaxReader` methods
- `xml.StaxWriter` methods
- `com.TCPRequest.create()`

**Using * wildcards in server URLs**

To create a URL base, add a wildcard (`/*`) to the end of a URL in the FGLPROFILE file entry. A server application that starts with this URL (and that is not explicitly defined elsewhere) shares the configuration with other applications that also start with the same base URL. If an application has its own server configuration explicitly defined, it uses its specific entries instead of those defined by the wildcard configuration.

Consider this excerpt from a hypothetical FGLPROFILE file:

```plaintext
authenticate.auth.login    = "xxx"
authenticate.auth.password = "yyy"
authenticate.auth.scheme   = "Basic"
security.sec.certificate = "client.crt"
security.sec.privatekey   = "client.pem"

ws.myapp.url             = "http://mycompany.com/sample/*"
ws.myapp.authenticate    = "auth"
ws.myapp.security        = "sec"

ws.thirdapp.url          = "http://mycompany.com/sample/application3"
ws.thirdapp.authenticate = "auth3"
authenticate.auth3.login = "aaa"
authenticate.auth3.password = "bbb"
authenticate.auth3.scheme = "Basic"
```

Based on this example:

- Requests to "http://mycompany.com/sample/application1" and "http://mycompany.com/sample/demos/shoppingcart" use the same authentication and HTTPS configuration.
- A request to "http://mycompany.com/sample/application3" uses its specific authentication "auth3". No security configuration is defined for this URL, nor does it fall back on the shared security configuration defined for the base URL.

**Using regular expressions in server URLs**

To create a URL regex pattern, use the `ws.idws.regex.url` FGLPROFILE file entry, and define a regular expression that matches all server URLs to be associated with the "idws" server configuration.

**Note:** If the `ws.idws.url` is defined, the `regex.url` entry is ignored.

```plaintext
authenticate.auth.login    = "xxx"
```
Related reference

Server configuration on page 3493

XML to BDL serialization options

By default, Genero Web Services XML to BDL serialization will raise errors, if the XML content does not match the BDL variable receiving the data.

To relax the serialization, you can specify the options `xml_ignoreunknownattributes` and `xml_ignoreunknownelements` to the `xml.Serializer` class.

For example, if the BDL variable is defined as follows:

```plaintext
DEFINE data RECORD ATTRIBUTES(XMLName="Demo"),
    val1 INTEGER ATTRIBUTES(XMLName="Value1"),
    val2 STRING ATTRIBUTES(XMLName="Value2"),
    attr INTEGER ATTRIBUTES(XMLAttribute,XMLName="MyAttr")
END RECORD
```

The following XML document will by default raise conversion errors because the "badAttr" attribute and the "Unexpected" node are not defined in the corresponding BDL variable:

```xml
<Demo MyAttr="hello" badAttr= "BAD">
    <Value1>128</Value1>
    <Unexpected>Will be ignored</Unexpected>
    <Value2>Hello</Value2>
</Demo>
```

To avoid the conversion error, set the following options:

```plaintext
CALL xml.Serializer.setOption( "xml_ignoreunknownattributes", TRUE )
CALL xml.Serializer.setOption( "xml_ignoreunknownelements", TRUE )
... 
CALL xml.Serializer.DomToVariable( doc, data )
```

Note: The options to relax the serialization cannot apply in complex cases, and the conversion error may still occur.

For more details, see `xml.Serializer.SetOption` on page 3043.

WSHelper library

The WSHelper library.

The WSHelper file

It is recommended that the `WSHelper.42m` library file is linked into every Genero Web Services Server or Client program. It is located in `$FGLDIR/lib`.

```plaintext
authenticate.auth.password = "yyy"
authenticate.auth.scheme = "Basic"

ws.myident.regex.url = "http://.*\strasbourg\4js\com:[0-9]{4}/.*"
ws.myident.authenticate = "auth"
```
**WSHelper variables and records**
The WSHelper library provides public variables and records.

**WSQueryType**
The WSHelper library defines `WSHelper.WSQueryType` as a dynamic array:

```plaintext
PUBLIC TYPE WSQueryType DYNAMIC ARRAY OF RECORD
    name STRING,
    value STRING
END RECORD
```

The fields of the record are:
1. `name` is the name of the query argument.
2. `value` is the value of the query argument.

**WSServerCookiesType**
The WSHelper library defines `WSHelper.WSServerCookiesType` as a dynamic array:

```plaintext
PUBLIC TYPE WSServerCookiesType DYNAMIC ARRAY OF RECORD
    name STRING, # Cookie name
    value STRING, # Cookie value
    path STRING, # Cookie path (or null)
    domain STRING, # Cookie domain (or null)
    expires DATETIME YEAR TO SECOND, # Cookie expiration date (or null)
    httpOnly BOOLEAN,
    secure BOOLEAN
END RECORD
```

The fields of the record are:
1. `name` is the name of the cookie to be set. This field is mandatory. It will be URL-encoded on the wire.
2. `value` is the value of the cookie to be set. This field is mandatory. It will be URL-encoded on the wire.
3. `path` is the main path the cookie has to be set for. Any path containing that main path will then return the cookie. If no name is set, the cookie will be set by the path the client has provided.
4. `domain`: the domain (hostname) or sub-domain (ex: .strasbourg.4js.com) the cookie will be set on client side. If not set, the domain will be the hostname provided by the client.
5. `expires`: a `DATETIME YEAR TO SECOND` (on local time) from when that cookie will expire, and thus not be sent by the client anymore. If `NULL`, the cookie is called session cookie and will be sent as long as the client keeps the session open.
6. `http_only` is set to `TRUE` if the cookie is only for the HTTP layer, `FALSE` if cookie can be accessible in JavaScript.
7. `secure`: true if cookie must only be sent in HTTPS, false for HTTP and HTTPS.
**WSHelper APIs**
The WSHelper library provides Genero BDL HTTP helper functions.

### Table 740: WSHelper functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SplitUrl(url STRING)</code></td>
<td>Splits a complete URL string into pieces.</td>
</tr>
<tr>
<td></td>
<td>RETURNS (scheme STRING, host STRING, port STRING, path STRING, query STRING)</td>
</tr>
<tr>
<td><code>FindQueryStringValue(query STRING, name STRING)</code></td>
<td>Get a query string value by name.</td>
</tr>
<tr>
<td></td>
<td>RETURNS STRING</td>
</tr>
<tr>
<td><code>SplitQueryString(query STRING)</code></td>
<td>Splits the query string of an URL into an array or key-value pairs.</td>
</tr>
<tr>
<td></td>
<td>RETURNS WSHelper.WSQueryType</td>
</tr>
</tbody>
</table>

**WSHelper.FindQueryStringValue**
Get a query string value by name.

**Syntax**

```gdsl
FindQueryStringValue(query STRING, name STRING)
RETURNS STRING
```

1. `query` is the query string where to look for a value.
2. `name` is the name of the value wanted.

**Usage**
Get a query string value by name.

In case of error, a NULL value will be returned.

**WSHelper functions example**

```gdsl
IMPORT FGL WSHelper

MAIN

DEFINE val, scheme, host, port, path, query STRING
DEFINE ind INTEGER
DEFINE ret WSHelper.WSQueryType

CALL WSHelper.SplitQueryString("name1=val1&name2=val2&name3=val3")
    RETURNING ret

FOR ind = 1 TO ret.getLength()
```

DISPLAY "Query",ind
DISPLAY " name is",ret[ind].name
DISPLAY " value is ",ret[ind].value
END FOR
LET val = WSHelper.FindQueryStringValue("name1=val1&name2=val2","name1")
CALL WSHelper.SplitUrl("https://cube.strasbourg.4js.com:3128/GWS-492/TestSplitURL/test1")
  RETURNING scheme, host, port, path, query
END MAIN

Output:

<table>
<thead>
<tr>
<th>Query</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>name is</td>
<td>name1</td>
</tr>
<tr>
<td>value is</td>
<td>val1</td>
</tr>
<tr>
<td>Query</td>
<td>2</td>
</tr>
<tr>
<td>name is</td>
<td>name2</td>
</tr>
<tr>
<td>value is</td>
<td>val2</td>
</tr>
<tr>
<td>Query</td>
<td>3</td>
</tr>
<tr>
<td>name is</td>
<td>name3</td>
</tr>
<tr>
<td>value is</td>
<td>val3</td>
</tr>
</tbody>
</table>

**WSHelper.SplitQueryString**
Splits the query string of an URL into an array or key-value pairs.

**Syntax**

```
SplitQueryString(
  query STRING )
RETURNS WSHelper.WSQueryType
```

1. *query* is the query string to be split into a dynamic array.

**Usage**

Split a given query string into a dynamic array of key-value pairs, defined as WSHelper.WSQueryType.

The pieces are returned in dynamic array of key-value pairs defined as a WSHelper.WSQueryType. See **WSHelper variables and records** on page 3500 for more information regarding **WSQueryType**.

NULL may be returned if a value is not found.

In case of error, a NULL value will be returned.

**WSHelper functions example**

```
IMPORT FGL WSHelper
MAIN
DEFINE val, scheme, host, port, path, query STRING
DEFINE ind INTEGER
DEFINE ret WSHelper.WSQueryType
CALL WSHelper.SplitQueryString("name1=val1&name2=val2&name3=val3")
  RETURNING ret
FOR ind = 1 TO ret.getLength()
  DISPLAY "Query",ind
```
DISPLAY " name is",ret[ind].name  
DISPLAY " value is ",ret[ind].value  
END FOR

LET val = WSHelper.FindQueryStringValue("name1=val1&name2=val2","name1")

CALL WSHelper.SplitUrl("https://cube.strasbourg.4js.com:3128/GWS-492/TestSplitURL/test1")
  RETURNING scheme, host, port, path, query  
END MAIN

Output:

Query          1  
  name is name1  
  value is val1  
Query          2  
  name is name2  
  value is val2  
Query          3  
  name is name3  
  value is val3  

**WSHelper.SplitUrl**

Splits a complete URL string into pieces.

**Syntax**

```plaintext
SplitUrl(
  url STRING )
RETURNS ( scheme STRING,
  host STRING,
  port STRING,
  path STRING,
  query STRING )
```

1. `url` is a STRING with the URL to be split in pieces.
2. `scheme` is the URL scheme (http, https, file, ...). It includes all parts before //
3. `host` is the hostname.
4. `port`: the port number, or NULL if there is none.
   
   **Note**: If NULL is returned, there is always a default port depending on the scheme: 443 for HTTPS, and 80 for HTTP.
5. `path` is the path of the URL.
6. `query` is the query string of the URL, or NULL if there is none.

**Usage**

Splits a complete URL string into pieces.

In case of error, a NULL value will be returned.

**WSHelper functions example**

```plaintext
IMPORT FGL WSHelper  

MAIN  
  DEFINE val, scheme, host, port, path, query STRING  
  DEFINE ind INTEGER  
```
DEFINE ret WSHelper.WSQueryType

CALL WSHelper.SplitQueryString("name1=val1&name2=val2&name3=val3")
RETURNING ret

FOR ind = 1 TO ret.getLength()
    DISPLAY "Query", ind
    DISPLAY " name is ", ret[ind].name
    DISPLAY " value is ", ret[ind].value
END FOR

LET val = WSHelper.FindQueryStringValue("name1=val1&name2=val2", "name1")

CALL WSHelper.SplitUrl("https://cube.strasbourg.4js.com:3128/GWS-492/TestSplitURL/test1")
    RETURNING scheme, host, port, path, query

END MAIN

Output:

Query 1
    name is name1
    value is val1
Query 2
    name is name2
    value is val2
Query 3
    name is name3
    value is val3

OAuthAPI library
The OAuthAPI library.

The OAuthAPI file
The OAuthAPI library is delivered in the FGLGWS package. It is recommended that the OAuthAPI.42m library file is linked into every Genero Web Services client program that requires access to a Web service secured by OpenID Connect. It is located in $FGLDIR/lib.

OAuthAPI overview
The OAuthAPI library supports the OAuth protocol that authenticates user access and issues access tokens.

The OAuth protocol is widely-used as a means of securing access to Web services. It allows client access by verifying the identity of the end user. Third party Identity Providers (IdP) usually provide this service, or you can secure your RESTful services using the Genero Identity Provider service that is delivered in $FGLDIR/web_utilities/services/gip. For more information, see the Genero Identity Provider (GIP) section in the Genero Application Server User Guide.

The OAuthAPI library provides functions that enables you to get metadata provided by the IdP, such as endpoints, access tokens, scopes, user profile, etc. It has functions that can support Web services that are started:

• Behind a Genero Application Server (GAS).
• Or for Web services that are not behind GAS, for example, in the case of a mobile app.
• Or for a service (server side) to connect (as a service client) to another service.

An application started behind a GAS has a slightly different means of implementing authentication than a mobile app, or a service to service. You will notice, however, that OAuth implementation follows basically the same pattern.
This involves the client calling an initialization function to register the token to access the secure RESTful Web service. Once OAuth is initialized, functions that create HTTP requests with the access token, and manage metadata can be used. See Post-initialization functions on page 3511.

## Authenticate with GAS

To authenticate an application that is behind Genero Application Server, OAuth is implemented through delegation.

A Genero delegate service for OpenID Connect is delivered in $FGLDIR/web_utilities/services. This must be running to manage all delegated requests for applications or services run on the GAS. For more information, see the pages on How to implement delegation in the Genero Application Server User Guide.

You call the `init()` method in your client app. This initializes OAuth by:

- Reading the GAS environment variable OIDC_ACCESS_TOKEN that contains the initial access token the app got from the Genero Identity Provider (GIP), or the third party Identity Provider.
- Initiating a global variable with that access token.

Then any REST request used in your application will automatically set the access token in a HTTP request header and get access to the service. See the Main program code for access to secure service on page 3414 topic.

## Authenticate without GAS

To authenticate an application that is not behind GAS, for example a mobile app, you need to implement OAuth in a specific way.

In your application code, these functions need to be called in the following order:

1. **FetchOpenIDMetadata()** to retrieve the metadata from the Identity Provider from the required URL.
2. **RetrievePasswordToken()** to return the access token.
3. **initService()** to register the access token.

See the code sample in OAuth access without GAS on page 3521.

## Authenticate a service to a service

To successfully authenticate a service (server side) to connect (as a service client) to another service protected by an access token, you will need to implement OAuth this way.

The access token got from the IdP to check access to the service, is also required to access the client service.

1. Call the InitService() function to initiate OAuth and to register the access token.
2. Then you can call any of the OAuthAPI methods, such as CreateHTTPAuthorizationRequest, to perform requests to the other service.

See the code sample in InitService().

## OAuthAPI variables and records

The OAuthAPI library provides public variables and records.

### RegisterRequestType

The OAuthAPI library defines the OAuthAPI.RegisterRequestType as a record:

```plaintext
PUBLIC
TYPE RegisterRequestType RECORD
   redirect_uris DYNAMIC ARRAY OF STRING, # REQUIRED
   response_types DYNAMIC ARRAY OF STRING, # OPTIONAL
   grant_types DYNAMIC ARRAY OF STRING, # OPTIONAL
   client_name STRING,
   client_description STRING,
   scope STRING # OPTIONAL
END RECORD
```

For more information on these parameters, see the Authorization Request section of the OAuth specification.
RegisterResponseType

The OAuthAPI library defines the OAuthAPI.RegisterResponseType as a record:

```plaintext
PUBLIC TYPE RegisterResponseType RECORD
    client_id STRING, # REQUIRED
    client_secret STRING, # OPTIONAL
    grant_types DYNAMIC ARRAY OF STRING, # OPTIONAL
    redirect_uris DYNAMIC ARRAY OF STRING, # OPTIONAL
    client_name STRING,
    client_description STRING,
    scope STRING # OPTIONAL
END RECORD
```

For more information on these parameters, see the Authorization Response section of the OAuth specification.

OpenIDMetadataType

The OAuthApp library defines the OAuthApp.OpenIDMetadataType as a public record:

```plaintext
PUBLIC TYPE OpenIDMetadataType RECORD
    issuer STRING, # REQUIRED
    authorization_endpoint STRING, # REQUIRED
    token_endpoint STRING, # REQUIRED
    userinfo_endpoint STRING, # RECOMMENDED
    jwks_uri STRING, # REQUIRED
    registration_endpoint STRING, # RECOMMENDED
    scopes_supported DYNAMIC ARRAY OF STRING, # REQUIRED
    response_types_supported DYNAMIC ARRAY OF STRING, # REQUIRED
    response_modes_supported DYNAMIC ARRAY OF STRING, # OPTIONAL
    grant_types_supported DYNAMIC ARRAY OF STRING, # OPTIONAL
    acr_values_supported DYNAMIC ARRAY OF STRING, # OPTIONAL
    subject_types_supported DYNAMIC ARRAY OF STRING, # REQUIRED
    id_token_signing_alg_values_supported DYNAMIC ARRAY OF STRING, # REQUIRED
    id_token_encryption_alg_values_supported DYNAMIC ARRAY OF STRING, # OPTIONAL
    id_token_encryption_enc_values_supported DYNAMIC ARRAY OF STRING, # OPTIONAL
    userinfo_signing_alg_values_supported DYNAMIC ARRAY OF STRING, # OPTIONAL
    userinfo_encryption_alg_values_supported DYNAMIC ARRAY OF STRING, # OPTIONAL
    userinfo_encryption_enc_values_supported DYNAMIC ARRAY OF STRING, # OPTIONAL
    request_object_signing_alg_values_supported DYNAMIC ARRAY OF STRING, # OPTIONAL
    request_object_encryption_alg_values_supported DYNAMIC ARRAY OF STRING, # OPTIONAL
    request_object_encryption_enc_values_supported DYNAMIC ARRAY OF STRING, # OPTIONAL
    token_endpoint_auth_methods_supported DYNAMIC ARRAY OF STRING, # OPTIONAL
    token_endpoint_auth_signing_alg_values_supported DYNAMIC ARRAY OF STRING, # OPTIONAL
    display_values_supported DYNAMIC ARRAY OF STRING, # OPTIONAL
    claim_types_supported DYNAMIC ARRAY OF STRING, # OPTIONAL
    service_documentation STRING, # OPTIONAL
    claims_supported DYNAMIC ARRAY OF STRING, # RECOMMENDED
    claims_locales_supported STRING, # OPTIONAL
    ui_locales_supported STRING, # OPTIONAL
    claims_parameter_supported STRING, # OPTIONAL
```
request_parameter_supported STRING, # OPTIONAL
request_uri_parameter_supported STRING, # OPTIONAL
require_request_uri_registration STRING, # OPTIONAL
op_policy_uri STRING, # OPTIONAL
op_tos_uri STRING, # OPTIONAL
end_session_endpoint STRING # OPTIONAL
END RECORD

For further details, refer to the OpenID Connect Core specification.
**OAuthAPI APIs**

The OAuthAPI library provides functions for working with OAuth.
<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Init</strong></td>
<td>To be called in a Genero app accessing a secure RESTful Web service started behind a Genero Application Server.</td>
</tr>
<tr>
<td>(</td>
<td></td>
</tr>
<tr>
<td>cnx_timeout INTEGER,</td>
<td></td>
</tr>
<tr>
<td>client_id STRING,</td>
<td></td>
</tr>
<tr>
<td>client_secret STRING</td>
<td></td>
</tr>
<tr>
<td>RETURNS BOOLEAN</td>
<td></td>
</tr>
<tr>
<td><strong>InitService</strong></td>
<td>To be called in a Genero Web service started via OpenID Connect accessing another secure RESTful Web service as a client.</td>
</tr>
<tr>
<td>(</td>
<td></td>
</tr>
<tr>
<td>cnx_timeout INTEGER,</td>
<td></td>
</tr>
<tr>
<td>access_token STRING</td>
<td></td>
</tr>
<tr>
<td>RETURNS BOOLEAN</td>
<td></td>
</tr>
<tr>
<td><strong>GetOpenIDMetadata</strong></td>
<td>Get metadata from the Identity Provider for a service running on a Genero Application Server (GAS).</td>
</tr>
<tr>
<td>()</td>
<td></td>
</tr>
<tr>
<td>RETURNS OAuthAPI.OpenIDMetadataType</td>
<td></td>
</tr>
<tr>
<td><strong>FetchOpenIDMetadata</strong></td>
<td>Fetch metadata from the Identity Provider at the URL provided.</td>
</tr>
<tr>
<td>(</td>
<td></td>
</tr>
<tr>
<td>timeout INTEGER,</td>
<td></td>
</tr>
<tr>
<td>idp STRING</td>
<td></td>
</tr>
<tr>
<td>RETURNS OAuthAPI.OpenIDMetadataType</td>
<td></td>
</tr>
<tr>
<td><strong>GetIDPIssuer</strong></td>
<td>Get endpoint of the Identity Provider.</td>
</tr>
<tr>
<td>()</td>
<td></td>
</tr>
<tr>
<td>RETURNS STRING</td>
<td></td>
</tr>
<tr>
<td><strong>GetIDSubject</strong></td>
<td>Get OAuth subject identifier of ID Token.</td>
</tr>
<tr>
<td>()</td>
<td></td>
</tr>
<tr>
<td>RETURNS INTEGER</td>
<td></td>
</tr>
<tr>
<td><strong>GetIDScopes</strong></td>
<td>Get OAuth ID Token authorization scopes.</td>
</tr>
<tr>
<td>()</td>
<td></td>
</tr>
<tr>
<td>RETURNS DYNAMIC ARRAY OF STRING</td>
<td></td>
</tr>
<tr>
<td><strong>ExtractTokenFromHTTPRequest</strong></td>
<td>Return the OAuth access token.</td>
</tr>
<tr>
<td>(</td>
<td></td>
</tr>
<tr>
<td>req comm.HTTPServiceRequest)</td>
<td></td>
</tr>
<tr>
<td>RETURNS STRING</td>
<td></td>
</tr>
<tr>
<td><strong>RetrieveServiceToken</strong></td>
<td>Return the OAuth service access token via client app credentials.</td>
</tr>
<tr>
<td>(</td>
<td></td>
</tr>
<tr>
<td>timeout INTEGER,</td>
<td></td>
</tr>
<tr>
<td>TokenServiceURL STRING,</td>
<td></td>
</tr>
<tr>
<td>client_id STRING,</td>
<td></td>
</tr>
<tr>
<td>secret_id STRING,</td>
<td></td>
</tr>
<tr>
<td>scope STRING</td>
<td></td>
</tr>
<tr>
<td>RETURNS STRING, INTEGER</td>
<td></td>
</tr>
<tr>
<td><strong>RetrievePasswordToken</strong></td>
<td>Return the OAuth service access token via user name and password.</td>
</tr>
<tr>
<td>(</td>
<td></td>
</tr>
<tr>
<td>timeout INTEGER,</td>
<td></td>
</tr>
<tr>
<td>TokenServiceURL STRING,</td>
<td></td>
</tr>
<tr>
<td>usr STRING</td>
<td></td>
</tr>
<tr>
<td>pass STRING</td>
<td></td>
</tr>
<tr>
<td>scope STRING</td>
<td></td>
</tr>
<tr>
<td>RETURNS ( STRING, INTEGER )</td>
<td></td>
</tr>
<tr>
<td><strong>GetMyAccessToken</strong></td>
<td>Get a valid access token.</td>
</tr>
<tr>
<td>()</td>
<td></td>
</tr>
<tr>
<td>RETURNS STRING</td>
<td></td>
</tr>
</tbody>
</table>
Initialization functions
Functions that initialize OAuth by getting the access token.

OAuthAPI.init
To be called in a Genero app accessing a secure RESTful Web service started behind a Genero Application Server.

Syntax

```
Init(
    cnx_timeout INTEGER,
    client_id STRING,
    client_secret STRING )
RETURNS BOOLEAN
```

1. `cnx_timeout` is a connection timeout to the REST service with value in seconds.
2. `client_id` is the application ID assigned to the app when registered.
3. `client_secret` is the application secret created for the app.

Returns FALSE if the mandatory access token is null.

Usage
Use this function to initiate information provided by the Identity Provider to access the secure RESTful Web service. It calls environment variables containing the OAuth information got from the Genero Identity Provider (GIP) or a third party Identity Provider (IdP). It registers the access token and access information (such as subject, scopes, endpoints, etc.) with the GWS engine.

In case of error, a NULL value will be returned.

```
IMPORT FGL OAuthAPI
DEFINE my_user_id INTEGER
MAIN
    # ...
    # Init OAuthAPI
    IF NOT OAuthAPI.init(5, "AF350CBC-8801-4DFB-9A78-A95B25BB32AF", "8JEq3HBfXrmj/8vMP66iaRQnGrWVyjqr") THEN
        DISPLAY "Error: unable to initialize OAuth"
        EXIT PROGRAM 1
    ELSE
        LET my_user_id = OAuthAPI.getIDSubject
    END IF
    # ...
END MAIN
```

OAuthAPI.InitService
To be called in a Genero Web service started via OpenID Connect accessing another secure RESTful Web service as a client.

Syntax

```
InitService(
    cnx_timeout INTEGER,
    access_token STRING )
RETURNS BOOLEAN
```
1. `cnx_timeout` is a connection timeout from the REST service with value in seconds. The connection timeout is needed to avoid apps getting blocked by requests.

2. `access_token` is the valid access token for accessing the RESTful Web service.

   Returns FALSE if the mandatory access token is null.

**Usage**

Use this function to register the access token to be used when a service (server side) has to connect (as a service client) to another service protected by an access token. It checks if the OAuth service is initiated.

The role of the function is to register the access token, so that you can then call any of the OAuthAPI methods, such as `CreateHTTPAuthorizationRequest`, to perform requests to the other service.

In case of error, a NULL value will be returned.

In the code sample, the `WSContext` on page 3481 dictionary is defined to get some information coming from the service configuration file (.xcf).

The access token set during token validation to check access to the service is required to access the client service. Therefore, you need to call `InitService()` with it.

Typically, only the access token is required. If you need to get metadata, a call to `FetchOpenIDMetadata()` saves the metadata in an `OpenIDMetadataType` on page 3506 record.

**OAuthAPI.InitService function**

```falg
IMPORT FGL OAuthAPI
PRIVATE DEFINE ctx DICTIONARY ATTRIBUTE(WSContext) OF STRING
MAIN
  DEFINE metadata OAuthAPI.OpenIDMetadataType
  DEFINE access_token STRING
  DEFINE idp STRING

  # retrieve access_token
  LET access_token = ctx["OIDC_ACCESS_TOKEN"]

  # retrieve IDP URL
  LET idp = ctx["Parameter-IDP"]

  # retrieve metadata
  CALL OAuthAPI.FetchOpenIDMetadata(5, idp) RETURNING metadata.*

  # Init OAuth service
  IF NOT OAuthAPI.InitService(5, access_token) THEN
    DISPLAY "Cannot initiate OAuth service"
  END IF

  CALL MyOtherService(access_token)
  # ...
END MAIN
```

**Post-initialization functions**

Functions that manage metadata, tokens, and scopes.
**OAuthAPI.GetOpenIDMetadata**
Get metadata from the Identity Provider for a service running on a Genero Application Server (GAS).

**Syntax**

```
GetOpenIDMetadata ()
RETURNS OAuthAPI.OpenIDMetadataType
```

NULL may be returned if metadata is not found.

**Usage**

Use this function to retrieve metadata (such as registration endpoint, supported scopes, user information endpoints, etc.) provided by the Genero Identity Provider for access to a secure RESTful Web service. A request is made to return the metadata and the information is stored in the `OpenIDMetadataType` record.

**Note:**

If the service is not started behind the GAS, call the `FetchOpenIDMetadata()` function instead.

In case of error, a NULL value will be returned.

**OAuthAPI.GetOpenIDMetadata function**

```
IMPORT FGL OAuthAPI

DEFINE metadata OAuthAPI.OpenIDMetadataType
DEFINE ind INTEGER

MAIN
  # ...
  CALL OAuthAPI.GetOpenIDMetadata() RETURNING metadata.*
  IF metadata.issuer IS NULL THEN
    ERROR "IdP not available"
  ELSE
    DISPLAY "Registration endpoint is: ", metadata.registration_endpoint
    FOR ind = 1 TO metadata.scopes_supported.getLength()
      DISPLAY "Scope is: ", metadata.scopes_supported[ind]
  END FOR
  END IF
  # ...
END MAIN
```

**Related concepts**

**OAuthAPI.FetchOpenIDMetadata** on page 3518
Fetch metadata from the Identity Provider at the URL provided.

**ID token functions**

Functions that get access token metadata.

**OAuthAPI.GetIDPIssuer**
Get endpoint of the Identity Provider.

**Syntax**

```
GetIDPIssuer ()
RETURNS STRING
```

NULL may be returned if the metadata is not found.
Usage

Use this function to get the endpoint of the Identity Provider securing the RESTful Web service. IdP issuer information can be retrieved from the GWS engine once it has been initialized by `OAuthAPI.init` on page 3510.

In case of error, a NULL value will be returned.

**OAuthAPI.GetIDPIssuer function**

```fgo
IMPORT FGL OAuthAPI
DEFINE issuer STRING

MAIN
  IF NOT OAuthAPI.init(5, "AF350C80C-80A9-4DFA-9A78-A95B25BB32AF", "8JEq3HBfxrmj/8vMP66iaRQnGrWVyjqr" ) THEN
    MESSAGE "Error: unable to initialize OAuth"
    EXIT PROGRAM 1
  ELSE
    LET issuer = OAuthAPI.GetIDPIssuer()
  END IF

END MAIN
```

**OAuthAPI.GetIDSubject function**

Get OAuth subject identifier of ID Token.

**Syntax**

```fgo
GetIDSubject()
RETURNS INTEGER
```

NULL may be returned if the metadata is not found.

Usage

Use this function to get the subject identifier of the Identity Provider securing the RESTful Web service. IdP subject information can be retrieved from the GWS engine once it has been initialized by `OAuthAPI.init` on page 3510.

In case of error, a NULL value will be returned.

**OAuthAPI.GetIDSubject function**

```fgo
IMPORT FGL OAuthAPI
DEFINE sub INTEGER

MAIN
  IF NOT OAuthAPI.init(5, "AF350C80C-80A9-4DFA-9A78-A95B25BB32AF", "8JEq3HBfxrmj/8vMP66iaRQnGrWVyjqr" ) THEN
    MESSAGE "Error: unable to initialize OAuth"
    EXIT PROGRAM 1
  ELSE
    LET sub = OAuthAPI.GetIDSubject()
  END IF

END MAIN
```

**OAuthAPI.GetIDScopes**
Get OAuth ID Token authorization scopes.

**Syntax**

```
GetIDScopes()
RETURNS DYNAMIC ARRAY OF STRING
```

**Usage**

Use this function to get the list of authorization OAuth 2.0 scopes that the Identity Provider securing the RESTful Web service supports. IdP scope information can be retrieved from the GWS engine once it has been initialized by OAuthAPI.init on page 3510.

In case of error, a NULL value will be returned.

**OAuthAPI.GetIDScopes function**

```fgl
IMPORT FGL OAuthAPI
MAIN
DEFINE ind INTEGER
DEFINE scopes DYNAMIC ARRAY OF STRING

# ...
IF NOT OAuthAPI.init(5, "AF350CBC-8801-4DFB-9A78-A95B25BB32AF", "8Jeq3HBfxrmj/8vMP66iaRQnGrWVyjqr" ) THEN
  MESSAGE "Error: unable to initialize OAuth"
  EXIT PROGRAM 1
ELSE
  LET scopes = OAuthAPI.GetIDScopes()
  FOR ind = 1 TO scopes.getLength()
    DISPLAY "Scope supported is: ", scopes[ind]
  END FOR
END IF
# ...
END MAIN
```

**Access token functions**

Functions that manage OAuth access tokens.

**OAuthAPI.GetMyAccessToken**

Get a valid access token.

**Syntax**

```
GetMyAccessToken()
RETURNS STRING
```

NULL may be returned if the access token is not found or can not be renewed.

**Usage**

Use this function to retrieve the access token provided by the Identity Provider for access to a secure RESTful Web service. IdP token information is retrieved from the GWS engine once it has been initialized by OAuthAPI.init on page 3510. If the access token has expired, a new token is requested (if a refresh token is available) and returned.

In case of error, a NULL value will be returned.
OAuthAPI.GetMyAccessToken function

```plaintext
IMPORT FGL OAuthAPI
DEFINE access_token STRING
MAIN
IF NOT OAuthAPI.init(5, "AF350CBC-8801-4DFB-9A78-A95B25BB32AF", "8JEq3HBfxrmj/8vMP66iaRQnGrWVyjqr") THEN
    MESSAGE "Error: unable to initialize OAuth"
    EXIT PROGRAM 1
ELSE
    LET access_token = OAuthAPI.GetMyAccessToken()
END IF
# ...
END MAIN
```

Authorization request functions
Functions that manage HTTPRequest with OAuth access token.

OAuthAPI.CreateHTTPAuthorizationRequest
Create an HTTPRequest with OAuth access token.

Syntax

```plaintext
CreateHTTPAuthorizationRequest (url STRING )
RETURNS com.HTTPRequest
```

1. `url` is a STRING with the URL to connect to.

Returns a com.HTTPRequest object. NULL may be returned if the access token can not be renewed.

Usage
Use this function to create an HTTPRequest object with OAuth access token for access to a secure RESTful Web service.

In case of error, a NULL value will be returned.

OAuthAPI.CreateHTTPAuthorizationRequest function

```plaintext
IMPORT com
IMPORT FGL OAuthAPI
MAIN
DEFINE req com.HTTPRequest
DEFINE resp com.HTTPResponse
DEFINE url STRING
TRY
    # call the init function first
    # get request path
    LET url = fgl_getenv("OIDC_USERINFO_ENDPOINT")
    # Create oauth request
    LET req = OAuthAPI.CreateHTTPAuthorizationRequest(url)
    # Perform request
END TRY
```
CALL req.setMethod("GET")
CALL req.setHeader("Accept", "application/json")
CALL req.DoRequest()

# Retrieve response
LET resp = req.getResponse()

# Process response
CASE resp.getStatusCode()
  WHEN 200 # Success
    # ...
  OTHERWISE
    DISPLAY " Error code is: ", resp.getStatusCode()
    EXIT PROGRAM 1
END CASE

CATCH
  DISPLAY "ERROR : ", status, SQLCA.SQLERRM
  EXIT PROGRAM 1
END TRY
END MAIN

OAuthAPI.RetryHTTPRequest
Retry an HTTPRequest with OAuth access token to check if the access token has expired.

Syntax

RetryHTTPRequest ( 
  resp com.HTTPResponse  )
RETURNS BOOLEAN

1. resp is a com.HTTPResponse object.
   Returns TRUE if the access token has expired and the request must be sent again, FALSE otherwise.

Usage
Use this function to check whether the access token to a secure RESTful Web service has expired. If expired, it will perform the refresh and then do the request.
In case of error, a NULL value will be returned.

OAuthAPI.RetryHTTPRequest function

IMPORT com
IMPORT FGL OAuthAPI

MAIN

DEFINE req com.HTTPRequest
DEFINE resp com.HTTPResponse
DEFINE url STRING

TRY
  # Get path
  LET url = fgl_getenv("OIDC_USERINFO_ENDPOINT")

  WHILE TRUE
    # Create oauth request
    LET req = OAuthAPI.CreateHTTPAuthorizationRequest(url)

    # Perform request
CALL req.setMethod("GET")
CALL req.setHeader("Accept", "application/json")
CALL req.DoRequest()

# Retrieve response
LET resp = req.getResponse()
# Retry if access token has expired
IF NOT OAuthAPI.RetryHTTPRequest(resp) THEN
   EXIT WHILE
END IF
END WHILE

# Process response
CASE resp.getStatusCode()
   WHEN 200 #Success
   # ...
   OTHERWISE
      DISPLAY " Error code is: ", resp.getStatusCode()
   EXIT PROGRAM 1
END CASE
CATCH
   DISPLAY "ERROR : ", status, SQLCA.SQLERRM
   EXIT PROGRAM 1
END TRY
END MAIN

Related concepts
OAuth access without GAS on page 3521
Example getting OAuth access when the application is not behind GAS, for example a mobile app.

Helper functions
Functions that manage OpenID Connect metadata for Web services not behind a Genero Application Server.
OAuthAPI.RetrievePasswordToken
Return the OAuth service access token via username and password.

Syntax

RetrievePasswordToken(
   timeout INTEGER,
   TokenServiceURL STRING,
   usr STRING,
   pass STRING,
   scope STRING )
RETURNS ( STRING, INTEGER )

1. timeout. Defines the number of seconds.
2. TokenServiceURL. This is the token endpoint of the Identity Provider (IdP) securing the service.
3. usr. This is the user's login details.
4. pass. This is the user password.
5. scope. This is a space-separated list of scopes defining user access.

Returns a valid access token and when it expires in seconds. NULL may be returned if the access token is not available.

Usage
Use this function to retrieve a valid access token for a client app accessing a RESTful Web service using the username and password.
In case of error, a NULL value will be returned.

**OAuthAPI.RetrievePasswordToken function**

```plaintext
IMPORT FGL OAuthAPI

MAIN

DEFINE metadata OAuthAPI.OpenIDMetadataType
DEFINE token STRING
DEFINE expire INTEGER
DEFINE usr, pass, STRING
DEFINE scope STRING
DEFINE idp_url STRING

TRY
    # ...
    CALL OAuthAPI.FetchOpenIDMetadata(20, idp_url)
        RETURNING metadata.*
    IF metadata.issuer IS NULL THEN
        ERROR "IdP not available"
        EXIT PROGRAM 1
    ELSE
        CALL OAuthAPI.RetrievePasswordToken(5, metadata.token_endpoint, usr, pass, scope )
            RETURNING token, expire
        IF token IS NULL THEN
            DISPLAY "Unable to retrieve token"
            EXIT PROGRAM 1
        ELSE
            DISPLAY "Access token value :",token
            DISPLAY SFMT("Token expires in %1 seconds",expire)
        END IF
    END IF
END IF
CATCH
    DISPLAY "ERROR : ",status,SQLCA.SQLERRM
    EXIT PROGRAM 1
END TRY
END MAIN
```

**Related concepts**

**OAuthAPI.RetrieveServiceToken** on page 3520
Return the OAuth service access token via client app credentials.

**OAuth access without GAS** on page 3521
Example getting OAuth access when the application is not behind GAS, for example a mobile app.

**OAuthAPI.FetchOpenIDMetadata**
Fetch metadata from the Identity Provider at the URL provided.

**Syntax**

```plaintext
FetchOpenIDMetadata(
    timeout INTEGER,
    idp STRING )
RETURNS OAuthAPI.OpenIDMetadataType
```

1. `timeout` defines the number of seconds.
2. `idp` is the URL of the Identity Provider (IdP).

NULL may be returned if metadata is not found.
Usage
Use this function to retrieve the metadata (such as registration endpoint, supported scopes, user information endpoints, etc.) provided by the Identity Provider for access to a secure RESTful Web service. The information is retrieved in a requested to the IdP via the URL provided and stored in the OpenIDMetadataType record.

Note:
If the service is started behind the GAS, call the GetOpenIDMetadata() function instead.
In case of error, a NULL value will be returned.

OAuthAPI.FetchOpenIDMetadata function

```
IMPORT FGL OAuthAPI
PRIVATE DEFINE metadata OAuthAPI.OpenIDMetadataType
DEFINE idp_url STRING
MAIN
  # ...
  CALL OAuthAPI.FetchOpenIDMetadata(20, idp_url)
  RETURNING metadata.*
  IF metadata.issuer IS NULL THEN
    ERROR "IdP not available"
  ELSE
    DISPLAY "Registration endpoint is: ", metadata.registration_endpoint
  END IF
  # ...
END MAIN
```

Related concepts
OAuth access without GAS on page 3521
Example getting OAuth access when the application is not behind GAS, for example a mobile app.

OAuthAPI.ExtractTokenFromHTTPRequest
Return the OAuth access token.

Syntax

```
ExtractTokenFromHTTPRequest (req com.HTTPServiceRequest) RETURNS STRING
```

1. req is a com.HTTPServiceRequest object.

Returns the access token. NULL may be returned if the access token is not found.

Usage
Use this function to extract the access token from an com.HTTPServiceRequest object accessing a RESTful Web service.
In case of error, a NULL value will be returned.

OAuthAPI.ExtractTokenFromHTTPRequest function

```
DEFINE req com.HttpServiceRequest
DEFINE access_token STRING

TRY
  # Start server for all registered Web Services
  CALL com.WebServiceEngine.Start()

  WHILE TRUE
    LET req = com.WebServiceEngine.GetHttpServiceRequest(-1)
    IF req IS NULL THEN
      DISPLAY "HTTP request timeout...: ", CURRENT YEAR TO FRACTION
      EXIT PROGRAM 1
    ELSE
      # Retrieve access token
      LET access_token = OAuthAPI.ExtractTokenFromHTTPRequest(req)
      IF access_token IS NOT NULL THEN
        DISPLAY "token is: ", access_token
      END IF
    END IF
  END WHILE
CATCH
  DISPLAY "ERROR : ",status,SQLCA.SQLERRM
  EXIT PROGRAM 1
END TRY
END MAIN

Related concepts
OAuth access without GAS on page 3521
Example getting OAuth access when the application is not behind GAS, for example a mobile app.

OAuthAPI.RetrieveServiceToken
Return the OAuth service access token via client app credentials.

Syntax

RetrieveServiceToken(
  timeout INTEGER,
  TokenServiceURL STRING,
  client_id STRING,
  secret_id STRING,
  scope STRING )
RETURNS STRING, INTEGER

1. timeout. Defines the number of seconds.
2. TokenServiceURL. This is the token endpoint of the Identity Provider (IdP) securing the service.
3. client_id. This is the application ID assigned to the app when registered.
4. client_secret. This is the application secret created for the app.
5. scope. This is a space-separated list of scopes defining user access.

Returns a valid access token and when it expires in seconds. NULL may be returned if the access token is not available.

Usage

Use this function to retrieve a valid access token for a client app accessing a RESTful Web service using the client app's own client id and client secret credentials.

In case of error, a NULL value will be returned.
OAuthAPI.RetrieveServiceToken function

```plaintext
IMPORT FGL OAuthAPI

MAIN

DEFINE metadata OAuthAPI.OpenIDMetadataType
DEFINE token STRING
DEFINE expire INTEGER
DEFINE client_id STRING
DEFINE secret_id STRING
DEFINE scope STRING
DEFINE idp_url STRING

TRY
    # ...
    CALL OAuthAPI.FetchOpenIDMetadata(20, idp_url)
    RETURNING metadata.*
    IF metadata.issuer IS NULL THEN
        ERROR "IdP not available"
        EXIT PROGRAM 1
    ELSE
        CALL OAuthAPI.RetrieveServiceToken(5, metadata.token_endpoint,
            client_id, secret_id, scope )
        RETURNING token, expire
        IF token IS NULL THEN
            DISPLAY "Unable to retrieve token"
            EXIT PROGRAM 1
        ELSE
            DISPLAY "Access token value :",token
            DISPLAY SFMT("Token expires in %1 seconds",expire)
        END IF
    END IF
CATCH
    DISPLAY "ERROR : ",status,SQLCA.SQLERRM
    EXIT PROGRAM 1
END TRY

END MAIN
```

**Examples**

OAuthAPI Helper function examples.

OAuth access without GAS

Example getting OAuth access when the application is not behind GAS, for example a mobile app.

**OAuthAPI helper functions**

```plaintext
IMPORT com
IMPORT FGL OAuthAPI
IMPORT util

PUBLIC TYPE QueryUserInfoResponseBodyType DYNAMIC ARRAY OF RECORD
    user_id INTEGER,
    given_name STRING,
    family_name STRING,
    email STRING
END RECORD

# Error codes
PUBLIC CONSTANT C_SUCCESS = 0

PUBLIC FUNCTION QueryUserInfo(idp_url STRING, user_login STRING,
    user_password STRING) RETURNS(INTEGER, QueryUserInfoResponseBodyType)
```
DEFINE fullpath base.StringBuffer
DEFINE contentType STRING
DEFINE req com.HTTPRequest
DEFINE resp com.HTTPResponse
DEFINE resp_body QueryUserInfoResponseBodyType
DEFINE json_body STRING
DEFINE expire INTEGER
DEFINE meta OAuthAPI.OpenIDMetadataType

CALL OAuthAPI.FetchOpenIDMetadata(5, ipd_url) RETURNING meta.*
CALL OAuthAPI.RetrievePasswordToken(5, meta.token_endpoint, user_login, user_password, "profile") RETURNING access_token, expire
CALL OAuthAPI.initService(5, access_token)

TRY

# Prepare request path
LET fullpath = base.StringBuffer.Create()
CALL fullpath.append("/userinfo/v1/query")

WHILE TRUE

# Create oauth request and configure it
LET req =
OAuthAPI.CreateHTTPAuthorizationRequest(meta.userinfo_endpoint
SFMT("%1%2", Endpoint.Address.Uri, fullpath.toString()))

# Perform request
CALL req.setMethod("GET")
CALL req.setHeader("Accept", "application/json")
CALL req.DoRequest()

# Retrieve response
LET resp = req.getResponse()

# Retry if access token has expired
IF NOT OAuthAPI.RetryHTTPRequest(resp) THEN
END IF
EXIT WHILE
END IF

# Process response
INITIALIZE resp_body TO NULL
LET contentType = resp.getHeader("Content-Type")
CASE resp.getStatusCode()
WHEN 200 #Success
IF contentType MATCHES "*application/json*" THEN
# Parse JSON response
LET json_body = resp.getTextResponse()
CALL util.JSON.parse(json_body, resp_body)
RETURN C_SUCCESS, resp_body
END IF
RETURN -1, resp_body
OTHERWISE
RETURN resp.getStatusCode(), resp_body
END CASE
CATCH
RETURN -1, resp_body
END TRY
END FUNCTION
Attributes to customize XML serialization

See The Serializer class on page 3033 for information on setting serialization options when mapping BDL and XML data.

BDL to/from XML type mappings

Starting with Genero 2.0, you can add optional attributes to the definition of program variables to be used for XML serialization. These attributes can be used to map a BDL data type used in the input or output message of a Genero Web Service application to a specific XML data type, rather than using the default.

For example, if an XML Schema boolean data type is required for an application, and the corresponding BDL type is a SMALLINT, you can use an attribute to map the BDL SMALLINT variable to the XML boolean.

The following example uses the XSDBoolean attribute to map a BDL SMALLINT variable to an XML Schema boolean type, and assigns an uppercase name as the XMLName attribute:

```plaintext
GLOBALS
DEFINE invoice_out RECORD
  ok SMALLINT ATTRIBUTES(XSDBoolean,XMLName="OK")
END RECORD
END GLOBALS
```

If you assign your own XMLName attributes, be sure to respect the conventions when using the RPC Service Style.

See the Tutorial: Writing a GWS Server application for additional information about input and output messages.

Default BDL/XML mapping

By default, Genero Web Services maps BDL variables in the input or output messages of a WS application to their corresponding XML data types, enabling values to be passed between applications and Web Services. The XML data types conform to the standard XML Schema Definition (XSD):

Table 742: Default XML Mapping

<table>
<thead>
<tr>
<th>Data type of BDL variable</th>
<th>Default XML data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE</td>
<td>xsd:base64binary</td>
</tr>
<tr>
<td>CHAR</td>
<td>xsd:string</td>
</tr>
<tr>
<td>DATE</td>
<td>xds:date</td>
</tr>
<tr>
<td>DATETIME YEAR TO FRACTION(1-5)</td>
<td>xsd:dateTime</td>
</tr>
<tr>
<td>DATETIME YEAR TO SECOND</td>
<td>xsd:dateTime</td>
</tr>
<tr>
<td>DATETIME YEAR TO HOUR</td>
<td>xsd:dateTime</td>
</tr>
<tr>
<td>DATETIME YEAR TO MINUTE</td>
<td>xsd:dateTime</td>
</tr>
<tr>
<td>DATETIME YEAR TO YEAR</td>
<td>xsd:gYear</td>
</tr>
<tr>
<td>DATETIME YEAR TO MONTH</td>
<td>xsd:gYearMonth</td>
</tr>
<tr>
<td>DATETIME YEAR TO DAY</td>
<td>xsd:date</td>
</tr>
<tr>
<td>DATETIME MONTH TO MONTH</td>
<td>xsd:gMonth</td>
</tr>
<tr>
<td>DATETIME MONTH TO DAY</td>
<td>xsd:gMonthDay</td>
</tr>
<tr>
<td>DATETIME DAY TO DAY</td>
<td>xsd:gDay</td>
</tr>
<tr>
<td>DATETIME HOUR TO HOUR</td>
<td>xsd:time</td>
</tr>
<tr>
<td>Data type of BDL variable</td>
<td>Default XML data type</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>DATETIME HOUR TO MINUTE</td>
<td>xsd:time</td>
</tr>
<tr>
<td>DATETIME HOUR TO SECOND</td>
<td>xsd:time</td>
</tr>
<tr>
<td>DATETIME HOUR TO FRACTION(1-5)</td>
<td>xsd:time</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>xsd:decimal</td>
</tr>
<tr>
<td>FLOAT</td>
<td>xsd:double</td>
</tr>
<tr>
<td>INTEGER</td>
<td>xsd:int</td>
</tr>
<tr>
<td>INTERVAL</td>
<td>xsd:duration</td>
</tr>
<tr>
<td>SMALLFLOAT</td>
<td>xsd:float</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>xsd:short</td>
</tr>
<tr>
<td>STRING</td>
<td>xsd:string</td>
</tr>
<tr>
<td>TEXT</td>
<td>xsd:string</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>xsd:string</td>
</tr>
<tr>
<td>TINYINT</td>
<td>xsd:byte</td>
</tr>
<tr>
<td>BIGINT</td>
<td>xsd:long</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>xsd:boolean</td>
</tr>
</tbody>
</table>

In addition, the Web Service Style that you use determines what default XMLName on page 3548 attributes are assigned to variables.

**Type mapping attributes**

The attributes listed in this table cannot have values.

**Table 743: Mapping between simple BDL and XML data types**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSDAnySimpleType</td>
<td>Map BDL STRING or VARCHAR to XML Schema simpleType.</td>
</tr>
<tr>
<td>XSDAnyType</td>
<td>Map BDL STRING or VARCHAR to XML Schema anyType.</td>
</tr>
<tr>
<td>XSDAnyURI</td>
<td>Map BDL STRING or VARCHAR to XML Schema anyURI.</td>
</tr>
<tr>
<td>XSDBase64binary</td>
<td>Map BDL BYTE to the XML Schema base64binary.</td>
</tr>
<tr>
<td>XSDBoolean</td>
<td>Map BDL BOOLEAN, SMALLINT or INTEGER to XML Schema boolean.</td>
</tr>
<tr>
<td>XSDByte</td>
<td>Map BDL TINYINT, SMALLINT or BIGINT to XML Schema byte.</td>
</tr>
<tr>
<td>XSDDate</td>
<td>Map BDL DATE or DATETIME to XML Schema date.</td>
</tr>
<tr>
<td>XSDDateTime</td>
<td>Map BDL DATETIME to XML Schema dateTime.</td>
</tr>
<tr>
<td>XSDDecimal</td>
<td>Map BDL DECIMAL to XML Schema decimal.</td>
</tr>
<tr>
<td>XSDDouble</td>
<td>Map BDL FLOAT to XML Schema double.</td>
</tr>
<tr>
<td>XSDDuration</td>
<td>Map BDL INTERVAL to XML Schema duration.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Definition</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>XSDEntities</td>
<td>Map BDL STRING or VARCHAR to XML Schema entities.</td>
</tr>
<tr>
<td>XSDEntity</td>
<td>Map BDL STRING or VARCHAR to XML Schema entity.</td>
</tr>
<tr>
<td>XSDFloat</td>
<td>Map BDL SMALLFLOAT to XML Schema float.</td>
</tr>
<tr>
<td>XSDGday</td>
<td>Map BDL DATETIME to XML Schema gDay.</td>
</tr>
<tr>
<td>XSDGMonth</td>
<td>Map BDL DATETIME to XML Schema gMonth.</td>
</tr>
<tr>
<td>XSDGMonthDay</td>
<td>Map BDL DATETIME to XML Schema gMonthDay.</td>
</tr>
<tr>
<td>XSDGYear</td>
<td>Map BDL DATETIME to XML Schema gYear.</td>
</tr>
<tr>
<td>XSDGYearMonth</td>
<td>Map BDL DATETIME to XML Schema gYearMonth.</td>
</tr>
<tr>
<td>XSDHexBinary</td>
<td>Map BDL BYTE to XML Schema hexBinary.</td>
</tr>
<tr>
<td>XSDID</td>
<td>Map BDL STRING or VARCHAR to XML Schema id.</td>
</tr>
<tr>
<td>XSDIDREF</td>
<td>Map BDL STRING or VARCHAR to XML Schema idRef.</td>
</tr>
<tr>
<td>XSDIDREFS</td>
<td>Map BDL STRING or VARCHAR to XML Schema idRefs.</td>
</tr>
<tr>
<td>XSDInt</td>
<td>Map BDL INTEGER or BIGINT to XML Schema int.</td>
</tr>
<tr>
<td>XSDInteger</td>
<td>Map BDL DECIMAL to XML Schema integer.</td>
</tr>
<tr>
<td>XSDLanguage</td>
<td>Map BDL STRING or VARCHAR to XML Schema language.</td>
</tr>
<tr>
<td>XSDLon</td>
<td>Map BDL BIGINT or DECIMAL to XML Schema long.</td>
</tr>
<tr>
<td>XSDNCName</td>
<td>Map BDL STRING or VARCHAR to XML Schema NCName.</td>
</tr>
<tr>
<td>XSDName</td>
<td>Map BDL STRING or VARCHAR to XML Schema Name.</td>
</tr>
<tr>
<td>XSDNegativeInteger</td>
<td>Map BDL DECIMAL to XML Schema negativeInteger.</td>
</tr>
<tr>
<td>XSDNMTOKEN</td>
<td>Map BDL STRING or VARCHAR to XML Schema NMToken.</td>
</tr>
<tr>
<td>XSDNMTOKENS</td>
<td>Map BDL STRING or VARCHAR to XML Schema NMTokens.</td>
</tr>
<tr>
<td>XSDNonNegativeInteger</td>
<td>Map BDL DECIMAL to XML Schema nonNegativeInteger.</td>
</tr>
<tr>
<td>XSDNonPositiveInteger</td>
<td>Map BDL DECIMAL to XML Schema nonPositiveInteger.</td>
</tr>
<tr>
<td>XSDNormalizedString</td>
<td>Map BDL STRING or VARCHAR to XML Schema normalizedString.</td>
</tr>
<tr>
<td>XSDNotation</td>
<td>Not supported.</td>
</tr>
<tr>
<td>XSDPositiveInteger</td>
<td>Map BDL DECIMAL to XML Schema positiveInteger.</td>
</tr>
<tr>
<td>XSDQName</td>
<td>Map BDL STRING or VARCHAR to XML Schema QName.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Definition</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>XSDShort</td>
<td>Map BDL SMALLINT or BIGINT to XML Schema short.</td>
</tr>
<tr>
<td>XSDString</td>
<td>Map BDL STRING, Char, Text or VarChar to XML Schema string.</td>
</tr>
<tr>
<td>XSDTime</td>
<td>Map BDL DATETIME to XML Schema time.</td>
</tr>
<tr>
<td>XSDToken</td>
<td>Map BDL STRING or VARCHAR to XML Schema token.</td>
</tr>
<tr>
<td>XSDUnsignedByte</td>
<td>Map BDL SMALLINT or BIGINT to XML Schema unsignedByte.</td>
</tr>
<tr>
<td>XSDUnsignedInt</td>
<td>Map BDL BIGINT or DECIMAL to XML Schema unsignedInt.</td>
</tr>
<tr>
<td>XSDUnsignedLong</td>
<td>Map BDL DECIMAL to XML Schema unsignedLong.</td>
</tr>
<tr>
<td>XSDUnsignedShort</td>
<td>Map BDL INTEGER or BIGINT to XML Schema unsignedShort.</td>
</tr>
</tbody>
</table>

**XSDAnySimpleType**

Map BDL STRING or VARCHAR to XML Schema anySimpleType.

**XSDAnyType**

Map BDL STRING or VARCHAR to XML Schema anyType.

**XSDAnyURI**

Map BDL STRING or VARCHAR to XML Schema anyURI.

**XSDBase64binary**

Map BDL BYTE to XML Schema base64binary.

**Example**

```sql
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
    val1 BYTE ATTRIBUTES(XSDBase64binary,XMLName="Val")
END RECORD

<Role>
    <Val>F0FFC8D27FF001547FC219E1FFF009F0FFC8D27FF001547D</Val>
</Role>
```

**XSDBase64binary**

Map BDL BOOLEAN, SMALLINT or INTEGER to XML Schema boolean.

**Example**

```sql
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
    val INTEGER ATTRIBUTES(XSDBase64binary,XMLName="Val")
END RECORD

<Role>
    <Val>true</Val>
</Role>
```
### XSDByte
Map BDL TINYINT, SMALLINT or BIGINT to XML Schema `byte`.

**Example**

```plaintext
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
    val1 SMALLINT ATTRIBUTES(XSDByte,XMLName="Val")
END RECORD

<Root>
    <Val>-126</Val>
</Root>
```

### XSDDate
Map BDL DATE or DATETIME to XML Schema `date`.

**Example**

```plaintext
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
    val1 DATE ATTRIBUTES(XSDDate,XMLName="Val")
END RECORD

<Root>
    <Val>2006-06-29+01:00</Val>
</Root>
```

### XSDDateTime
Map BDL DATETIME to XML Schema `dateTime`.

**Example**

```plaintext
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
    val1 DATETIME ATTRIBUTES(XSDDateTime,XMLName="Val")
END RECORD

<Root>
    <Val>2006-06-29T09:35:26.13584+01:00</Val>
</Root>
```

### XSDDecimal
Map BDL DECIMAL to XML Schema `decimal`.

**Example**

```plaintext
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
    val1 DECIMAL(5,3) ATTRIBUTES(XSDDecimal,XMLName="Val")
END RECORD

<Root>
    <Val>12.345</Val>
</Root>
```

### XSDDouble
Map BDL FLOAT to XML Schema `double`.

**Example**

```plaintext
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
    val1 DECIMAL(5,3) ATTRIBUTES(XSDDecimal,XMLName="Val")
END RECORD

<Root>
    <Val>12.345</Val>
</Root>
```
Example

```sql
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 FLOAT ATTRIBUTES(XSDDouble,XMLName="Val")
END RECORD

<Root>
  <Val>12.78e-2</Val>
</Root>
```

**XSDDuration**

Map BDL INTERVAL to XML Schema `duration`.

Example

```sql
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 INTERVAL DAY TO SECOND ATTRIBUTES(XSDDuration,XMLName="Val")
END RECORD

<Root>
  <Val>P3DT10H30M45S</Val>
</Root>
```

**XSDEntities**

Map BDL STRING or VARCHAR to XML Schema `ENTITIES`.

**XSDEntity**

Map BDL STRING or VARCHAR to XML Schema `ENTITY`.

**XSDFloat**

Map BDL SMALLFLOAT to XML Schema `float`.

Example

```sql
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 SMALLFLOAT ATTRIBUTES(XSDFloat,XMLName="Val")
END RECORD

<Root>
  <Val>126.435</Val>
</Root>
```

**XSDGDay**

Map BDL DATETIME to XML Schema `gDay`.

Example

```sql
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 DATETIME DAY TO DAY ATTRIBUTES(XSDGDay,XMLName="Val")
END RECORD

<Root>
  <Val>---25</Val>
</Root>
```
**XSDGMonth**

Map BDL DATETIME to XML Schema `gMonth`.

**Example**

```xml
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 DATETIME MONTH TO MONTH ATTRIBUTES(XSDGMonth,XMLName="Val")
END RECORD

<Root>
  <Val>--12</Val>
</Root>
```

**XSDGMonthDay**

Map BDL DATETIME to XML Schema `gMonthDay`.

**Example**

```xml
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 DATETIME MONTH TO DAY ATTRIBUTES(XSDGMonthDay,XMLName="Val")
END RECORD

<Root>
  <Val>--12-31</Val>
</Root>
```

**XSDGYear**

Map BDL DATETIME to XML Schema `gYear`.

**Example**

```xml
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 DATETIME YEAR TO YEAR ATTRIBUTES(XSDGYear,XMLName="Val")
END RECORD

<Root>
  <Val>2006</Val>
</Root>
```

**XSDGYearMonth**

Map BDL DATETIME to XML Schema `gYearMonth`.

**Example**

```xml
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 DATETIME YEAR TO MONTH ATTRIBUTES(XSDGYearMonth,XMLName="Val")
END RECORD

<Root>
  <Val>2006-06</Val>
</Root>
```

**XSDHexBinary**

Map BDL BYTE to XML Schema `hexBinary`.
**Example**

```plaintext
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 BYTE ATTRIBUTES(XSDHexBinary,XMLName="Val")
END RECORD

<Root>
  <Val>0FB6</Val>
</Root>
```

**XSDID**

Map BDL STRING or VARCHAR to XML Schema **ID**.

**XSDIDREF**

Map BDL STRING or VARCHAR to XML Schema **IDREF**.

**XSDIDREFS**

Map BDL STRING or VARCHAR to XML Schema **IDREFS**.

**XSDInt**

Map BDL INTEGER or BIGINT to XML Schema **int**.

**Example**

```plaintext
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 INTEGER ATTRIBUTES(XSDInt,XMLName="Val")
END RECORD

<Root>
  <Val>-1258</Val>
</Root>
```

**XSDInteger**

Map BDL DECIMAL to XML Schema **integer**.

**Example**

```plaintext
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 DECIMAL(32,0) ATTRIBUTES(XSDInteger,XMLName="Val")
END RECORD

<Root>
  <Val>12678</Val>
</Root>
```

**XSDLanguage**

Map BDL STRING or VARCHAR to XML Schema **language**.

**XSDLong**

Map BDL BIGINT or DECIMAL to XML Schema **long**.

**Example**

```plaintext
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 DECIMAL(19,0) ATTRIBUTES(XSDLong,XMLName="Val")
END RECORD
```
Web services

END RECORD

```
<Root>
    <Val>1267488</Val>
</Root>
```

**XSDNCName**
Map BDL STRING or VARCHAR to XML Schema **NCName**.

**XSDName**
Map BDL STRING or VARCHAR to XML Schema **Name**.

**XSDNegativeInteger**
Map BDL DECIMAL to XML Schema **negativeInteger**.

**Example**

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
    val1 DECIMAL(32,0) ATTRIBUTES(XSDNegativeInteger,XMLName="Val")
END RECORD
```

```
<Root>
    <Val>-4828</Val>
</Root>
```

**XSDNMTOKEN**
Map BDL STRING or VARCHAR to XML Schema **NMToken**.

**XSDNMTOKENS**
Map BDL STRING or VARCHAR to XML Schema **NMTokens**.

**XSDNonNegativeInteger**
Map BDL DECIMAL to XML Schema **nonNegativeInteger**.

**Example**

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
    val1 DECIMAL(32,0) ATTRIBUTES(XSDNonNegativeInteger,XMLName="Val")
END RECORD
```

```
<Root>
    <Val>1589</Val>
</Root>
```

**XSDNonPositiveInteger**
Map BDL DECIMAL to XML Schema **nonPositiveInteger**.

**Example**

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
    val1 DECIMAL(32,0) ATTRIBUTES(XSDNonPositiveInteger,XMLName="Val")
END RECORD
```

```
<Root>
    <Val>-8574</Val>
</Root>
**XSDNormalizedString**
Map BDL STRING or VARCHAR to XML Schema `normalizedString`.

**XSDnotation**
Not supported.

**XSDPositiveInteger**
Map BDL DECIMAL to XML Schema `positiveInteger`.

**Example**

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
   val1 DECIMAL(32,0) ATTRIBUTES(XSDPositiveInteger,XMLName="Val")
END RECORD

<Root>
  <Val>+41893</Val>
</Root>
```

**XSDQName**
Map BDL STRING or VARCHAR to XML Schema `QName`.

**XSDShort**
Map BDL SMALLINT or BIGINT to XML Schema `short`.

**Example**

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
   val1 SMALLINT ATTRIBUTES(XSDShort,XMLName="Val")
END RECORD

<Root>
  <Val>12678</Val>
</Root>
```

**XSDString**
Map BDL STRING, CHAR, TEXT or VARCHAR to XML Schema `string`.

**Example**

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
   val1 STRING ATTRIBUTES(XSDString,XMLName="Val")
END RECORD

<Root>
  <Val>Hello world, how are you ?</Val>
</Root>
```

**XSDTime**
Map BDL DATETIME to XML Schema `time`.

**Example**

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
   val1 DATETIME ATTRIBUTES(XSDTime,XMLName="Val")
END RECORD

<Root>
  <Val>Hello world, how are you ?</Val>
</Root>
```
**XSDToken**
Map BDL STRING or VARCHAR to XML Schema `token`.

**XSDUnsignedByte**
Map BDL SMALLINT or BIGINT to XML Schema `unsignedByte`.

**Example**

```xml
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
    val1 SMALLINT ATTRIBUTES(XSDUnsignedByte,XMLName="Val")
END RECORD
```

```xml
<Root>
    <Val>254</Val>
</Root>
```

**XSDUnsignedInt**
Map BDL BIGINT or DECIMAL to XML Schema `unsignedInt`.

**Example**

```xml
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
    val1 DECIMAL(32,0) ATTRIBUTES(XSDUnsignedInt,XMLName="Val")
END RECORD
```

```xml
<Root>
    <Val>1267896754</Val>
</Root>
```

**XSDUnsignedLong**
Map BDL DECIMAL to XML Schema `unsignedLong`.

**Example**

```xml
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
    val1 DECIMAL(32,0) ATTRIBUTES(XSDUnsignedLong,XMLName="Val")
END RECORD
```

```xml
<Root>
    <Val>12678967543233</Val>
</Root>
```

**XSDUnsignedShort**
Map BDL INTEGER or BIGINT to XML Schema `unsignedShort`.

**Example**

```xml
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
    val1 INTEGER ATTRIBUTES(XSDUnsignedShort,XMLName="Val")
END RECORD
```

```xml
<Root>
    <Val>12678967543233</Val>
</Root>
XML facet constraint attributes

The following attributes are facet constraints depending on the XSD data type used on a simple BDL variable to restrict the allowed value-space.

(Notice that some attributes are allowed only on some XSD data types).

Several facet constraints can be set on the same data type, and a mandatory values is expected (for example, XSDMinLength="8").

Table 744: Facet constraints between simple BDL and XML data types

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSDLength</td>
<td>Define the exact number of XML character or bytes.</td>
</tr>
<tr>
<td>XSDMinLength</td>
<td>Define the minimum number of XML character or bytes.</td>
</tr>
<tr>
<td>XSDMaxLength</td>
<td>Define the maximum number of XML character or bytes.</td>
</tr>
<tr>
<td>XSDEnumeration</td>
<td>Define a list of allowed values separated by the character</td>
</tr>
<tr>
<td>XSDWhiteSpace</td>
<td>Perform a XML string manipulation before serialization or deserialization.</td>
</tr>
<tr>
<td>XSDPattern</td>
<td>Define the regular expression the value has to match.</td>
</tr>
<tr>
<td>XSDMinInclusive</td>
<td>Define the inclusive minimum value based on the data type where it is set.</td>
</tr>
<tr>
<td>XSDMaxInclusive</td>
<td>Define the inclusive maximum value based on the data type where it is set.</td>
</tr>
<tr>
<td>XSDMinExclusive</td>
<td>Define the exclusive minimum value based on the data type where it is set.</td>
</tr>
<tr>
<td>XSDMaxExclusive</td>
<td>Define the exclusive maximum value based on the data type where it is set.</td>
</tr>
<tr>
<td>XSDTotalDigits</td>
<td>Define the total number of digits.</td>
</tr>
<tr>
<td>XSFractionDigits</td>
<td>Define the number of digits of the fraction part.</td>
</tr>
</tbody>
</table>

XSDLength

Restrict the length of the data to the exact number of XML characters allowed when set on a BDL STRING, VARCHAR, CHAR or TEXT, or the number of bytes allowed when set on a BDL BYTE.

Note:

1. XSDMinLength and XSDMaxLength can be used together, but XSDMaxLength value must be greater then XSDMinLength
2. XSDMaxLength cannot be used with XSDLength

Example

```
DEFINE myStr STRING ATTRIBUTES(XSDString, XSDLength="12",
```
XSDMinLength
Restrict the length of the data to the minimum number of XML characters allowed when set on a BDL STRING, VARCHAR, CHAR or TEXT, or the number of bytes allowed when set on a BDL BYTE.

Note:
1. XSDMinLength and XSDMaxLength can be used together, but XSDMaxLength value must be greater than XSDMinLength
2. XSDMaxLength cannot be used with XSDLLength

Example

```bdl
DEFINE myStr STRING ATTRIBUTES(XSDString, XSDMinLength="12", XMLName="MyString")
DEFINE myByte BYTE ATTRIBUTES(XSDBase64Binary, XSDMinLength="8000", XMLName="MyPicture")
```

XSDMaxLength
Restrict the length of the data to the maximum number of XML characters allowed when set on a BDL STRING, VARCHAR, CHAR or TEXT, or the number of bytes allowed when set on a BDL BYTE.

Note:
1. XSDMinLength and XSDMaxLength can be used together, but XSDMaxLength value must be greater than XSDMinLength
2. XSDMaxLength cannot be used with XSDLLength

Example

```bdl
DEFINE myStr STRING ATTRIBUTES(XSDString, XSDMaxLength="12", XMLName="MyString")
DEFINE myByte BYTE ATTRIBUTES(XSDBase64Binary, XSDMaxLength="8000", XMLName="MyPicture")
```

XSDEnumeration
Restrict the allowed value-space to a list of values separated by the characters |.

Note:
1. To escape the separator character, simply double it like the following ||
2. This attribute can be set on any simple BDL variable excepted on XSDBoolean.

Example

```bdl
DEFINE myStr STRING ATTRIBUTES(XSDString, XSDEnumeration="one|two|three|four", XMLName="MyString")
```

```bdl
DEFINE myDec DECIMAL(3,1) ATTRIBUTES(XSDDecimal, XSDEnumeration="12.1|11.8|-24.7", XMLName="MyDecimal")
```
**XSDWhiteSpace**

Perform a XML string manipulation before serialization or deserialization depending on the following possible values:

- **preserve**: the XML string is not modified.
- **replace**: the XML string is modified by replacing each `\n`, `\t`, `\r` by a single space.
- **collapse**: the XML string is modified by replacing each `\n`, `\t`, `\r` by a single space, then each sequence of several spaces are replaced by one single space. Leading and trailing spaces are removed too.

**Note:**

1. The whiteSpace facet is always performed before any other facet constraints, or serialization or deserialization process.
2. For any BDL variable excepted STRING, CHAR and VARCHAR, only collapse is allowed.

**Example**

```nbdkind
DEFINE myStr STRING ATTRIBUTES(XSDString, XSDWhiteSpace="replace", XMLName="MyString")

DEFINE myDec DECIMAL(3,1) ATTRIBUTES(XSDDecimal, XSDWhiteSpace="collapse", XMLName="MyDecimal")
```

**XSDPattern**

Define a regular expression the value has to match to be serialized or deserialized without any error.

**Note:**

1. The regular expression is defined in the XML Schema Part 2 specification available [here](#).
2. Backslash characters `\` in a regular expression must be escaped by duplicating it.

**Example**

```nbdkind
DEFINE myStr STRING ATTRIBUTES(XSDString, XSDPattern="A.*Z", XMLName="MyString")

DEFINE myZipCode INTEGER ATTRIBUTES(XSDInt, XSDPattern="[0-9]{5}", XMLName="MyZipCode")

DEFINE myOtherZipCode INTEGER ATTRIBUTES(XSDInt, XSDPattern="\\d{5}", XMLName="myOtherZipCode") # regex is \d{5} see note
```

**XSDMinInclusive**

Define the minimum inclusive value allowed and depending on the data type where it is set, namely all numeric, date and time data types.

**Note:** The minimum value cannot exceed the implicit minimum value supported by the data type itself or the compiler will complain. For instance, with XSDShort the minimum value is -32768.

**Example**

```nbdkind
DEFINE myCode SMALLINT ATTRIBUTES(XSDShort, XSDMinInclusive="-1000", XMLName="MyCode")

DEFINE myRate DECIMAL(4,2) ATTRIBUTES(XSDDecimal, XSDMinInclusive="100.01", XMLName="MyRate")
```
XSDMaxInclusive
Define the maximum inclusive value allowed and depending on the data type where it is set, namely all numeric, date and time data types.

Note: The maximum value cannot exceed the implicit maximum value supported by the data type itself or the compiler will complain. For instance, with XSDShort the maximum value is 32767.

Example

```plaintext
DEFINE myCode SMALLINT ATTRIBUTES(XSDShort, XSDMaxInclusive="1000", XMLName="MyCode")
```

```plaintext
DEFINE myRate DECIMAL(4,2) ATTRIBUTES(XSDDecimal, XSDMaxInclusive="299.99", XMLName="MyRate")
```

XSDMinExclusive
Define the minimum exclusive value allowed and depending on the data type where it is set, namely all numeric, date and time data types.

Note: The minimum value cannot exceed or be equal to the implicit minimum value supported by the data type itself or the compiler will complain. For instance, with XSDShort the minimum value is -32768.

Example

```plaintext
DEFINE myCode SMALLINT ATTRIBUTES(XSDShort, XSDMinExclusive="-1000", XMLName="MyCode")
```

```plaintext
DEFINE myRate DECIMAL(4,2) ATTRIBUTES(XSDDecimal, XSDMinExclusive="100.01", XMLName="MyRate")
```

XSDMaxExclusive
Define the maximum exclusive value allowed and depending on the data type where it is set, namely all numeric, date and time data types.

Note: The maximum value cannot exceed or be equal to the implicit maximum value supported by the data type itself or the compiler will complain. For instance, with XSDShort the maximum value is 32767.

Example

```plaintext
DEFINE myCode SMALLINT ATTRIBUTES(XSDShort, XSDMaxExclusive="1000", XMLName="MyCode")
```

```plaintext
DEFINE myRate DECIMAL(4,2) ATTRIBUTES(XSDDecimal, XSDMaxExclusive="299.99", XMLName="MyRate")
```

XSDTotalDigits
Define the maximum number of digits allowed on a numeric data type, fraction part inclusive if there is one.

Note:
1. The total digits value cannot be equal or lower then 0.
2. On a BDL decimal, the total digits value cannot be lower than the precision of the BDL decimal itself.
3. Notice that a decimal without any precision and scale value is a decimal(16), therefore the total digits value must be equal or greater than 16.

Example

```plaintext
DEFINE myCode SMALLINT ATTRIBUTES(XSDShort, XSDTotalDigits="4", XMLName="MyCode")
```
**XSDMaxExclusive**

Define the maximum number of digits allowed on the fraction part of a numeric data type.

**Note:**

1. The fraction digits value set on a BDL data type without XSDDecimal set, can only be 0.
2. On a BDL DECIMAL, the fraction digits value cannot be lower than the scale of the BDL DECIMAL itself, and must be lower than the XSDTotalDigits value if set.

**Example**

```plaintext
DEFINE myCode SMALLINT ATTRIBUTES(XSDShort, XSDFractionDigits="0", XSDMaxExclusive="1000", XMLName="MyCode")
DEFINE myRate DECIMAL(4,2) ATTRIBUTES(XSDDecimal, XSDTotalDigits="5", XSDMaxExclusive="299.99", XMLName="MyRate")
```

**Customizing XML serialization**

The following attributes are used to change the default serialization of BDL into XML, and vice versa. Some attributes cannot have values, some may have values, and some attributes have mandatory values.

The following attributes cannot have values:

**Table 745: XML Serialization customizing - Attributes that cannot have values**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMLOptional</td>
<td>Define whether the variable can be missing.</td>
</tr>
<tr>
<td>XMLNillable on page 3540</td>
<td>Define an XML element to be explicitly null and serialized with the xsi:nil=&quot;true&quot; value.</td>
</tr>
<tr>
<td>XMLElement</td>
<td>Map a BDL simple data type to an XML Element.</td>
</tr>
<tr>
<td>XMLElementNillable on page 3542</td>
<td>Define the default for all element members in a RECORD (defined by TYPE or DEFINE) to be serialized as xsi:nil=&quot;true&quot; if NULL.</td>
</tr>
<tr>
<td>XMLAttribute</td>
<td>Map a BDL simple data type to an XML Attribute.</td>
</tr>
<tr>
<td>XMLBase</td>
<td>Set the base type of an XML Schema simpleContent.</td>
</tr>
<tr>
<td>XMLAll</td>
<td>Map a BDL Record to an XML Schema all structure.</td>
</tr>
<tr>
<td>XMLChoice</td>
<td>Map a BDL Record to an XML Schema choice structure.</td>
</tr>
<tr>
<td>XMLSequence</td>
<td>Map a BDL Record to an XML Schema sequence structure.</td>
</tr>
<tr>
<td>XMLSimpleContent</td>
<td>Map a BDL Record to an XML Schema simpleContent structure.</td>
</tr>
<tr>
<td>XSComplexType</td>
<td>Map a BDL Record type definition to an XML Schema complexType.</td>
</tr>
<tr>
<td>XMLList</td>
<td>Map a one-dimensional array to an XML Schema list.</td>
</tr>
<tr>
<td>XMLSelector</td>
<td>Define which member of an XMLChoice record is selected.</td>
</tr>
<tr>
<td>XMLAny</td>
<td>Map a xml.DomDocument object to a wildcard XML element node.</td>
</tr>
<tr>
<td>XMLAnyAttribute</td>
<td>Map a BDL one-dimensional dynamic array of a record with 3 strings to XML wildcard attributes.</td>
</tr>
</tbody>
</table>
Values are mandatory for the following attributes: (for example, XMLName="myname")

Table 746: XML Serialization customizing - Attributes that must have values

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMLName</td>
<td>Define the XML Name of a variable in an XML document.</td>
</tr>
<tr>
<td>XMLNamespace</td>
<td>Define the XML Namespace of a variable in an XML document.</td>
</tr>
<tr>
<td>XMLType</td>
<td>Force the XML type name of a variable.</td>
</tr>
<tr>
<td>XMLTypenamespace</td>
<td>Force the XML type namespace of a variable.</td>
</tr>
<tr>
<td>XSTypename</td>
<td>Define the XML Type Name of a BDL type definition.</td>
</tr>
<tr>
<td>XSTypenamespace</td>
<td>Define the XML Type Namespace of a BDL type definition.</td>
</tr>
<tr>
<td>XMLElementNamespace</td>
<td>Define the default XML namespace of all children defined as XMLElement in a</td>
</tr>
<tr>
<td></td>
<td>Record.</td>
</tr>
<tr>
<td>XMLAttributeNameSpace</td>
<td>Define the default XML namespace of all children defined as XMLAttribute in a</td>
</tr>
<tr>
<td></td>
<td>Record.</td>
</tr>
</tbody>
</table>

Values may be required for the following attributes: (for example, XMLOptimizedContent="image/*")

Table 747: XML Serialization customizing - Attributes that may have values

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMLOptimizedContent</td>
<td>Set on STRING or BYTE data type so that such string content represents a file on disk to be transmitted as base64 binary in SOAP via HTTP attachment.</td>
</tr>
</tbody>
</table>

XMLOptional

Define whether a variable can be omitted or not. It specifies how a NULL value is interpreted in XML if it is optional. For an option to set an XML node nillable when the BDL variable is NULL but not optional, see the XMLNillable on page 3540 attribute.

Note:
1. The attribute cannot be set on a type definition.
2. The attribute cannot be set if the main variable is not a RECORD.

Example

The variable "ValTwo" is optional when null.

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
   val1 INTEGER ATTRIBUTES(XSDint,XMLName="ValOne"),
   val2 FLOAT ATTRIBUTES(XSDdouble,XMLName="ValTwo",XMLOptional)
END RECORD
```

In the resulting XML document, "ValTwo" is defined when not NULL

```
<Root>
   <ValOne>458</ValOne>
   <ValTwo>58.48</ValTwo>
</Root>
```

In the resulting XML document, "ValTwo" is not defined when NULL.

```
<Root>
   <ValOne>458</ValOne>
</Root>
```
XMLNillable

Define an XML element to be explicitly null and serialized with the `xsi:nil="true"` value. It specifies how a NULL value is set on an XML node when the BDL variable is NULL but not optional.

**Note:** The XMLNillable attribute cannot be set on a TYPE definition. If you need to set nillable in a record defined from a database schema with LIKE clause, the XMLElementNillable on page 3542 attribute supports an implementation that sets all elements in the record to XMLNillable.

The XMLNillable attribute is used on its own, or in combination with the XMLOptional on page 3539 attribute to specify conditions for the XML node depending on how you want the BDL variable value to be serialized:

- XML node optional (when null)
- XML node nillable (when null)
- Option of either XML node optional or nillable, with preference for nillable.

Examples are shown for each use.

**Example optional**

The variable "val2" is optional when null

```bdd
DEFINE var RECORD
  val1 STRING,
  val2 INTEGER ATTRIBUTES(XMLOptional),
  val3 FLOAT
END RECORD
```

The resulting XML document is:

```xml
<var>
  <val1>Hello</val1>
  <val3>3.1415</val3>
</var>
```

**Example nillable**

The variable "val2" is nillable when BDL variable is NULL but not optional.

```bdd
DEFINE var RECORD
  val1 STRING,
  val2 INTEGER ATTRIBUTES(XMLNillable),
  val3 FLOAT
END RECORD
```

In the resulting XML document "val2" is serialized with the `xsi:nil="true"`

```xml
<var>
  <val1>Hello</val1>
  <val2 xsi:nil="true"/>
  <val3>3.1415</val3>
</var>
```

**Example optional or nillable**

The variable "val2" is either optional or nillable if BDL variable is NULL.

```bdd
DEFINE var RECORD
  val1 STRING,
  val2 INTEGER ATTRIBUTES(XMLNillable,XMLOptional),
  val3 FLOAT
END RECORD
```
In the resulting XML document "val2" is missing as it is defined by XMLOptional (the default).

```xml
<var>
  <val1>Hello</val1>
  <val3>3.1415</val3>
</var>
```

Or the resulting XML document can show "val2" nillable:

```xml
<var>
  <val1>Hello</val1>
  <val2 xsi:nil="true"/>
  <val3>3.1415</val3>
</var>
```

**Note:** If both XMLOptional and XMLNillable are set and the BDL variable is NULL, the serialization process from BDL to XML uses XMLOptional by default. If you prefer to serialize as xsi:nil="true", you must specify the "preferred" value as a parameter of XMLNillable.

```bdl
DEFINE var RECORD
  val1 STRING,
  val2 INTEGER ATTRIBUTES(XMLNillable="preferred", XMLOptional),
  val3 FLOAT
END RECORD
```

### XML transformation into BDL

When getting an XML transformation into BDL, the Web Service engine will not raise an error and the variable is set correctly whether the variable tag is missing or has xsi:nil="true" set. In either case, the BDL variable is set as NULL.

**XMLElement (Optional)**

Map a BDL simple data type to an XML Element.

**Note:** The attribute cannot be set on a type definition.

**Example**

```bdl
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 INTEGER ATTRIBUTES(XMLElement, XSDunsignedShort, XMLName="Val1"),
  rec RECORD ATTRIBUTES(XMLName="Rec")
    val2 FLOAT ATTRIBUTES(XMLElement, XMLName="Val2"),
    val3 STRING ATTRIBUTES(XMLElement, XMLName="Val3")
END RECORD
END RECORD
```

```xml
<Root>
  <Val1>148</Val1>
  <Rec1>
    <Val2>25.8</Val2>
    <Val3>Hello world</Val3>
  </Rec1>
</Root>
```
XMLElementNillable

The XMLElementNillable attribute specifies that all members of a RECORD have to be serialized as xsi:nil="true" when NULL. This has the same effect as defining each individual element with the XMLNillable attribute.

**Note:** The XMLElementNillable attribute can only be set on records defined by TYPE or DEFINE.

The XMLElementNillable attribute supports the record defined with LIKE clause to use the table definition from a database schema:

```define var RECORD ATTRIBUTE(XMLElementNillable) LIKE customer.*```

If columns in the database table are allowed to be null, you must add the XMLElementNillable attribute to the record definition.

The XMLElementNillable attribute is not inherited by sub records.

**Basic example**

```define var RECORD ATTRIBUTE(XMLElementNillable)
   val1 STRING,
   val2 STRING
END RECORD```

```let var.val1 = "Hello"
let var.val2 = NULL```

In the resulting XML document, "val2" is serialized with the xsi:nil="true":

```<var>
   <val1>Hello</val1>
   <val2 xsi:nil="true"/>
</var>```

**Example using a sub record**

Since XMLElementNillable attribute is not inherited by sub records, you must set it on each sub-record if needed:

```define var RECORD ATTRIBUTE(XMLElementNillable)
   val1 STRING,
   val2 STRING,
   subrec RECORD
   val3 STRING
END RECORD
END RECORD```

In the resulting XML document, for instance, only "val1" and "val2" are serialized as xsi:nil="true" when NULL, but "val3" is not.

To have "val3" serialized as xsi:nil="true" when NULL, define:

```define var RECORD ATTRIBUTE(XMLElementNillable)
   val1 STRING,
   val2 STRING,
   subrec RECORD ATTRIBUTE(XMLElementNillable)
   val3 STRING
END RECORD
END RECORD```

**Examples with XMLNillable and XMLOptional**

The XMLElementNillable attribute does not allow values, so you cannot write XMLElementNillable="preferred" as you do for XMLNillable.
If `XMLOptional` on page 3539 is set on a record member, `XMLElementNillable` is not taken into account.

If you need `XMLNillable` to take precedence over `XMLOptional`, you must then specify `XMLNillable="preferred"` on the record member.

In the following sample, if "val2" is `NULL`, it will serialize to the tag not being present because `XMLOptional` hides `XMLElementNillable`. On the other hand, if "val3" is `NULL`, it will be serialized as `xsi:nil="true"`, because the record member is defined with `XMLOptional` and `XMLNillable` (preferred):

```
DEFINE var RECORD ATTRIBUTE(XMLElementNillable)
   val1 STRING,
   val2 STRING ATTRIBUTE(XMLOptional),
   val3 STRING ATTRIBUTE(XMLOptional,XMLNillable="preferred")
END RECORD
LET var.val1 = "Hello"
LET var.val2 = NULL
LET var.val3 = NULL
```

Will be serialized as:

```
<var>
   <val1>Hello</val1>
   <val3 xsi:nil="true"/>
</var>
```

**XMLAttribute**

Map a BDL simple data type to an XML Attribute.

**Note:**

1. The attribute cannot be set on a type definition.
2. The attribute can only be set on a RECORD's member.

**Example**

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
   val1 INTEGER ATTRIBUTES(XMLAttribute,XSDunsignedShort,XMLName="Val1"),
   rec RECORD ATTRIBUTES(XMLName="Rec1")
      val2 FLOAT ATTRIBUTES(XMLAttribute,XMLName="Val2"),
      val3 STRING ATTRIBUTES(XMLElement,XMLName="Val3")
END RECORD
END RECORD
```

```
<Root Val1="148">
   <Rec1 Val2="25.8">
      <Val3>Hello world</Val3>
   </Rec1>
</Root>
```

**XMLBase**

Define the simple BDL variable used as the base type of an XML Schema `simpleContent` structure.

The attribute can be set on one and only one member of a RECORD defined with the `XMLSimpleContent` attribute

**XMLALL**

Map a BDL Record to an XML Schema `all` structure.

The order in which the record members appear in the XML document is not significant.
Example

```xml
DEFINE myall RECORD ATTRIBUTES(XMLAll,XMLName="Root")
  val1  INTEGER ATTRIBUTES(XMLName="Val1"),
  val2  FLOAT   ATTRIBUTES(XMLAttribute,XMLName="Val2"),
  val3  STRING  ATTRIBUTES(XMLName="Val3")
END RECORD

<Root Val2="25.8">
  <Val3>Hello world</Val3>
  <Val1>148</Val1>
</Root>

<Root Val2="25.8">
  <Val1>148</Val1>
  <Val3>Hello world</Val3>
</Root>
```

XMLChoice

Map a BDL Record to an XML Schema choice structure. The choice of the record's member is performed at runtime, and changes dynamically depending on a mandatory member. This specific member must be of type SMALLINT or INTEGER, and have an XMLSelector attribute set. The XMLChoice attribute also supports a "nested" value that removes the surrounding XML tag.

Note:

1. Valid selector values are indexes referring to members considered as XML element nodes. All other values will raise XML runtime errors.
2. Nested choice records cannot be defined as main variables; there must always be a surrounding variable.

Example

```xml
DEFINE mychoice RECORD ATTRIBUTES(XMLChoice,XMLName="Root")
  val1  INTEGER   ATTRIBUTES(XMLName="Val1"),
  val2  FLOAT     ATTRIBUTES(XMLAttribute,XMLName="Val2"),
  sel   SMALLINT  ATTRIBUTES(XMLSelector),
  val3  STRING    ATTRIBUTES(XMLName="Val3")
END RECORD

Case where "sel" value is 4

<Root Val2="25.8">
  <Val3>Hello world</Val3>
</Root>

Case where "sel" value is 1

<Root Val2="25.8">
  <Val1>148</Val1>
</Root>

Nested example:

```xml
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1  INTEGER   ATTRIBUTES(XMLName="Val1"),
  val2  FLOAT     ATTRIBUTES(XMLAttribute,XMLName="Val2"),
  choice RECORD   ATTRIBUTES(XMLChoice="nested")
    choice1  INTEGER ATTRIBUTES(XMLName="ChoiceOne"),
    choice2  FLOAT   ATTRIBUTES(XMLName="ChoiceTwo"),
  nestedSel SMALLINT ATTRIBUTES(XMLSelector)
END RECORD
```
Case where "nestedSel" value is 1

```xml
<Root Val2="25.8">
  <Val1>148</Val1>
  <ChoiceOne>6584</ChoiceOne>
  <Val3>Hello world</Val3>
</Root>
```

Case where "nestedSel" value is 2

```xml
<Root Val2="25.8">
  <Val1>148</Val1>
  <ChoiceTwo>85.8</ChoiceTwo>
  <Val3>Hello world</Val3>
</Root>
```

**XMLSequence (Optional)**

Map a BDL RECORD to an XML Schema sequence structure. The order in which the record members appear in the XML document must match the order of the BDL RECORD. The XMLSequence attribute also supports a "nested" value that removes the surrounding XML tag.

**Note:** Nested sequence records cannot be defined as main variables; there must always be a surrounding variable.

**Example**

```verbatim
DEFINE mysequence RECORD ATTRIBUTES(XMLSequence,XMLName="Root")
  val1 INTEGER ATTRIBUTES(XMLName="Val1"),
  val2 FLOAT ATTRIBUTES(XMLAttribute,XMLName="Val2"),
  val3 STRING ATTRIBUTES(XMLName="Val3")
END RECORD

<Root Val2="25.8">
  <Val1>-859</Val1>
  <Val3>Hello world</Val3>
</Root>
```

Nested example:

```verbatim
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 INTEGER ATTRIBUTES(XMLName="Val1"),
  val2 FLOAT ATTRIBUTES(XMLAttribute,XMLName="Val2"),
  sequence RECORD ATTRIBUTES(XMLSequence="nested")
    seq1 INTEGER ATTRIBUTES(XMLName="SeqOne"),
    seq2 FLOAT ATTRIBUTES(XMLName="SeqTwo")
END RECORD,
  val3 STRING ATTRIBUTES(XMLName="Val3")
END RECORD

<Root Val2="25.8">
  <Val1>148</Val1>
  <SeqOne>6584</SeqOne>
  <SeqTwo>85.597</SeqTwo>
  <Val3>Hello world</Val3>
</Root>
```
**XMLSimpleContent**

Map a BDL RECORD to an XML Schema `simpleContent` structure.

**Note:** One member must have the XMLBase attribute; all other members must have an XMLAttribute attribute. If not, the compiler complains.

**Example**

```xml
define mysimpletype record attributes(xmlsimplecontent, xmlname="root")
    base string attributes(xmlbase),
    val1 integer attributes(xmlattribute, xmlname="val1"),
    val2 float attributes(xmlattribute, xmlname="val2")
end record

<root val1="148" val2="25.8"> hello </root>
```

**XSComplexType**

Map a BDL RECORD type definition to an XML Schema `complexType`.

**Note:** You can have one member as a nested sequence or choice, or as an XMLList array with a nested sequence or choice as the array's elements; all other members must have an XMLAttribute attribute. If not, the compiler raises an error.

**Example**

```xml
type mycomplextype record attributes(xscomplextype,
    xstypename="mycomplextype", xstypenamespace="http://tempuri.org")
    name dynamic array attributes(xmllist) of record
        attributes(xmlsequence="nested")
        firstname string attributes(xmlname="firstname"),
        lastname string attributes(xmlname="lastname")
end record,
    date date attributes(xmlattribute, xmlname="date")
end record

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://tempuri.org" elementFormDefault="qualified">
    <xsd:complexType name="mycomplextype">
        <xsd:sequence maxOccurs="unbounded">
            <xsd:element name="firstName" type="xsd:string" />
            <xsd:element name="lastName" type="xsd:string" />
        </xsd:sequence>
        <xsd:attribute name="Date" type="xsd:date" use="required"/>
    </xsd:complexType>
</xsd:schema>
```

**XMLList**

Map a one dimensional array to an XML Schema element that has more than one occurrence.

**Example**

```xml
define myvar record attributes(xmlname="root")
    val1 integer attributes(xmlname="val1"),
    list dynamic array attributes(xmllist) of string
        attributes(xmlname="element"),
    val2 float attributes(xmlname="val2")
```

END RECORD

<Root>
    <Val1>148</Val1>
    <Element>hello</Element>
    <Element>how</Element>
    <Element>are</Element>
    <Element>you</Element>
    <Val2>0.58</Val2>
</Root>

**Note:** It is not possible to define an XMLList attribute on a main array.

**Related concepts**

- **Serialize XML from a dynamic array** on page 3553
  The XMLList and XMLName attributes can be used to serialize dynamic arrays to XML and vice versa.

**XMLSelector**

Define the index of the candidate among all members of an XMLChoice record that will be serialized or deserialized at runtime.

The index starts at 1.

The selector data type must be a SMALLINT or a INTEGER.

**XMLAny**

Map a Xml.DomDocument object to a wildcard XML element:

```plaintext
DEFINE myVar RECORD ATTRIBUTES/XMLName="Root",XMLNamespace="http://tempuri.org"
  val1 INTEGER ATTRIBUTES/XMLName="Val1"),
  any Xml.DomDocument ATTRIBUTES/XMLAny,XMLNamespace="##other"),
  val2 FLOAT ATTRIBUTES/XMLName="Val2"
END RECORD
```

```xml
<pre:Root xmlns:pre="http://tempuri.org" >
    <pre:Val1>149</pre:Val1>
    <pre2:Doc xmlns:pre2="http://www.mycompany.com">
        <pre2:Element>how</pre2:Element>
        <pre2:Element>are</pre2:Element>
        <pre2:Element>you</pre2:Element>
    </pre2:Doc>
    <pre:Val2>0.58</pre:Val2>
</pre:Root>
```

**Note:** Associated with XMLAny, the XMLNamespace attribute requires either:

- A list of space-separated URIs to accept each attribute belonging to one of this namespace URI as a wildcard attribute.
- The value `##any` to accept any attribute as a wildcard attribute.
- The value `##other` to accept any attribute not in the main schema namespace as a wildcard attribute.

For example:

- If XMLNamespace="http://tmpuri.org http://www.mycompany.com", then only the XML documents belonging to one of those namespaces will be accepted and serialized (or deserialized) into the Xml.DomDocument object.
- If XMLNamespace="##any", then any XML document will be accepted and serialized (or deserialized) into the Xml.DomDocument object.
• If XMLNamespace="##other", then any XML document not belonging to the targetNamespace of the XML Schema where the any definition is used will be accepted and serialized (or deserialized) into the Xml.DomDocument object.

**XMLAnyAttribute**

Map a one-dimensional dynamic array to wildcard XML attributes.

**Example**

```plaintext
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root", XMLNamespace="http://tempuri.org")
  val1  INTEGER ATTRIBUTES(XMLName="Val1"),
  val2  FLOAT ATTRIBUTES(XMLName="Val2"),
  attr  STRING ATTRIBUTES(XMLName="Attr", XMLAttribute),
  any   DYNAMIC ARRAY ATTRIBUTES(XMLAnyAttribute, XMLNamespace="##other")OF
    RECORD
      ns  STRING,
      name  STRING,
      value  STRING
    END RECORD
END RECORD
```

```xml
<pre:Root xmlns:pre="http://tempuri.org" pre:Attr="10"
  xmlns:pre2="http://www.mycompany.com" pre2:AnyAttr1="10" pre2:AnyAttr2="">
  <pre:Val1>148</pre:Val1>
  <pre:Val2>0.58</pre:Val2>
</pre:Root>
```

**Note:**

1. The attribute **XMLAnyAttribute** is only allowed on a one-dimensional dynamic array of a record with three members of type STRING. The **first** member is for the namespace of the wildcard attribute, the **second** member is for the name of the wildcard attribute, and the **third** member is for the value of the wildcard attribute. **The name cannot be null.**

2. Associated with the XMLAnyAttribute, the XMLNamespace attribute requires either:
   - A list of space-separated URIs to accept each attribute belonging to one of the namespace URIs as a wildcard attribute.
   - The value **##any** to accept any attribute as a wildcard attribute.
   - The value **##other** to accept any attribute not in the main schema namespace as a wildcard attribute.

   For example:
   - If XMLNamespace="http://tmpuri.org http://www.mycompany.com", then only the attributes belonging to one of those namespaces will be accepted and serialized (or deserialized) into the array.
   - If XMLNamespace="##any", then any attribute will be accepted and serialized (or deserialized) into the array.
   - If XMLNamespace="##other", then any attributes not belonging to the targetNamespace of the XML Schema where the anyAttribute definition is used will be accepted and serialized (or deserialized) into the array.

**XMLName**

Define the name of a variable in an XML document.

**Note:** The attribute cannot be set on a type definition.

**Example**

```plaintext
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1  INTEGER ATTRIBUTES(XMLName="Val1"),
  val2  FLOAT,
  val3  INTEGER ATTRIBUTES(XMLName="Val3")
```
XMLNamespace

Define the namespace of a variable in an XML document.

Note:
1. If the attribute is set on a Record, by default all members defined as XMLElement of that record are in the same namespace.
2. If the attribute is set on an Array, by default all elements defined as XMLElement of that array are in the same namespace.
3. The attribute cannot be set on a type definition.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root", XMLNamespace="http://tempuri.org")
  attr1 INTEGER ATTRIBUTES(XMLAttribute,XMLName="Attr1"),
  val1 FLOAT ATTRIBUTES(XMLName="Val1", XMLNamespace="http://www.mycompany.com"),
  val2 INTEGER ATTRIBUTES(XMLName="Val2"),
  attr2 STRING ATTRIBUTES(XMLAttribute, XMLName="Attr2",
    XMLNamespace="http://anyuri.org")
END RECORD
```

```
<Root xmlns:fs1="http://tempuri.org" Attr1="158"
  xmlns:fs2="http://www.mycompany.com">0.5</fs1:Val2>
</fs1:Root>
```

XMLElement

The XMLList and XMLName attributes can be used to serialize dynamic arrays to XML and vice versa.

XMLType

Force the XML type name of a variable by adding xsi:type at serialization or by checking xsi:type at deserialization.

Note: The attribute must be used with the XMLTypenamespace attribute; otherwise, the compiler complains.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root",
  XMLNamespace="http://tempuri.org")
  val1 FLOAT ATTRIBUTES(XMLName="Val1"),
  val2 INTEGER ATTRIBUTES(XMLName="Val2",
    XMLType="MyRecord",
    XMLTypenamespace="http://mynamespace.org")
END RECORD
```

```
<fs1:Root xmlns:fs1="http://tempuri.org">
  <fs1:Val1>0.5</fs1:Val1>
</fs1:Root>
```
XMLTypenamespace

Force the XML type namespace of a variable by adding `xsi:type` at serialization or by checking `xsi:type` at de-serialization.

**Note:** The attribute must be used with the XMLType attribute; otherwise the compiler complains.

**Example**

```bdl
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root", 
XMLNamespace="http://tempuri.org")

  val1 FLOAT ATTRIBUTES(XMLName="Val1"),
  val2 INTEGER ATTRIBUTES(XMLName="Val2", 
                          XMLType="MyRecord", 
                          XMLTypenamespace="http://mynamespace.org")

END RECORD
```

```xml
<fjs1:Root xmlns:fjs1="http://tempuri.org">
    <fjs1:Val1>0.5</fjs1:Val1>
    <fjs1:Val2 xmlns:fjs2="http://mynamespace.org" xsi:type="fjs2:MyRecord">-18547</fjs1:Val2>
</fjs1:Root>
```

XSTypename

Define the XML Schema name of a BDL type definition.

**Note:**

1. The attribute must be used with the XSTypenamespace attribute; otherwise the compiler complains.
2. The attribute is only allowed on a type definition.

**Example**

```bdl
TYPE myType RECORD ATTRIBUTES(XMLSequence, 
XSTypeName="MyFirstType", 
XSTypeNamespace="http://tempuri.org")

  val1 FLOAT ATTRIBUTES(XMLElement,XMLName="Val1"),
  val2 INTEGER ATTRIBUTES(XMLElement,XMLName="Val2",XMLOptional),
  attr STRING ATTRIBUTES(XMLAttribute,XMLName="Attr")

END RECORD
```

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" targetNamespace="http://tempuri.org" elementFormDefault="qualified">
  <xsd:complexType name="MyFirstType">
    <xsd:sequence>
      <xsd:element name="Val1" type="xsd:double" />
      <xsd:element name="Val2" type="xsd:int" minOccurs="0" />
    </xsd:sequence>
    <xsd:attribute name="Attr" type="xsd:string" use="required" />
  </xsd:complexType>
</xsd:schema>
```

XSTypenamespace

Define the XML Schema namespace of a BDL type definition.

**Note:**

1. The attribute must be used with the XSType attribute; otherwise the compiler complains.
2. The attribute is only allowed on a type definition.

Example

```plaintext
TYPE myType RECORD ATTRIBUTES(XMLChoice,
    XSTypeName="MyFirstChoice",
    XSTypeNamespace="http://tempuri.org" )
val1  FLOAT  ATTRIBUTES(XMLElement,XMLName="Val1"),
val2  INTEGER ATTRIBUTES(XMLElement,XMLName="Val2",XMLOptional),
attr  STRING  ATTRIBUTES(XMLAttribute,XMLName="Attr",XMLOptional),
set   INTEGER ATTRIBUTES(XMLSelector)
END RECORD
```

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://tempuri.org"  elementFormDefault="qualified" >
    <xsd:complexType name="MyFirstChoice">
        <xsd:choice>
            <xsd:element name="Val1" type="xsd:double" />
            <xsd:element name="Val2" type="xsd:int" minOccurs="0" />
        </xsd:choice>
        <xsd:attribute name="Attr" type="xsd:string" />
    </xsd:complexType>
</xsd:schema>
```

XMLElementNamespace

Define the default namespace of all members of a record also defined as XML elements.

Example

```plaintext
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root",
    XMLNamespace="http://tempuri.org",
    XMLElementNamespace="http://www.mycompany.com")
val1  FLOAT  ATTRIBUTES(XMLElement,XMLName="Val1"),
val2  INTEGER ATTRIBUTES(XMLElement,XMLName="Val2"),
attr  STRING  ATTRIBUTES(XMLAttribute,XMLName="Attr"),
END RECORD
```

```xml
<fjs1:Root xmlns:fjs1="http://tempuri.org"
    xmlns:fjs2="http://www.mycompany.com">
    <fjs2:Val1>0.5</fjs2:Val1>
    <fjs2:Val2>-18547</fjs2:Val2>
</fjs1:Root>
```

XMLAttributeNamespace

Define the default namespace of all members of a record also defined as XML attributes.

Example

```plaintext
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root",
    XMLNamespace="http://tempuri.org",
    XMLAttributeNamespace="http://www.mycompany.com")
val1  FLOAT  ATTRIBUTES(XMLElement,XMLName="Val1"),
val2  INTEGER ATTRIBUTES(XMLElement,XMLName="Val2"),
attr  STRING  ATTRIBUTES(XMLAttribute,XMLName="Attr"),
attr2  DATE   ATTRIBUTES(XMLAttribute,XMLName="Attr2",XMLNamespace="http://anyuri.org"),
END RECORD
```

```xml
<fjs1:Root xmlns:fjs1="http://tempuri.org"
    xmlns:fjs2="http://www.mycompany.com"
    xmlns:fjs3="http://anyuri.org">
    <fjs2:Attr1>Hello</fjs2:Attr1>
    <fjs2:Val1>0.5</fjs2:Val1>
    <fjs2:Val2>-18547</fjs2:Val2>
</fjs1:Root>
```
XMLOptimizedContent

Set on STRING or BYTE data type so that such string content represents a file on disk to be transmitted as base64 binary in SOAP via HTTP attachment.

The XMLOptimizedContent attribute is allowed on BYTE or STRING data type.

The XMLOptimizedContent attribute adds value when used in one of these scenarios:

1. When MTOM is enabled and the XMLOptimizedContent attribute is set on a BYTE, it sets the variable that must be transmitted transparently as an HTTP part on the wire. See Message Transmission Optimization Mechanism (MTOM) on page 3297 for more information.
2. When MTOM is enabled and the XMLOptimizedContent attribute is set on a STRING, it contains the name of the file to be transmitted as a Base64 binary, with an optional mime-type hint, in order to avoid loading a file in a BYTE. See Message Transmission Optimization Mechanism (MTOM) on page 3297 for more information.
3. If the XMLOptimizedContent attribute is set to "swaRef" (XMLOptimizedContent="swaRef") and set on a STRING, the contents of the file are transmitted according to the SoapWithAttachmentRef specification. See swaRef (SOAP with attachments using wsi:swaRef) on page 3298 for more information.

If the XMLOptimizedContent attribute is set on a BDL STRING data type, it is expected to contain the name of the file that the Genero Web Service will load based on the current directory during transport of the MTOM request.

If the XMLOptimizedContent attribute is set on a BDL STRING data type, it will contain the absolute path of the file GWS has received during the transport of a MTOM response. The absolute path is based on the Genero temp directory settings and contains a UUID generated name. The programmer is in charge of moving the file or removing the file from disk.

The optional value is a hint to specify the mime-type of the data. The hint may be used by Java or .NET to generate different kinds of objects, depending on the information provided by the hint. For instance, on Java a mime type of "image" will generate a java.awt.Image object instead of a byte[] object.

Example

The Genero code:

```genero
define rec record
data1 byte attributes(xmloptimizedcontent,xmlname="mydata"),
date2 string attributes(xmloptimizedcontent="image/*",xmlname="myimage"),
data3 byte
end record
```

The XSD representation:

```xml
c<xsd:complexType name="rec">
  c<xsd:sequence>
    c<xsd:element type="xsd:base64Binary" name="MyData"/>
    c<xsd:element xmime:expectedContentTypes="image/*"
      type="xsd:base64Binary" name="MyImage"/>
    c<xsd:element type="xsd:base64Binary" name="data3"/>
  c</xsd:sequence>
 c</xsd:complexType>
```

Things to observe:

1. With the BYTE data type, regardless of whether the XMLOptimizedContent attribute is set, the same type (xsd:base64Binary) appears in the XSD.
2. The STRING holds the path to the document to transfer and is viewed as a BYTE in the XML exchange.
3. With the STRING data type, a mime type is specified.
Serialize XML from a dynamic array

The XMLList and XMLName attributes can be used to serialize dynamic arrays to XML and vice versa.

These attributes have distinct uses depending on the output you want.

The XMLList on page 3546 attribute is used especially to produce a list of the same elements from an array, without a surrounding tag. This output on its own is not valid XML because there is no parent node. For valid XML output, the attribute must be used inside another variable, such as a record. Compare Example 1: List of the same elements on page 3553 and Example 3: List inside a record on page 3554.

It is useful to use XMLList specifically to map a one dimensional array to an XML schema element that has more than one occurrence, inside another XML element.

An array (without XMLList), on the other hand, always produces a surrounding tag and thus a valid XML document. See Example 2: Array of the same elements on page 3553 and Example 4: Array inside a record on page 3554.

The XMLName on page 3548 attribute's function is to provide a name for XML elements. If you do not use the XMLName attribute, the tags used in the XML output are the names of the 4GL variables. Or if there are no variables, they are named "<element>".

Tip: The variables with these attributes you use to output to XML, may also be used to load an XML document into DYNAMIC ARRAY.

**Example 1: List of the same elements**

This example shows how an array defined with XMLList outputs a list of the same elements. Typically, this array is defined inside another variable (see Example 3: List inside a record on page 3554), because the resulting XML is not valid otherwise. The XMLName attribute is set to name the element.

```
DEFINE list DYNAMIC ARRAY(XMLList) OF STRING ATTRIBUTE(XMLName="MyElt")
```

This would produce an XML output like this:

```
<MyElt>One</MyElt><MyElt>Two</MyElt>
```

When viewed in an XML viewer, it would display as shown.

```
<MyElt>One</MyElt>
<MyElt>Two</MyElt>
```

**Example 2: Array of the same elements**

In this example an array is defined without XMLList. This produces output that is a valid XML document with an array of the same elements, and a surrounding tag. The XMLName attribute tags both the array and the elements.

```
DEFINE arr DYNAMIC ARRAY ATTRIBUTE(XMLName="SurroundingTag") OF STRING ATTRIBUTE(XMLName="MyElt")
```

The XML output may look like this:

```
<SurroundingTag><MyElt>One</MyElt><MyElt>Two</MyElt></SurroundingTag>
```

When viewed in an XML viewer, it would display as shown.

```
<SurroundingTag>
  <MyElt>One</MyElt>
  <MyElt>Two</MyElt>
</SurroundingTag>
```
Example 3: List inside a record

The XMLList must be defined inside another variable, such as a record, in order to produce output that is a valid XML document. Several lists may be included in the record.

```pascal
DEFINE myVar1 RECORD ATTRIBUTES(XMLName="Root")
  val1  INTEGER ATTRIBUTES(XMLName="Val1"),
  list  DYNAMIC ARRAY ATTRIBUTE(XMLList) OF STRING
    ATTRIBUTES(XMLName="MyElt"),
  val2  FLOAT   ATTRIBUTES(XMLName="Val2")
END RECORD
```

This would produce XML output like this:

```xml
<Root>
  <Val1>148</Val1>
  <MyElt>hello</MyElt>
  <MyElt>there</MyElt>
  <Val2>0.58</Val2>
</Root>
```

When viewed in an XML viewer, it would display as shown.

```
<Root>
  <Val1>148</Val1>
  <MyElt>hello</MyElt>
  <MyElt>there</MyElt>
  <Val2>0.58</Val2>
</Root>
```

Example 4: Array inside a record

This example shows that the array is output inside a surrounding tag inside the record.

```pascal
DEFINE myVar2 RECORD ATTRIBUTES(XMLName="Root")
  val1  INTEGER ATTRIBUTES(XMLName="Val1"),
  arr  DYNAMIC ARRAY ATTRIBUTES(XMLName="Sample") OF STRING
     ATTRIBUTES(XMLName="MyElt"),
  val2  FLOAT   ATTRIBUTES(XMLName="Val2")
END RECORD
```

This would produce XML output like this:

```xml
<Root>
  <Val1>148</Val1>
  <Sample>
    <MyElt>hello</MyElt>
    <MyElt>there</MyElt>
  </Sample>
  <Val2>0.58</Val2>
</Root>
```

When viewed in an XML viewer, it would display as shown.

```
<Root>
  <Val1>148</Val1>
  <Sample>
    <MyElt>hello</MyElt>
    <MyElt>there</MyElt>
  </Sample>
  <Val2>0.58</Val2>
</Root>
```

Error handling in GWS calls (STATUS)

When errors are encountered, the methods of GWS classes can throw exceptions and set the STATUS variable with the appropriate error number.

By default, the program will stop if an exception is thrown. You can trap the GWS errors with a WHENEVER ERROR handler or with a TRY/CATCH block. In the example, the readTextRequest() API is surrounded by a TRY/CATCH block:

```pascal
DEFINE req com.HTTPServiceRequest,
```
data STRING

LET req = com.WebServiceEngine.getHTTPServiceRequest(5)

TRY
  CALL req.readTextRequest() RETURNING data
CATCH
  CALL show_err(SFMT("Unexpected HTTP request read exception: %1", STATUS))
END TRY

For some errors, a human-readable description of the error code is available in the SQLCA.SQLERRM register.

**Related concepts**

Exceptions on page 502
Describes exception (error) handling in the programs.

**Related reference**

Genero BDL errors on page 3154
System error messages sorted by error number.

** Interruption handling in GWS calls (INT_FLAG)**

Genero Web Services (GWS) tests INT_FLAG to check if an application has been interrupted.

If INT_FLAG is set to TRUE, the DVM interrupts the GWS function processing and an exception is raised with error code -15553.

**Important:** Set the INT_FLAG register to FALSE before calling a GWS function. For example, after a dialog was stopped with a cancel action, the INT_FLAG is set to TRUE. If you do not reset INT_FLAG to FALSE, the next GWS function may be canceled.

As a general rule, surround GWS calls with a TRY/CATCH block (or WHENEVER ERROR handler), to detect both communication errors and interruptions.

```BDL
TRY
  LET INT_FLAG=FALSE
  ... req.sendXMLRequest(doc)
  ... CATCH
    CASE STATUS
    WHEN -15553 -- TCP socket error
      IF INT_FLAG THEN
        MESSAGE "An interruption occurred."
      ELSE
        ERROR "TCP socket error: ", SQLCA.SQLERRM
      END IF
    ELSE
    ... END CASE
END TRY
```

**Server API functions - version 1.3 only**

Server API functions can create a Web Services server in Genero BDL.

**Note:** These functions are valid for backward compatibility, but they are not the preferred way to handle Genero Web Services. See the com package for the preferred classes and methods for handling Web services.
Table 748: APIs to create a Web Services server in Genero BDL (version 1.3 only)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>fgl_ws_server_setNamespace</strong> (namespace VARCHAR)</td>
<td>Defines the namespace of the service on the Web and must be called first, before all other functions of the API.</td>
</tr>
<tr>
<td><strong>fgl_ws_server_start</strong> (tcpPort VARCHAR)</td>
<td>Creates and starts the Web services server.</td>
</tr>
<tr>
<td><strong>fgl_ws_server_publishFunction</strong> (operationName VARCHAR, inputNamespace VARCHAR, inputRecordName VARCHAR, outputNamespace VARCHAR, outputRecord VARCHAR, functionName VARCHAR)</td>
<td>Publishes the given BDL function as a Web-Function on the Web.</td>
</tr>
<tr>
<td><strong>fgl_ws_server_generateWSDL</strong> (serviceName VARCHAR, serviceLocation VARCHAR, fileName VARCHAR)</td>
<td>Generates the WSDL file based on the BDL-server program.</td>
</tr>
<tr>
<td><strong>fgl_ws_server_process</strong> (timeout INTEGER)</td>
<td>Waits for an incoming SOAP request for a given time (in seconds) and then processes the request, or returns, if there has been no request during the given time.</td>
</tr>
<tr>
<td><strong>fgl_ws_server_setFault</strong> (faultMessage VARCHAR)</td>
<td>Return a SOAP fault string to the client at the end of the function's execution.</td>
</tr>
<tr>
<td><strong>fgl_ws_server_getFault</strong> ()</td>
<td>Retrieves the last fault string the user has set in a Web-Function, or an empty string if there is none.</td>
</tr>
</tbody>
</table>

**fgl_ws_server_setNamespace() (version 1.3)**
Defines the namespace of the service on the Web and must be called first, before all other functions of the API.

**Note:** This function is valid for backward compatibility, but is not a preferred way to handle Genero Web Services. See the com package for the preferred classes and methods for handling Web services.

**Syntax**

```
fgl_ws_server_setNamespace ( namespace VARCHAR )
```

1. `namespace` is the name of the namespace.

**Example**

```
CALL fgl_ws_server_setNamespace("http://tempuri.org/")
```
fgl_ws_server_start() (version 1.3)
Creates and starts the Web services server.

Note: This function is valid for backward compatibility, but is not a preferred way to handle Genero Web Services. See the com package for the preferred classes and methods for handling Web services.

Syntax

```c
fgl_ws_server_start(
  tcpPort VARCHAR
)
```

1. `tcpPort` is a string representing either:
   - the **socket port number** (for a single Web Service server)
   - the **host** and **port** value separated by a colon (for a Web Service server connecting to an application server).
     The value of **port** is an offset beginning at 6400.

Note: If the FGLAPPSERVER environment variable is set, the tcpPort value is ignored, and replaced by the value of FGLAPPSERVER.

Usage

For development or testing purposes, you may start a Web Service server as a single server where only one request at a time will be able to be processed. For deployment, you may start a Web Service server with an application server able to handle several connections at one time using a load-balancing algorithm. The value of the parameter passed to the function determines which method is used.

Examples:

To start a standalone Web Service server:

```c
CALL fgl_ws_server_start("8080") # A single Server is listening
# on port number: 8080
```

To start a Web Service server attempting to connect to an application server:

```c
CALL fgl_ws_server_start("zeus:5") # The server attempt to connect
# to an application server located
# on host zeus and listening
# on the port number 6405
```

Possible runtime errors

- -15504: PORT_ALREADY_USED
- -15514: PORT_NOT_NUMERIC
- -15515: NO_AS_FOUND
- -15516: LICENSE_ERROR

fgl_ws_server_publishFunction() (version 1.3)
Publishes the given BDL function as a Web-Function on the Web.

Note: This function is valid for backward compatibility, but is not a preferred way to handle Genero Web Services. See the com package for the preferred classes and methods for handling Web services.

Syntax

```c
fgl_ws_server_publishFunction(
  operationName VARCHAR,
  inputNamespace VARCHAR,
  inputRecordName VARCHAR,
  outputNamespace VARCHAR,
  outputRecordName VARCHAR,
  description VARCHAR
)
```
1. operationName is the name by which the operation will be defined on the Web. The name is case sensitive.
2. inputNamespace is the namespace of the incoming operation message.
3. inputRecordName is the name of the BDL record representing the Web Function input message or "" if there is none.
4. outputNamespace is the namespace of the outgoing operation message.
5. outputRecord is the name of the BDL record representing the Web Function output message or "" if there is none.
6. functionName is the name of the BDL function that is executed when the Web Service engine receives a request with the operation name defined above.

Example

CALL fgl_ws_server_publishFunction(
    "MyWebOperation",
    "http://www.tempuri.org/webservices/", "myfunction_input",
    "http://www.tempuri.org/webservices/", "myfunction_output",
    "my_bdl_function")

Possible runtime errors

• -15503: FUNCTION_ALREADY_EXISTS
• -15501: FUNCTION_ERROR
• -15502: FUNCTION_DECLARATION_ERROR
• -15512: INPUT_VARIABLE_ERROR
• -15513: OUTPUT_VARIABLE_ERROR
• -15503: BDL_XML_ERROR
• -15518: INPUT_NAMESPACE_MISSING
• -15519: OUTPUT_NAMESPACE_MISSING

fgl_ws_server_generateWSDL() (version 1.3)
Generates the WSDL file based on the BDL-server program.

Note: This function is valid for backward compatibility, but is not a preferred way to handle Genero Web Services. See the com package for the preferred classes and methods for handling Web services.

Syntax

```javascript
fgl_ws_server_generateWSDL(
    serviceName VARCHAR,
    serviceLocation VARCHAR,
    fileName VARCHAR )
RETURNS INTEGER
```

1. serviceName is the name of the web service.
2. serviceLocation is the URL of the server.
3. fileName is the name of the file that will be generated.

Usage

The function returns:

• 0 if the file has been correctly generated.
• Any other values if the operation has failed.
Example

```plaintext
DEFINE mystatus INTEGER

LET mystatus=fgl_ws_server_generateWSDL(
"CustomerService",
"http://localhost:8080",
"C:/mydirectory/myfile.wsdl")

IF mystatus=0 THEN
  DISPLAY "Generation of WSDL done..."
ELSE
  DISPLAY "Generation of WSDL failed!"
END IF
```

fgl_ws_server_process() (version 1.3)

Waits for an incoming SOAP request for a given time (in seconds) and then processes the request, or returns, if there has been no request during the given time.

If a DEFER INTERRUPT or DEFER QUIT instruction has been defined, the function returns even if it is an infinite wait.

Note: This function is valid for backward compatibility, but is not a preferred way to handle Genero Web Services. See the com package for the preferred classes and methods for handling Web services.

Syntax

```plaintext
fgl_ws_server_process (timeout INTEGER)
RETURNS INTEGER
```

1. `timeout` is the maximum waiting time for an incoming request (or -1 for an infinite wait)

Usage

The function can return one of the following values:

- 0 Request has been processed
- -1 Timeout has been reached
- -2 The application server asks the runner to shutdown
- -3 A client connection has been unexpectedly broken
- -4 An interruption has been raised
- -5 The HTTP header of the request was incorrect
- -6 The SOAP envelope was malformed
- -7 The XML document was malformed

Example

```plaintext
MAIN

DEFINE mystatus INTEGER

DEFER INTERRUPT

LET mystatus=fgl_ws_server_process(5)# wait for 5 seconds

IF mystatus=0 THEN
  DISPLAY "Request processed."
END IF
IF mystatus=-1 THEN
```
DISPLAY "No request."
END IF
IF mystatus=-2 THEN  # terminate the application properly
  EXIT PROGRAM  # if connected to application server
END IF
IF mystatus=-3 THEN
  DISPLAY "Client connection unexpectedly broken."
END IF
IF mystatus=-4 THEN
  DISPLAY "Server process has been interrupted."
END IF
IF mystatus=-5 THEN
  DISPLAY "Malformed or bad HTTP request received."
END IF
IF int_flag<>0 THEN
  LET int_flag=0
  EXIT PROGRAM
END IF

END MAIN

**fgl_ws_server_setFault() (version 1.3)**
Return a SOAP fault string to the client at the end of the function's execution.

This function can be called in a published Web-Function.

**Note:** This function is valid for backward compatibility, but is not a preferred way to handle Genero Web Services. See the com package for the preferred classes and methods for handling Web services.

**Syntax**

```plaintext
fgl_ws_server_setFault(
  faultMessage VARCHAR )
```

1. *faultMessage* is a string containing the SOAP Fault string that will be returned to the client.

**Example**

```plaintext
CALL fgl_ws_server_setFault(
  "The server is not able to manage this request."
)
```

**fgl_ws_server_getFault() (version 1.3)**
Retrieves the last fault string the user has set in a Web-Function, or an empty string if there is none.

**Note:** This function is valid for backward compatibility, but is not a preferred way to handle Genero Web Services. See the com package for the preferred classes and methods for handling Web services.

**Syntax**

```plaintext
fgl_ws_server_getFault ()
RETURNS STRING
```

**Usage**
The function returns a string containing the SOAP fault string.

This function is only for testing the Web Services functions before they are published on the Web.

**Example**

```plaintext
DEFINE div_input RECORD
```
a INTEGER,
b INTEGER
END RECORD

DEFINE div_output RECORD
     result INTEGER
END RECORD

FUNCTION TestServices()
    DEFINE string VARCHAR(100)
    ...
    # Test divide by zero operation
    LET div_input.a=15
    LET div_input.b=0
    CALL service_operation_div()
    LET string=fgl_ws_server_getFault()
    DISPLAY "Operation div error: ", string
    ...
END FUNCTION

FUNCTION service_operation_div()
    ...
    IF div_input.b = 0 THEN
        CALL fgl_ws_server_setFault("Divide by zero")
        RETURN
    END IF
    ...
END FUNCTION

Configuration API functions - version 1.3 only

Configuration API functions can modify the behavior of the Web Services engine for the client and/or server.

Note: These functions are valid for backward compatibility, but they are not the preferred way to handle Genero Web Services. See the com package for the preferred classes and methods for handling Web services.

Table 749: Configuration API functions for Web Services engine behavior modification

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fgl_ws_setOption(</td>
<td>Sets an option flag with a given value, changing the global behavior of the Web Services engine.</td>
</tr>
<tr>
<td>optionName VARCHAR,</td>
<td></td>
</tr>
<tr>
<td>optionValue INTEGER)</td>
<td></td>
</tr>
<tr>
<td>fgl_ws_getOption(</td>
<td>Returns the value of an option flag.</td>
</tr>
<tr>
<td>optionName VARCHAR)</td>
<td></td>
</tr>
<tr>
<td>RETURNS INTEGER</td>
<td></td>
</tr>
</tbody>
</table>

fgl_ws_setOption()
Sets an option flag with a given value, changing the global behavior of the Web Services engine.

Note: This function is valid for backward compatibility, but is not a preferred way to handle Genero Web Services. See the com package for the preferred classes and methods for handling Web services.

Syntax

```plaintext
fgl_ws_setOption(
   optionName VARCHAR,
   optionValue INTEGER)
```
1. *optionName* is one of the global **option flags**.
2. *optionValue* is the value of the flag.

**Example**

```
CALL fgl_ws_setOption("http_invoketimeout",5)
```

**Possible runtime errors**

- -15511: INVALID_OPTION_NAME

**fgl_ws_getOption()**

Returns the value of an option flag.

**Note:** This function is valid for backward compatibility, but is not a preferred way to handle Genero Web Services. See the **com package** for the preferred classes and methods for handling Web services.

**Syntax**

```
fgl_ws_getOption (  
    optionName VARCHAR)  
RETURNS INTEGER
```

1. *optionName* is one of the global **option flags**.

**Example**

```
DEFINE value INTEGER
LET value=fgl_ws_getOption("http_invoketimeout")
```

**Possible runtime errors**

- -15511: INVALID_OPTION_NAME

**Option flags**

**Table 750: Option flags**

<table>
<thead>
<tr>
<th>Flag</th>
<th>Client or Server</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>http_invoketimeout</td>
<td>Client</td>
<td>Defines the maximum time in seconds a client has to wait before the client connection raises an error because the server is not responding.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A value of -1 means that it has to wait until the server responds.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The default value is -1.</td>
</tr>
<tr>
<td>tcp_connectiontimeout</td>
<td>Client</td>
<td>Defines the maximum time in seconds a client has to wait for the establishment of a TCP connection with a server.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A value of -1 means infinite wait.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The default value is 30 seconds except for Windows®, where it is 5 seconds.</td>
</tr>
<tr>
<td>Flag</td>
<td>Client or Server</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>soap_ignoretimezone</td>
<td>Both</td>
<td>Defines if, during the marshalling and unmarshalling process of a BDL DATETIME data type, the SOAP engine should ignore the time zone information. A value of zero means false. The default value is false.</td>
</tr>
<tr>
<td>soap_usetypedefinition</td>
<td>Both</td>
<td>Defines if the Web Services engine must specify the type of data in all SOAP requests. This will add an &quot;xsi:type&quot; attribute to each parameter of the request. A value of zero means false. The default value is false.</td>
</tr>
<tr>
<td>wsdl_decimalsize</td>
<td>Server</td>
<td>Defines if, during the WSDL generation, the precision and scale of a DECIMAL variable will be taken into account. See WSDL generation option notes on page 3563. A value of zero means false. The default value is true.</td>
</tr>
<tr>
<td>wsdl_arraysize</td>
<td>Server</td>
<td>Defines if, during the WSDL generation, the size of a BDL array will be taken into account. See WSDL generation option notes on page 3563. A value of zero means false. The default value is true.</td>
</tr>
<tr>
<td>wsdl_stringsize</td>
<td>Server</td>
<td>Defines if, during the WSDL generation, the size of a CHAR or VARCHAR variable will be taken into account. See WSDL generation option notes on page 3563. A value of zero means false. The default value is true.</td>
</tr>
</tbody>
</table>

**WSDL generation option notes**

1. For a BDL type DECIMAL (5, 2), when `wsdl_decimalsize` is TRUE, the generated WSDL file contains the total size and the size of the fractional part of the decimal:

   ```xml
   <types>
   <simpleType name="echoDecimal5_2_a_dec5_2_out_FGLDecimal">
   <restriction base="decimal">
   <totalDigits value="5" />
   ```
When `wsdl_decimalsize` is FALSE, the total size and the size of the fractional part are not mentioned:

```
<message name="echoDecimal5_2">
  <part name="dec5_2" type="xsd:decimal" />
</message>
```

2. If the WSDL file does not contain the size, the client application has no way of knowing the size. In this scenario, a default value for the size is generated. For example, the exported server type `DECIMAL(5,2)` becomes a `DECIMAL(32)` on the client side.

3. It is better to keep the options `wsdl_arraysize`, `wsdl_stringsize` and `wsdl_decimalsize` set to TRUE (default) so that the BDL client application can do an exact type mapping.

### Using fglwsdl to generate code from WSDL or XSD schemas

The `fglwsdl` tool generates Genero (.4gl) code from WSDL / XSD schemas.

See `fglwsdl` on page 2081 for a complete description of the tool and its options.

#### Generate TYPE definitions from global XML elements or attributes

If a WSDL or a XSD has global XML elements or attributes defined with an inlined type, the `-fInlineTypes` option of `fglwsdl` generates a TYPE definition representing that inline type, using the original WSDL/XSD name of the element or attribute, concatenated with the string `'GlobalAttributeType'` or `'GlobalElementType'`.

For example, when using `fglwsdl -fInlineTypes`, the following schema:

```
<xs:element name="getAlertListRequestFlow">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="getAlertListRequest" type="amp:getAlertListRequest" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

will produce:

```
TYPE tgetAlertListRequestFlowGlobalElementType RECORD
  ATTRIBUTES (XMLSequence)
    getAlertListRequest tgetAlertListRequest
      ATTRIBUTES (XMLName="getAlertListRequest")
  END RECORD
DEFINE getAlertListRequestFlow tgetAlertListRequestFlowGlobalElementType
  ATTRIBUTES (XMLName="getAlertListRequestFlow")
```

Instead of:

```
DEFINE getAlertListRequestFlow RECORD
  ATTRIBUTES (XMLName="getAlertListRequestFlow",XMLSequence)
    getAlertListRequest tgetAlertListRequest
      ATTRIBUTES (XMLName="getAlertListRequest")
  END RECORD
```
These topics cover programming subjects about mobile applications

**Types of Genero Mobile apps**

Genero supports different types of mobile app architectures: development mode, standalone apps, partially-connected apps, and client-server apps.

When you are developing your app, and you execute the app on your development machine for display on a device or emulator, you are running the app in **development mode**.

When you follow the procedure to deploy your application to the device for testing or to distribute your app to your end users, you have a **deployed app**. A deployed app might be an app that executes irrespective of network availability, it might be an app that accesses device peripherals, it might be an app that requires network access.

**Note:** The DVM refers to the dynamic virtual machine, which is the process that executes the app code.

**Development mode**

In **development mode**, the app process is running on a server, and the forms are displayed on a mobile front-end.

This architecture is typically used when developing your application, as you can easily modify, recompile and execute your sources on the development machine, without the need to deploy the app on the device.

![Development mode diagram](image)

**Figure 143: Development mode**

**Standalone apps**

A **standalone app** has the DVM and display client entirely on the mobile device. This app executes irrespective of network availability and can access device peripherals such as the camera, contacts, email, calendar, GPS, and storage via exposed APIs (front calls). For database needs, this app can only connect to a local SQLite database.
Figure 144: Standalone app

Partially-connected apps

A partially-connected app has the DVM and display client entirely on the mobile device, yet this app includes items that require a network connection. This app must be able to run when no network connection is available. This app uses a network API to talk to any back-end.

Examples include:

- Web Services performed with JSON over HTTP; use RESTful methods to write data synchronization routines. With this example, business logic executes within the device’s Virtual Machine and the user is able to store captured data to a local SQLite database. When the network becomes available, the user synchronizes the stored data with the remote server’s database.
  
  **Note:** As of Genero Mobile 1.1, you can also write a Web Service using SOAP.

- A web component that runs Google Maps.

This app first operates without a network connection, and must be able to run without a network connection. Once network connectivity is restored, the app can perform network-dependent tasks such as synchronizing with a remote database, make a web service call, or use a web component.

If you are using GMI, and the device goes into standby mode, the application does not run in the background and activities with the network are suspended.
Client-server apps

With a client-server app or connected app, the bulk of the app runs on a remote server and the display client sits on the mobile device.

As with any deployed app, this app first starts on the mobile device; the DVM for the deployed app runs on the mobile device. The role of the deployed app, however, is to connect to a remote corporate server as an online terminal. It is the deployed app that launches the remote application using the runOnServer frontcall. The remote application's DVM and business logic reside on the remote server, somewhere in the network. The remote application is not limited to a SQLite database.

In the event that the network is interrupted, the Genero Mobile client app is suspended until service resumes.
Licensing considerations

Note: For specific mobile app license details, see your Software License Agreement, or contact your nearest Four Js sales office.

When deploying a Genero Mobile app, the type of mobile app architecture you need may have additional requirements for Genero licenses as follows:

- Development mode needs a development license.
- Partially-connected app consumes a Genero runtime license when the app connects to the Web service to synchronize with the network database.
- Client-server app consumes a Genero runtime license on the Genero Application Server.

Mobile development mode

Set up a development environment to display app forms on a mobile front-end.

In a command-line development environment, the runtime system (DVM) executes the application code on the development machine, displaying app forms through a mobile front-end installed on a device or emulator.

Once the app development is complete, you can build the real mobile app to be deployed on devices, as described in Deploying mobile apps on page 3588.

Important: When the Genero Mobile Development Client is running on a device, a Genero BDL program can connect and execute front calls. To avoid any security issues, stop the Genero Mobile Development Client app when you have completed your work, and consider uninstalling the app before giving the mobile device to another person or department.

Genero mobile development client for Android™

Set up a development environment to display app forms on an Android™ device.

Using the GMA front-end on Android™ devices

To display Genero application forms on an Android™ device in development mode (with programs executing on a computer), the GMA front-end app must be installed on the device (or emulator).

Note: With the GMA front-end installed on the mobile device, you can perform a classic GUI connection based on FGLSERVER.

Installing the GMA front-end application

Before installing the GMA front-end, fulfill the prerequisites to build an Android™ app, as described in Building Android apps with Genero on page 3591.

Perform the following steps, to install the GMA front-end from the archive:

1. Get the GMA package (for example, fjs-gma-1.30.00-build201703031626-allos.zip), contact your support channel for download details.
2. Unzip the archive, containing an APK ready to install.
3. Plug your device via USB cable to the computer.
4. Install the GMA front-end APK with the gmabuildtool command with the test option.

For example:

```
$ . ~/genero/devel/fgl/mobile/java-1.8.env
$ . ~/genero/devel/fgl/mobile/android-sdk.env
$ mkdir /tmp/gma
$ cd /tmp/gma
$ unzip ~/Download/fjs-gma-1.30.00-build201703031626-allos.zip
```

Archive: ~/Download/fjs-gma-1.30.00-build201703031626-allos.zip
Configure FGLSERVER and run the app

Once the GMA front-end is installed on the device, make sure that WIFI is enabled (or check that the TCP port is forwarded when using an emulator), you can start the GMA app.

The main GMA screen shows the IP address of the device and the TCP port it is listening to (0=6400).

On the development machine, define the FGLSERVER environment variable with the IP address of the device (or.

Now you are ready to run your app on the server and display on the Android™ device.

TCP port forwarding for GMA on Android emulator

When using an Android™ emulator, you can forward the TCP port with the adb forward command:

```
adb forward tcp:local_port tcp:gma_device_port
```

For example:

```
adb forward tcp:6401 tcp:6400
```

You can then connect to GMA with:

```
FGLSERVER=127.0.0.1:1
```

Enabling Universal Rendering

Universal Rendering can be enabled by using the gui.rendering FGLPROFILE entry.

Related concepts

Deploying mobile apps on Android devices on page 3589

This section contains information to create a mobile application to be deployed on Android™ devices.

Genero mobile development client for iOS

Set up a development environment to display app forms on an iOS device.

Installing the GMI front-end on iOS devices

To display Genero application forms on a device, the GMI front-end must be installed on the iOS device (or emulator).

Genero supports two types of GMI front-end apps for iOS:

- The ready-to-use "Genero Mobile Development Client" for iOS, available through the App Store.
  
  **Important:** Due to Apple® limitations, the Genero Mobile Development Client app is not allowed to listen to a TCP port to provide a GUI service. In order to establish the GUI front-end connection, the front-end must connect to the runtime system running on the development machine.

- A self-made GMI front-end, using the gmibuildtool to build a GMI front-end app with your own Apple® developer account.

  **Note:** With this configuration, you can perform a classic GUI connection based on FGLSERVER.
Using the Genero Mobile Development Client for iOS

Go to the App Store. Search for "Genero Mobile". Select and install the "Genero Mobile Development Client".

Once the GMI development client is installed on the device, make sure that WIFI is enabled and start the GMI app.

Note: Because of iOS app limitations defined by Apple, an app shipped on the App Store cannot listen to a TCP port to provide a GUI service. In order to display Genero form on the GMI development client, you will have to establish the GUI connection from the device to the server, after starting the fglrun process with the --gui-listen option.

The main GMI screen shows a URL field to let you enter the IP address / hostname and the TCP port the runtime system will listen to.

Note: Make sure that the firewall on the development machine allows incoming connections for the TCP port number specified with the --gui-listen fglrun option.

Start the application on the development server, by using the fglrun --gui-listen=portnum command:

```
fglrn --gui-listen=6500 main.42m
```

On the iOS device, enter the following in the URL field:

```
fgl://dev-server-hostname:6500
```

Then tap the [Connect] button to establish the GUI connection.

Build your own GMI front-end

In order to use the classic GUI connection mode with fglrun connecting to the mobile front-end via FGLSERVER, it is possible to create your own GMI front-end, with your own Apple® certificate and provisioning profile.

Note: As with other iOS apps, a self-made GMI front-end can only be created on a macOS® computer.

The generated GMI can then be deployed on your device or simulator. The GMI front-end will listen on the port 6400, to display applications running on a server through the FGLSERVER setting.

Before creating your own GMI front-end, fulfill the prerequisites to build an iOS app as described in Building iOS apps with Genero on page 3606.

In order to build your own GMI front-end:

1. Make sure that the installed the fjs-fglgmi*.zip archive into FGLDIR.
2. Go to the $FGLDIR/demo/MobileDemo/gmiclient directory.
3. Delete the complete build directory if it exists (can be done with a make clean command).
4. Compile the GMI front-end program files (main.42m, main.42f, etc): See Makefile for details.
5. Build the GMI front-end:
   - To build and install the GMI front-end on the simulator, first make sure that the simulator is started (open -a Simulator command), then execute the make command with the gmi.install rule:
     ```
     $ open -a Simulator
     $ make gmi.install
     ```
   - To build and install the GMI front-end on the device plugged to your Mac, get a development certificate and provisioning profile, then execute the make command with the gmi.install rule, by specifying the TARGET, the IDENTITY and the PROVISIONING_PROFILE variables:
     ```
     $ make TARGET=phone IDENTITY=WKRRJZ999 \
     PROVISIONING_PROFILE="/Library/MobileDevice/Provisioning Profiles/myapp.mobileprovision" \
     gmi.install
     ```
Once the GMI front-end is installed on the device, make sure that WIFI is enabled and start the GMI app. The main GMI screen shows the IP address of the device and the TCP port it is listening to (0=6400). On the development machine, define the FGLSERVER environment variable with the IP address of the device. Now you are ready to run your app on the server and display on the iOS device.

**Enabling Universal Rendering**

Universal Rendering can be enabled by using the `gui.rendering FGLPROFILE entry`.

**Related concepts**

Deploying mobile apps on iOS devices on page 3604

This section contains information to create a mobile application to be deployed on iOS devices.

### Language limitations

Genero language features not supported on mobile devices.

**Important:** This topic is provided as a quick glance at Genero Business Development Language limitations in mobile applications. Details can be found in the BDL reference topics.

**Platform specifics**

**Features with limited support**

The following language options have limited support:

- The `RUN` instruction has limited support on mobile platforms.
  - The `RUN` instruction is not supported on mobile devices, because of operating system limitations.
  - `RUN command WITHOUT WAITING` is not supported when programs run on an application server and display on a mobile device, because the Genero GUI protocol is not able to handle multiple connections at the same time.

**Unsupported features**

The following language features are not supported:

- The `INPUT ARRAY` instruction is not supported.
- The `base.Channel.openPipe` method is not supported.
- The Java interface cannot be used in apps running on iOS devices: There is no standard free JVM available.

### Environment variables

You may need to set environment variables for your app.

**Automatic environment variables**

When executing on a mobile device:

- `DBDATE` defaults to the regional settings defined on the device.
- `FGLAPPPDIR` is automatically defined to the application directory (`appdir`).

**Set environment variables**

Setting environment variables for your app must be done in an `fglprofile` file. This `fglprofile` file must be located in the `appdir` directory, beside the main program module.
To add an environment variable for your mobile app, use the following syntax:

```
mobile.environment.DBFORMAT="$:,::
```

Any existing environment variable setting is overwritten by the value set (using `mobile.environment.envvarname`) in the `fglprofile` file.

For more details, see Setting environment variables in FGLPROFILE (mobile) on page 263.

**Note:** Environment variables set in an FGLPROFILE file are only read when the deployed application runs on the mobile device. They are not read during development mode (that is when the VM runs on the development machine and the mobile client displays on the device). The FGLPROFILE environment variable settings are only for the VM component and are ignored by the GMA/GMI front-end component.

---

**App localization**

Mobile apps can be designed to display localized texts based on the language selected on the device.

**Application locale definition**

Make sure that your application locale is defined properly for development and deployment environments.

**Note:** With Genero mobile applications, the application locale must be UTF-8.

For more details, see Quickstart guide for locale settings on page 515.

**Using localized strings**

Localized string files (.42s) must be deployed in directories matching the language identifiers (`en` for English, `zh_TW` for simplified Chinese, etc), beside the program module.

Default string files can be provided in `APPDIR/defaults` as fallback directory, if the current mobile language does not match one of the language-specific directories provided by the application.

The list of .42s files required by the application must be defined in the unique `fglprofile` configuration file located beside the program module of your application.

For more details, see Localized string files on mobile devices on page 546 and Deploying mobile apps on page 3588.

**Querying user's preferred language**

The application can get the user's preferred language and regional settings as configured on the mobile device, by using the `standard.feInfo` front call with the `userPreferredLang` argument.

For more details, see User's preferred language on page 533 and `feInfo` on page 2651.

---

**App execution**

Mobile apps are started and stopped, and can switch between foreground and background states when running.

**Mobile app execution modes**

Mobile apps can be fully embeeded as a standalone or can execute on a server and display forms on the mobile device.

On iOS and on Android™, standalone Genero apps are installed and started on the device as other apps. In this configuration, the application program files, the runtime system and the database are present on the mobile device.
When a permanent network connection is available (WIFI), it is also possible to have a lightweight app installed on the device, that executes the real application code on a server, displaying forms on the device. With this architecture, your mobile app benefits from the processing power of the server, and can access a large database.

For more details, see Starting programs on a mobile device on page 489.

**Background/foreground modes**

Describes how to handle background or foreground modes in mobile apps.

**Mobile apps foreground and background modes**

Mobile apps can change their state from background mode to foreground mode and vice-verse. For example, when the user switches to another app, or when going back to the home screen, the current app goes to background mode. An app goes to foreground mode, when it is re-selected from the active apps list.

**Detecting foreground/background mode changes**

Genero BDL provides two predefined actions, to detect when the state of a mobile app changes:

1. The action *enterbackground* is fired, when the app goes to background mode.
2. The action *enterforeground* is fired, when the app goes to foreground mode.

**Note:** The *enterforeground* action is not fired when the app starts. This action is only fired when returning to foreground mode, after it was in background mode.

To execute code when the app goes to background or foreground mode, use on **ON ACTION** handler:

```plaintext
ON ACTION enterbackground  
  LET skip_timers = TRUE  
ON ACTION enterforeground  
  IF NOT ask_password() THEN  
    EXIT PROGRAM  
  ENF IF
```

For example, when the app enters background mode, it is recommended that the program suspend any activity, and skip code that would be executed in **ON TIMER** triggers.

On the other hand, when the app enters foreground mode, the program can for example ask the user's login/password again, for security reasons, to make sure that the mobile device did not end up in other hands while the app was in background mode.

To control action view rendering defaults and current field validation behavior when the *enterforeground/enterbackground* actions are used, consider setting action default attributes for these action in your .4ad file as follows:

```plaintext
<ActionDefaultList>  
  ...  
  <ActionDefault name="enterbackground" validate="no" defaultView="no" contextMenu="no"/>  
  <ActionDefault name="enterforeground" validate="no" defaultView="no" contextMenu="no"/>  
  ...  
</ActionDefaultList>
```

Another option is to define these action defaults attributes in the **ON ACTION** handlers:

```plaintext
ON ACTION enterbackground (VALIDATE=NO, DEFAULTVIEW=NO)  
  ...  
ON ACTION enterforeground (VALIDATE=NO, DEFAULTVIEW=NO)  
  ...
```
Checking if the app is currently in foreground mode

To know the current state of a mobile app, use the `mobile.isForeground` front call:

```groovy
define fg boolean
    call ui.interface.frontcall("mobile", "isForeground", [], [fg])
    if fg then
        ...
    end if
```

What does the app when in background mode?

On iOS, when the app is in background mode, the program cannot do anything meaningful outside push notifications or audio play.

On Android™, when the app is in background mode, the program can continue its execution.

Avoid Android™ to terminate app when in background (GMA)

On Android™ devices, an app can switch between foreground to background states.

The Android™ system can decide to stop an app in background state, for example when resources are required for other apps.

Genero programs running on servers are typically not prepared to be stopped at any time; except in case of major failure, it’s the program that decides when it terminates. On mobile devices, Android™ can decide to stop the app when it is in background state.

By default, when the app goes to background state, a notification is shown by GMA, to keep the app in foreground state, and avoid Android™ stopping the app. The notification disappears, when the app returns to foreground state.

Use the `androidKeepForeground` style attribute to control the way the GMA forces Android™ to keep your app alive. Set this attribute to "no" if your app can be stopped by Android™, when it is in background state. When this style attribute is set to "no", GMA will not display a notification, when the app switches to background mode.

**Important:** When using `androidKeepForeground=no`, Android™ may stop the app at any time. Make sure that the code is ready for this case.

For more details, see User Interface style attributes on page 1219.

App user interface

This section includes topics about user interface programming for mobile.

In general, the user interface of a mobile app written in Genero displays and reacts as a desktop or web application, while simultaneously respecting the device operating system look-and-feel. There are parts of the interface, however, that display and react in a specific way.

Take a look at each of the user interface items in this chapter, to understand how they are portrayed in a mobile app. A user interface feature not listed means there is nothing mobile-specific to its display or behavior.

**Note:** The topics of this section relate to the native rendering mode. When using the Universal Rendering mode, GBC web browser rendering rules apply.
Images and icons

For this topic, an image can refer to the IMAGE item type or the icons used in the app.

Image format support

Mobile apps written in Genero supports all image formats supported by the device OS. However, each platform has its own restrictions on which image formats it supports:

- Android image format support
- iOS image format support

Mobile devices have a much higher pixel density (a higher resolution) than classic desktop monitors. An image which looks nice on a desktop can appear small or as an upscaled image on a mobile device.

Providing the image resource

Genero supports different solutions to provide the image data in a mobile app, depending on the need (button icon, application picture, etc). To understand how to get image resource on mobile apps with Genero, see Providing the image resource on page 1149.

Image sizing on mobile devices

The IMAGE item type defines an area for the display of an image on a form.

Image layout and sizing can be controlled with form item attributes to adapt to the type of mobile device.

For more details, see Controlling the image layout on page 1148.

Default action icons

In general, you want the icons used for your mobile app to be the standard icons used by all apps for the mobile platform. Genero is set up to use such icons by default. For more details, see Action views on mobile devices on page 1777.

Genero also supports icon centralization based on TTF icons, to get a global consistent look and feel for all your mobile apps. For more details, see Providing the image resource on page 1149.

Action rendering

Actions are rendered with default action views following mobile platforms standards.

The top and/or bottom parts of the mobile app screen is dedicated to displaying default action views to the user.

A default action view is an implicit graphical item that can be tapped to fire the corresponding action.

The default action views are rendered on the mobile device using platform-specific standards, which are covered in Action views on mobile devices on page 1777.
Table 751: Mobile platform differences for toolbars

<table>
<thead>
<tr>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
</table>

![Figure 147: Action rendering example on an Android™ device](image1)

![Figure 148: Action rendering example on an iOS device](image2)

**Related concepts**

- [Action views on mobile devices](#) on page 1777
  Action views are rendered following mobile specific standards.

**Toolbar rendering**

Toolbars allow control over where actions display (and in what order).

For desktop applications, the toolbar is a series of buttons typically contained in a toolbar object, located at the top of the form.

On Genero mobile apps, toolbars are rendered to the mobile platform standards.
Table 752: Mobile platform differences for toolbars

<table>
<thead>
<tr>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Genero toolbar items appear in the Android™ <strong>action bar</strong>.</td>
<td>Genero toolbar items appear in the iOS <strong>toolbar pane</strong>.</td>
</tr>
</tbody>
</table>

**Related concepts**

- **Toolbars on mobile devices** on page 1439
  Toolbars can be used to control action view rendering on mobile devices.

- **Action views on mobile devices** on page 1777
  Action views are rendered following mobile specific standards.

**Topmenu rendering**

Topmenus provide a hierarchical menu in the app.

A topmenu will be displayed on the device by using the platform GUI standard specifications:
Table 753: Genero Mobile and topmenus

<table>
<thead>
<tr>
<th>GMA</th>
<th>GMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Genero topmenus appear as Android™ menu icon.</td>
<td>Genero topmenus appear as iOS menu icon.</td>
</tr>
</tbody>
</table>

A navigation drawer is visible in the action bar if a topmenu is available to the displayed form. The navigation drawer is a panel to the left of the app icon (seen as three bars).

A menu icon is visible in the navigation bar if a topmenu is available to the displayed form.

Related concepts
Topmenus on mobile devices on page 1449
Topmenus can be used to implement a general options menu in mobile apps.
Action views on mobile devices on page 1777
Action views are rendered following mobile specific standards.

**Keyboard type**

Depending on the data being entered, a mobile device is expected to display the keyboard that is appropriate for the data.

There are a variety of keyboard types for mobile devices. A field dedicated to phone number input is expected to display a keyboard specifically for phone number input.

The `KEYBOARDHINT` form field attribute provides a hint regarding the kind of data the form field contains:

```
ATTRIBUTES
EDIT f01 = customer.cust_phone, KEYBOARDHINT = PHONE;
...
```

Although Genero mostly respects the provided hint, the variable data type that is bound to the form field is also examined, to determine what keyboard to display:

- If the field is defined as a `DATE` or `DATETIME` field, the date picker displays regardless of the `KEYBOARDHINT` setting.
- If the field is a character string data, a text keyboard displays if `KEYBOARDHINT=NUMBER` for example.

For more details, see `KEYBOARDHINT attribute` on page 1378.

**List views**

Form tables in a mobile app render as list views.

List views are commonly used in mobile apps to present an indexed list of items or selectable list of options. They are also used to let users navigate through hierarchically structured data.

List views are displayed as either full list views or embedded list views.

The list view only displays the first two columns’ content and any associated row image, regardless of the number of columns defined.

No column header/title is displayed in mobile list views. Thus the mobile user cannot manipulate columns (hide, reorder, resize, or sort).

With full list views, the built-in reduce filter allows the user to filter the rows displayed.

The `JUSTIFY` attribute of the second column can influence how the rows are displayed.

Various options affect the rendering and behavior, by defining `TABLE` container attributes, `DISPLAY ARRAY` dialog attributes and `ON ACTION` handler attributes.

For complete details on implementing table views in a mobile app, see `Using tables on mobile devices` on page 1867.

**Split views**

Split views refer to the ability to access two forms side by side on a mobile device.

**Side by side views on mobile apps**

Many mobile apps offer a specific form layout, splitting the screen in two in order to show a list of the left side and a detail form on the right side. Such kind of layout can be implemented in Genero with the `Split views` on page 1900.

**Differences in how split views are handled by the clients**

There are differences between the Genero Mobile for iOS (GMI) and Genero Mobile for Android (GMA) implementations of split views and parallel dialogs, to include:

- When the application displays in a single pane or in two panes.
• How a user switches between the two panes.

There are also differences in how the split view renders between GMA and GMI.

**Figure 151: The stores2 demo rendered on an Android device**

With Genero Mobile for Android:

• The navigator pane renders as a menu in the left-hand side of the title bar.
• All buttons are merged.
• The title is not displayed when there is a navigator pane. If there is no navigator pane, the title of the current window is displayed.

With Genero Mobile for iOS:
• The navigator pane renders along the bottom of the app.
• Each window has its own title and its own buttons.

Figure 152: The stores2 demo rendered on an iOS device

Front call support
Genero Mobile provides front calls to interface with the device capabilities.

Use front calls to access mobile devices capabilities. For example, with the mobile/takePhoto front call, you can open the camera app of the device to take a picture.

Note: In a classical Genero client/server configuration, a front call is a remote procedure call that involves a round-trip between the front-end and the server where the application executes. For a standalone mobile app, this does not cause any latency. For a server-side app (using the runOnServer front call), however, latency can result.

The details for each front call can be found in Built-in front calls summary on page 2636.

Color and theming
Mobile applications must follow the platform colors and theming.

User interface design on mobile devices
Genero BDL provides several ways to define colors and styles for a mobile app. This section introduces features that can be used to customize your mobile app and adapt to the target platform user interface design. As a general rule, avoid using non-standard ergonomics and decorations, use defaults to let Genero render your forms depending on the platform standards. For example, the GMA front-end will use Google material design on Android™ devices.

Defining TTF icon colors
By default, TTF icons get the color of the platform theme. A default color can be defined for all TTF icons of a window with the defaultTTFColor style attribute. In order to define a color for a specific icon, add an #RGB color specification in your image to font glyph mapping file.

For more details, see Using a simple image name (centralized icons) on page 1150

App color theme on Android
Android™ apps can be created with a specific color theme following the Google material design.

When building the APK with the gmabuildtool, you can specify the general app colors with the --build-app-colors option.

For more details, see Define app's color theme on page 3597.

iOS specific UI elements
On iOS devices, some UI elements are specific to the platform and are not represented in the Abstract User Interface tree. In order to define the decoration of these elements, specific style attributes have been defined. For more details, see Decorate iOS UI elements on page 3582.

Screen orientation
Detecting screen orientation changes with the mobile device.

Detecting screen orientation changes
Mobile apps can adapt to the device orientation, when switching between portrait and landscape mode.

Screen orientation changes can be detected with the windowresized predefined action:

```
ON ACTION windowresized
```
Controlling screen orientation on GMA / Android

Use the allowedOrientations Window style attribute, to control screen orientation options on Android devices:

```xml
<Style name="Window.main">
  <StyleAttribute name="allowedOrientations" value="portrait_reverse" />
</Style>
```

For the complete list of possible values, see allowedOrientations style attribute.

Controlling screen orientation on GMI / iOS

On iOS devices using GMI, the screen orientation possibilities must be defined in the Info.plist resource file. Keys such as UISupportedInterfaceOrientations must be defined in the file, for example:

```xml
<?xml version="1.0" encoding= "UTF-8"?>
<!DOCTYPE plist PUBLIC  
"-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd" >
<plist version= "1.0">
<dict>
  <key>UISupportedInterfaceOrientations</key>
  <array>
    <string>UIInterfaceOrientationPortrait</string>
    <string>UIInterfaceOrientationPortraitUpsideDown</string>
  </array>
  <key>UISupportedInterfaceOrientations~ipad</key>
  <array>
    <string>UIInterfaceOrientationPortrait</string>
    <string>UIInterfaceOrientationPortraitUpsideDown</string>
  </array>
</dict>
</plist>
```

For more details about Info.plist usage, see the section about defining iOS app properties in Info.plist.

Decorate iOS UI elements

This section describes the presentation style attributes that are supported for iOS devices.

Foreground colors

In order to define the foreground color for items used in the iOS navigation bar, toolbar, and some items in the forms (Buttons, SpinEdit, Radiogroups, and row checkmark and disclosure indicators in list views), the iosTintColor style attribute can be used at Window level:

```xml
<StyleList>
  <Style name="Window">
    <StyleAttribute name="iosTintColor" value="white" />
  </Style>
</StyleList>
```

Background colors

Background color of iOS specific elements can be defined with the following style attributes at the Window level:

- iosTabBarTintColor
• iosToolBarTintColor
• iosNavigationBarTintColor

For example:

```xml
<StyleList>
  <Style name="Window">
    <StyleAttribute name="iosTintColor" value="white" />
    <StyleAttribute name="iosNavigationBarTintColor" value="blue" />
    <StyleAttribute name="iosToolBarTintColor" value="blue" />
  </Style>
</StyleList>
```

**iosTintColor inheritance**

The **iosTintColor** style attribute can be used at Window level to defined the text color for Tabbar and Toolbar elements.

However, when defining the text color at Window level, it has an impact on the form elements such as Folder, Button, SpinEdit and RadioGroup too. For example, if you set **iosTintColor** to white at the window level, Folder, Button, SpinEdit and RadioGroup element will appear as if they were hidden.

To avoid this, set another text color at the Form element level, as shown in the next .4st style file:

```xml
<StyleList>
  <Style name="Window">
    <StyleAttribute name="iosTintColor" value="white" />
    <StyleAttribute name="iosNavigationBarTintColor" value="blue" />
    <StyleAttribute name="iosToolBarTintColor" value="blue" />
  </Style>
  <Style name="Form">
    <StyleAttribute name="iosTintColor" value="blue" />
  </Style>
</StyleList>
```

**Important:** Note that in the above example, the **iosToolBarTintColor** and **iosNavigationBarTintColor** define respectively the background color for the iOS navigation bar and the iOS toolbar.

For a complete description of these attributes, see [Style attributes reference](#) on page 1182.

**Database support on mobile devices**

On the device, a Genero app can use SQL for data management.

**Databases supported on mobile devices**

Only SQLite can be used on mobile devices. SQLite has a small footprint, is free and readily available.

The database driver (dbmsqt) and the SQLite library are built into the runtime system for mobile Genero apps. No database driver specification is required when running on mobile.

To read more about SQLite programming, see [Using SQLite database in mobile apps](#) on page 3584.

**Synchronizing data with a central database**

When local mobile app data needs to be synchronized with a central database, you must write your own synchronization routines using Web Services. You must implement a back-end service to collect mobile database updates and to send central database changes back to the mobile app.
Important: If you are using GMI, and the device goes into standby mode, the application does not run in the background and activities with the network are suspended. If you are synchronizing data with the server, and the device goes into standby, the synchronization is suspended until the device resumes from standby. If you have a long synchronization, you need to either disable the sleep to allow the synchronization to complete, or accept that the synchronization will suspend when the device goes into standby mode.

Related concepts
SQLite on page 1004

Using SQLite database in mobile apps
On the device, Genero Mobile uses the SQLite database only.

Running an app in development mode
When running an app in development mode (where the app runs on a computer), you can use any database server that Genero supports for the operating system of the server-side app.

Running an app on a mobile device
When running the application on the device, only SQLite can be used. The database must be created at the first application execution, or it must be delivered as the default database in the .ipa or .apk package.

Locale character set and length semantics
SQLite stores data in UTF-8 codeset, mobile apps will by default run in UTF-8 and with character length semantics (FGL_LENGTH_SEMANTICS=CHAR).

Creating the database
Mobile applications usually create their database at first execution. The SQLite database file must be created in the application sandbox, in a writable directory. If the database file does not exist in the current working directory (os.Path.pwd()), create an empty file and then perform a CONNECT TO instruction.
For more details, see Creating a database from programs on page 608.

Providing a default database
SQLite database file format is cross-platform. Instead of creating the database the first time the application starts, you might want to prepare a default database file in your development environment, and include it in the .ipa/.apk package.

Data types with SQLite
SQLite does not have strict data type checking as traditional databases. If you define a table column as a DECIMAL, you can still store character values in that column. Pay attention to this SQLite specific feature, to avoid invalid storage and type conversion errors in your application.
Consider using the following data types for maximum portability, especially when data needs to be synchronized with a central database server, where the data types must match to the types used in the mobile application: CHAR, VARCHAR, DATE, DATETIME YEAR TO MINUTE, DATETIME YEAR TO FRACTION(3), DECIMAL, SMALLINT, INTEGER, BIGINT, BYTE, TEXT.

Optimizing data changes
SQLite can be slow at doing commits, due to the data integrity technique used for transactions. Since each INSERT / UPDATE / DELETE statements acts as an individual transaction (i.e., auto-commit), there will be as many transactions/commits as data manipulation statements. For example, it takes about 10 seconds to insert 1000 rows on an Intel core i7 2.60GHz CPU / 5400.0 RPM HDD computer.
When executing code that modifies a lot of rows (for example, when inserting default data at first application execution, or when doing synchronization with a central database), enclose the SQL statements within a `BEGIN WORK / COMMIT WORK` transaction block to speed up the process:

```sql
BEGIN WORK
    FOR i=1 TO mylog.getLength()
        -- INSERT / UPDATE / DELETE statements
    END FOR
COMMIT WORK
```

### Enforcing foreign key constraints

SQLite 3.6.19 and + support foreign key constraints, with `ON DELETE CASCADE` and `ON UPDATE CASCADE` options. By default, however, foreign key constraints are not enforced. Each application must explicitly turn on the feature with a `PRAGMA` command. Immediately after the database connection, you can perform the `PRAGMA` command in an `EXECUTE IMMEDIATE` statement:

```sql
CONNECT TO connstr AS "c1"
EXECUTE IMMEDIATE "PRAGMA foreign_keys = ON"
```

### Truncating the SQLite database file

By default, when deleting rows, SQLite keeps the unused database file pages for future storage. As result, when deleting a large amount of data, the database file might be larger than necessary. Consider truncating the database file with the `VACUUM` SQL command (in an `EXECUTE IMMEDIATE` statement), if disk space is limited and when a lot of database rows were deleted.

Depending on the application, the `VACUUM` command can be executed:

- when starting the application, just after connecting to the database,
- after doing a large database operation (such as a synchronization with a central database),
- as a manual option that the user can trigger.

For example, after connecting to the database:

```sql
CONNECT TO connstr AS "c1"
EXECUTE IMMEDIATE "VACUUM"
```

### Sharing database files between Android apps

Two different Android™ apps (each packaged as a separate `.apk`) execute in their own sandbox, but have access to the storage area (SD-CARD) and therefore can share a common database file.

SQLite handles concurrent access to the same database file by setting a lock on the entire db file when modifying data (`INSERT/UPDATE/DELETE`). By default, if a writer process locks the file, other processes must wait until the lock owner process completes its transaction and releases the lock.

Because of Informix® compatibility, Genero BDL uses a default lock timeout or zero (i.e., not waiting for locks to be released). As result, when writing to a database file that is locked by another process, if the isolation level is `SERIALIZATION` (the default with SQLite), an application will get the SQL error -244.

To avoid this problem, you must change the default lock timeout with the `SET LOCK MODE` instruction, after starting the database session:

```sql
CONNECT TO connstr AS "c1"
SET LOCK MODE TO 5 -- seconds
```

The second process will then wait until the first process releases the lock. If transactions are short (milliseconds), having processes waiting for each other is transparent to the user.
Related concepts
SQL programming on page 590
Covers topics about interacting with a database server using SQL.
SQLite on page 1004

Accessing device functions

Mobile apps can access device functions by using front calls.

Accessing device functions using frontcalls

Mobile applications typically want to access device functions such as geolocation, multi-media content (photos, videos), messaging (contacts database, email, sms).

This can be easily achieved by using front calls dedicated to mobile features. Note that some functions are platform specific, for example to launch an Android activity, or access to iOS device settings.

As a general rule, execute your front call in a TRY / CATCH block to catch errors:

```study
DEFINE status STRING,
    latitude, longitude FLOAT
TRY
    CALL ui.Interface.frontCall("mobile", "getGeolocation",
        [], [status, latitude, longitude] )
CATCH
    ERROR "Could not get coordinates..."
END TRY
```

For more details, see Genero Mobile common front calls on page 2697, Genero Mobile Android front calls on page 2713, Genero Mobile iOS front calls on page 2718.

Accessing Android™ device functions using the Java Interface

**Note:** On Android™ devices, some system functions can only be accessed in the context of a JVM. Use the Java Interface with the `com.fourjs.gma.vm.FglRun` class to access such system specifics.

Web Services on mobile devices

Web Services can be used within mobile applications.

Web Services and Genero Mobile for Android

Points to consider when using Web Services on Android™ platforms:

- V3 SSL Certificates are required.

For complete details about Web Services on GMA, see Web Services on GMA (Android) on page 3277

Web Services and Genero Mobile for iOS

Points to consider when using Web Services on iOS platforms:

- GWS configuration FGLPROFILE entries related to SSL/TLS keys (security.*) are not supported (uses iOS native SSL/TLS).
- When executing a long-running HTTP request, the app may go into background mode, and raise the runtime error -15553 when switching back to foreground mode.

For complete details about Web Services on GMI, see Web Services on GMI (iOS) on page 3276.
Debugging a mobile app

Different solutions are available to debug a mobile app.

**Debugging a mobile app in development mode**

When executing a mobile app program on a server, displaying the user interface on a mobile front-end defined by `FGLSERVER`, it is possible to debug the BDL code with the `fglrun -d` option:

```
$ export FGLSERVER=device-ip-address
$ fglrun -d main.42m
```

For more details, see Starting `fglrun` in debug mode on page 2143.

**AUI protocol debugging**

With app running on a server or on the device, it is possible to show AUI protocol exchanges in the console running the program on the server, by setting the `FGLGUIDEBUG` environment variable to 1. When this variable is set, you can watch user interface events that occur during program execution and how they are treated by the runtime system.

To set the `FGLGUIDEBUG` environment variable for an app running on the device, use an `FGLPROFILE fglrun.environment` entry. The output can be inspected with the program logs as described later in this section.

For more details, see `FGLGUIDEBUG` on page 275.

**AUI protocol logging in development mode**

With an app running on a server, it can be useful to log AUI protocol exchanges between the runtime system and the mobile front-end, to inspect the content, or replay a scenario. This is possible with the `--start-guilog` and `--run-guilog` options of `fglrun`:

```
$ fglrun --start-guilog=case1.log
```

The AUI protocol log file produced by the `--start-guilog` option can then be shared for analysis.

For more details, see Front-end protocol logging on page 1120.

**Debugging a mobile app running on the device**

When executing the mobile app on a device, and if the app has been created with debug mode, it is possible to establish a connection to the runtime system executing on the mobile device, by using the `fgldb` command line tool.

**Important:** On iOS devices, after installing the app, you need to enable the debug port in the app settings, otherwise the app will not listen to the debug port.

For example:

```
$ fgldb -m 192.168.1.23:6400
108 DISPLAY ARRAY contlist TO sr.*
(fgldb)
```

This way you can debug an app running on a device, by using the source code located on the server where the `fgldb` command is executed.

For more details, see Debugging on a mobile device on page 2145.

**Building mobile apps in debug mode**

In order to enable debug features of an app running on a mobile device, you need to build the app in debug mode:

- For Android:
The `gmabuildtool` provides the `--mode debug` option, to create a debug version of the APK.

For more details, see Building Android apps with Genero on page 3591.

• For iOS:

  The `gmibuildtool` provides the `--mode debug` option, to create a debug version of the IPA. The certificate defined in the provisioning profile must be a development certificate.

  **Note:** After installing the debug version of the app on your iOS device, you need to enable the debug port in the app settings.

  For more details, see Building iOS apps with Genero on page 3606.

### Browse the AUI tree created on the mobile front-end side

The content of the Abstract User Interface tree created on the mobile front-end side can be inspected from a web browser, when the app has been created with debug mode, or in development mode by executing the app on a server and displaying on the device.

To inspect AUI tree, open a web browser and enter the following URL:

```
http://device-ip-address:6480 (or 6400)
```

For more details, see Inspecting the AUI tree of a front-end on page 1108.

### Viewing embedded app program logs

The program logs of an app running on a device can be viewed in a browser, if the app was created in debug mode. VM messages (runtime errors, standard output and standard error) are available. This feature is not available if the app is built in release mode.

To inspect program logs, open a web browser and enter the following URL:

```
http://device-ip-address:6480 (or 6400)
```

A menu will then appear in the web page, where you can choose the VM output to be inspected.

### Debugging a WEBCOMPONENT HTML/JavaScript

The HTML content and JavaScript code can be debugged on GMA and GMI mobile devices.

For more details, see Debugging a web component on page 1928.

**Related concepts**

- Integrated debugger on page 2142
  Describes the command-line debugger you can use to find bugs in your programs.
- Program profiler on page 2165
  Find out what function is causing the bottleneck in your program.

---

### Deploying mobile apps

This section describes how to build and deploy mobile apps with Genero.

**Related concepts**

- App execution on page 3572
  Mobile apps are started and stopped, and can switch between foreground and background states when running.
- App localization on page 3572
  Mobile apps can be designed to display localized texts based on the language selected on the device.
- Environment variables on page 3571
You may need to set environment variables for your app.

Database support on mobile devices on page 3583
On the device, a Genero app can use SQL for data management.

**Deploying mobile apps on Android™ devices**
This section contains information to create a mobile application to be deployed on Android™ devices.

**Directory structure for GMA apps**
Platform-specific rules need to be considered when deploying on Android™ devices (GMA).

**The application sandbox**
On Android™ devices, applications are deployed in an application sandbox. The application can access and store data outside of its space, but then the data is also accessible by the other applications.

**Directory structure for a GMA application**
Inside its application sandbox, an Android™ app uses the following directory structure:

```
appdir/
 |-- main.42m
 |-- *.42m
 |-- *.42f
 |-- fglprofile
     ...
 |-- defaults/*.42s
 |-- de/
     |-- *.42s
 |-- fr/
     |-- *.42s
 |-- zh/
     |-- *.42s
     ...
 |-- ... other resource files/dirs ...
  ...
 |-- webcomponents
     |-- component-type
         |-- component-type.html
         |-- other-web-comp-resource
         ...
  |-- appdata/
     |-- ... writable app files ...
 tmpdir/
 |-- ... temporary files ...
```

**Program files directory (appdir)**
Application program files (.42m, .42f, and so on) need to be deployed in the appdir application base directory.

The program files directory can be found in programs with the `base.Application.getProgramDir` on page 2408 method.

**Important:** On Android™, the program files directory returned by the `base.Application.getProgramDir()` method is the same directory as the default working directory, returned by `os.Path.pwd()`.

The FGLAPPDIR environment variable is automatically set to the appdir directory.
Program name (MAIN)

When deploying on mobile devices, the name of the program file must be main.42m or main.42r.

Note: When using the command-line app build scripts, the name of the program file must be main.42?. When using Genero Studio, the packaging script takes care of renaming this file, if you have not named it main.

As with other program files, the "MAIN" module must be located under the appdir application program directory.

Working directory

On Android™ devices, the default current working directory is the appdir directory, and can be used for writable files.

The current working directory can be found in programs with the os.Path.pwd on page 2806 method.

Files that need to be writable (such as SQLite database files) can be created directly under the appdir directory. However, to better organize application files, create sub-directories such as appdir/appdata, keeping original files directly under the appdir directory. For example, create the application database under os.Path.pwd() || "/database".

Temporary directory (tmpdir)

A temporary directory is available for the application.

In order to find the temporary directory for the app, use the standard.feInfo front call, with the "dataDirectory" parameter.

To create a temporary file name, use the os.Path.makeTempName() method.

Language directories for localized strings

When the app starts, the appropriate .42s string files will be loaded from the directory corresponding to the current language settings of the mobile device. String files to be loaded can be defined in the app's fglprofile, or you can use the main program name to avoid fglprofile settings.

For each language supported by your application, a directory must exist under appdir, with a name including the locale codes. Default string files (in English for example) can be provided under appdir/defaults, in case the regional settings of the device do not match one of the locale directories of the app, otherwise the application will stop with error -8006.

For example:

```
appdir/defaults/mystings.42s
appdir/fr/mystings.42s
appdir/de/mystings.42s
```

For more details, see Localized string files on mobile devices on page 546.

Deploying a custom fglprofile file

If you need to set fglprofile entries for your mobile application, create a file with the name fglprofile, and deploy it under the appdir directory, along with the other program files.

See Understanding FGLPROFILE on page 255 for more details about fglprofile settings.

Creating the initial database file

When a mobile application starts for the first time, it typically creates a new database, or copies an existing database template file from the file directory (base.Application.getProgramDir on page 2408) to the working directory (os.Path.pwd on page 2806).
Note: It is recommended that different database file names are used for the original and final application database, as folders pointed to by `base.Application.getProgramDir()` and `os.Path.pwd()` can be the same on Android™ devices.

For more details about database creation on mobile devices, see Creating a database from programs on page 608.

Building Android™ apps with Genero
Genero provides a command-line tool to create applications for Android™ devices.

Basics
Genero mobile apps for Android™ are distributed as APK packages like any other Android™ app. APK packages can be published on the Google Play Store.

Genero provides the `gmabuildtool` command line tool, to build the APK package for your mobile application.

Important: You need to comply with some Google conditions before you can publish your app on the Play Store. This topic describes how the `gmabuildtool` tool builds Google Play Store ready APK packages.

For testing purposes, the tool can deploy and automatically launch the app on a specific device or simulator.

The tool has also an option to update the Android™ SDK and an option to manage scaffold archives.

Note: This documentation section implies that you are familiar with Android™ app programming concepts and requirements. For example, you will need the Android™ SDK tools to be installed (and up to date) to build your Android™ apps. For more details, visit the Android SDK tools download page.

Prerequisites
Before starting the command line tool to build or deploy the app, fulfill the following prerequisites:

1. The Genero BDL development environment (FGLDIR) must be installed to compile your program files.
2. The Java JDK must be installed.
   
   Important: JDK version 8 is required to build Android™ apps. For the latest information regarding system requirements and Java support, please refer to the Supported platforms and databases document, available on the “Products” download page of the Four Js Web site.
3. The GMA software components must be installed.
   
   The GMA buildtool and GMA binary archive are provided in the GMA distribution archive (`fjs-gma-*.zip`).

To set up the GMA installation directory, perform the following steps:

a. Create a dedicated directory (`gma-install-dir`)

b. Extract the content of the `fjs-gma-*.zip`.

c. Add the `gma-install-dir` directory to your PATH environment variable, in order to find the `gmabuildtool` command.

The `gma-install-dir` will contain the `gmabuildtool` command, and the original GMA scaffold archive (`gma-install-dir/artifacts`).

4. The Android SDK must be installed (an internet connection is required to download the SDK packages). For more details, see Install Genero Mobile for Android on page 40.
5. Download/upgrade the required Android™ SDK packages, by using the `gmabuildtool updatesdk` command.

   Note: Execute the `gmabuildtool updatesdk` command every time a new version of the GMA buildtool and GMA binary archive is installed.

   For more details, see Update the Android SDK with the GMA buildtool on page 3592.
6. Android™-specific app resources such as icons (in all required sizes) are required, along with the application program files.
7. If you plan to publish your app on Google Play, register to Google Play as a developer and create a Google Play project.
Environment settings
Define the following environment variables before starting the command-line buildtool:

- Android™ SDK environment settings (ANDROID_SDK_ROOT, PATH)
- Java JDK environment settings (JAVA_HOME, PATH)

Update the Android™ SDK with the GMA buildtool
After a fresh installation of the GMA buildtool and GMA binary archive, upgrade the Android™ SDK, to download all required Android™ SDK packages, with the gmabuildtool updatesdk command.

The Android™ SDK installation directory is required for the SDK update, and is found in ANDROID_SDK_ROOT environment variable, or with the --android-sdk option:

```
gmabuildtool updatesdk
   --android-sdk /use/local/32bits/android-sdk/r22.6.2
```

Use the --accept-licenses option to silently accept all Android™ SDK licenses.

```
gmabuildtool updatesdk
   --accept-licenses
   ...
```

If you need to specify a proxy to download the Android™ SDK, use the --proxy-host and --proxy-port options:

```
gmabuildtool updatesdk
   --proxy-host amadeus --proxy-port 3232
   ...
```

Use the --no-install-extras option to skip the installation of extra SDK modules such as Google's driver for Windows® and the HAXM for Windows® and OS X:

```
gmabuildtool updatesdk
   --no-install-extras
   ...
```

Manage GMA scaffold archives
The GMA scaffold archives can be managed by the gmabuildtool scaffold command.

In order to get the list of plugins installed in the GMA development environment, use the --list-plugins option:

```
$ gmabuildtool scaffold --list-plugins
   ...
cordova-plugin-device-motion.aar
cordova-plugin-media.aar
   ...
```

To install additional plugins in the GMA installation directory, use the --install-plugins option, for example:

```
$ gmabuildtool scaffold --install-plugins path-to-plugin-sources
```

For more usage examples, see Cordova plugins on page 3644.
Building your APK with the GMA buildtool

The gmabuildtool build ... command creates the APK from a set of files, and following the options passed as parameter.

```
gmabuildtool build ...
        build options ...
```

The process will be explained in details in the next sections of this topic.

For a complete description of the build command options, see gmabuildtool on page 2096.

Gradle build cache

To build Android™ apps, the GMA buildtool uses the Gradle toolkit.

Gradle can speed up APK creation time, by reusing outputs saved in a cache produced from previous builds. The gmabuildtool command uses the Gradle build cache.

On Unix-like platforms, the gmabuildtool Gradle build cache is in the /tmp/gma directory. On Windows®, it is in C:\tmp\gma.

**Important:** The Gradle build cache directory used by gmabuildtool (/tmp/gma or C:\tmp\gma) can grow to a significant size and consequently fill the disk. Take care to monitor the size of this cache and manually clean out the cache files, to ensure that it does not become too large in size.

Cleaning intermediate build files

The build process is optimized to avoid a complete APK rebuild every time you invoke the GMA buildtool.

The GMA buildtool creates intermediate archive files, that can be reused in the next build, if no changes are detected in application program files. However, these files might be corrupted, in case of user interruption or gradle build failure.

In this situation, you can use the --clean option of the gmabuildtool build ... command, to clean up the scaffold build directory, and restart with a fresh build:

```
gmabuildtool build --clean ...
        build options ...
```

Force scaffold update during build

During the build process, if not yet done, the GMA buildtool will automatically unzip the original GMA scaffold archive (gma-install-dir/artifacts) into the --build-project directory. During next builds, the scaffold files present in the project build directory will be reused.

If needed (especially, when upgrading GMA), you can force an update of these scaffold files with the --build-force-scaffold-update option of the gmabuildtool build command:

```
gmabuildtool build --build-force-scaffold-update ...
        build options ...
```

**Note:** When using the --build-force-scaffold-update option, the GMA buildtool will ask for a confirmation to remove the scaffold files in your project directory. In order to build silently and answer yes to all questions asked during the build process, use the --build-quietly option:

```
gmabuildtool build --build-force-scaffold-update --build-quietly ...
        build options ...
```
Specifying the GBC for Universal Rendering

When your app uses Universal Rendering mode, use the --build-gbc-runtime option to specify which GBC has to be bundled in the APK:

```
gmabuildtool build --build-gbc-runtime gbc-archive ...
```

The gbc-archive parameter is the ZIP archive of the GBC front-end, to be created as described in the Create a runtime zip topic in the Genero Browser Client User Guide.

If the --build-gbc-runtime option is used, the embedded applications will implicitly be displayed with Universal Rendering. There is no need to set the gui.rendering option in the FGLPROFILE. However, if gui.rendering="native" is set, the native GUI mode will be used, even if the GBC option is specified during the build.

Using an options file

To simplify option specification, create a file with the list of options to be passed to the gmabuildtool with the --input-options argument. Each option must be specified in a dedicated line (note that the main command argument appears in the first line):

```
$ cat myoptions.txt
build
   --build-output-apk-name MyApp
   --build-app-name MyApp
   --build-app-package-name com.example.myapp
   ...
$ gmabuildtool --input-options ./myoptions.txt
```

Elements used in building the Android™ app

The gmabuildtool build command builds the Android™ APK package from the following:

- The GMA binary archive, containing the GMA front-end and the FGLGWS runtime system. The original scaffold is provided in gma-install-dir/artifacts and unzipped/copied for APK builds into a directory defined by the --build-project option.
- The GBC to be used for Universal Rendering (--build-gbc-runtime option),
- The compiled application program and resource files (.42m, .42f, etc) (--build-app-genero-program* options),
- The prefix for the APK file name to be generated (--build-output-apk-name option),
- The name of the app (--build-app-name option),
- The version code of the app (--build-app-version-code option),
- The version name of the app (--build-app-version-name option),
- Android™ app specific resources:
  - Android™ app icons (all sizes) (--build-app-icon* and --build-status-icon-* options).
  - Android™ app specifics (to sign the app, not required in development mode):
    - The keystore alias, used with the keytool to generate the keystore file (--build-jarsigner-alias option).
    - The keystore file, generated from keytool (for the --build-jarsigner-keystore option).

Generate the keystore file to sign your app

In order to build an APK that can be deployed on the market (Google Play), you need to sign your Android™ app.

First, you need to generate a keystore file with the keytool Android™ utility.
The keystore file and keystore alias will be used by the gmabuildtool to sign the APK with the jarsigner utility. These signing credentials are passed to the buildtool with the --build-jarsigner-keystore and --build-jarsigner-alias options.

**Important:** There is critical information regarding app signing in the Android documentation. For example, in order to install new versions as updates to your app, each APK must use the same certificate: The signing key must not change during the lifetime of your app. Make sure that you read all details about Android app signing.

### Generated APK file name

The file name of the APK package is formed from:

1. The APK file name prefix defined by the --build-output-apk-name option (by default, "app"),
2. When building a debug version, the -debug suffix,
3. The .apk file extension.

For example, if the APK file name prefix is MyApp and is a debug package, the resulting APK file name will be: MyApp-debug.apk.

### Default build directory structure

For convenience, the buildtool supports a default directory structure to find all files required to build the APK:

```plaintext
<table>
<thead>
<tr>
<th>top-dir</th>
</tr>
</thead>
<tbody>
<tr>
<td>-- main.42m and other program files, as described in Directory structure for GMA apps on page 3589</td>
</tr>
<tr>
<td>-- gma</td>
</tr>
<tr>
<td>-- project</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>-- ic_app_hdpi.png</td>
</tr>
<tr>
<td>-- ic_app_mdpi.png</td>
</tr>
<tr>
<td>-- ic_app_xhdpi.png</td>
</tr>
<tr>
<td>-- ic_app_xxhdpi.png</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
```

In the above directory structure:

1. `top-dir` is the top directory of the default structure. It typically holds your application program files. The program files directory can be specified with the --build-app-genero-program option.
2. `top-dir/gma` is the default directory containing the GMA binary archive and the app icons.
3. `top-dir/gma/project` contains the scaffold files copied automatically by gmabuildtool from gma-install-dir/artifacts. This directory can be specified with the --build-project option.

### Android™ permissions

To use a mobile device feature such as the camera, an Android™ app must be created with the corresponding Android™ permission.

Android™ distinguishes "Normal Permissions" and "Dangerous Permissions": Dangerous Permissions require a user validation at runtime, the first time the device feature is used.

**Note:** Before Android™ 6, Dangerous Permissions defined by the app were set at app installation. Starting with version 6, Dangerous Permissions must be requested by the app code on demand.

Android™ permissions can be specified with the --build-app-permissions option of the gmabuildtool. Define the list of permissions as a single argument, by using the comma as separator.

For example:

```plaintext
gmabuildtool build \
```
Assuming that the permission was specified when building the APK, a Dangerous Permission required for a core GMA feature (like internet access), or a built-in front call, will make the GMA automatically ask for user confirmation. For example, if the app code makes a `chooseContact` front call, the GMA will automatically request the user to access the contacts database.

Other permissions (not related to core GMA features or built-in front calls) need to be specified when building the app.

**Note:** Dangerous Permissions not related to core GMA features or built-in front calls, need to be enabled by the app code. To request the user for a specific permission, perform a `askForPermission` front call, before using the feature.

The permissions listed below show Android™ "Normal Permissions", required for core GMA app features and built-in mobile front calls.

**Note:** Normal Permissions listed below are set by default. They do not need to be specified when building the APK.

- `android.permissionINTERNET`
- `android.permissionACCESS_NETWORK_STATE`
- `android.permissionCHANGE_NETWORK_STATE`
- `android.permissionACCESS_WIFI_STATE`
- `android.permissionREORDER_TASKS`
- `android.permissionKILL_BACKGROUND_PROCESSES`
- `android.permissionMOUNT_FORMAT_FILESYSTEMS`
- `android.permissionREAD_LOGS`
- `android.permissionWAKE_LOCK`
- `com.google.android.c2dm.permission.RECEIVE`
- `packageName.permission.C2D_MESSAGE`

The following list shows the Android™ "Dangerous Permissions", required for core GMA app features and built-in mobile front calls.

**Important:** Dangerous Permissions listed below are not set by default: They must be specified explicitly with the `--build-app-permissions` option, when building the APK.

- `android.permissionACCESS_FINE_LOCATION`: For `getGeolocation` on page 2703 front call.
- `android.permissionACCESS_COARSE_LOCATION`: For `getGeolocation` on page 2703 front call.
- `android.permissionREAD_CONTACTS`: For `chooseContact` on page 2699 front call.
- `android.permissionGET_ACCOUNTS`: Only on Android™ 5.1 and lower (< API 23), for `chooseContact` on page 2699 front call.
- `android.permissionREAD_PHONE_STATE`: For `feInfo (deviceid, iccid, imei)` front calls. Starting with Android 10 (API level 29), to access deviceid, iccid and imei identifiers, the device must be configured manually by adding the `READ_PRIVILEGED_PHONE_STATE` privileged permission. To set this permission manually, you need an Android Enterprise Program with fully managed devices (devices are owned by your company). Third-party apps installed from the Google Play Store can’t declare such privileged permissions. For more details, see [Android documentation](https://developer.android.com/guide/topics/permissions/android-permissions).
- `android.permissionWRITE_EXTERNAL_STORAGE`: For `importContact` on page 2706, `takeVideo` on page 2712, `takePhoto` on page 2712 front calls.
- `android.permissionREAD_EXTERNAL_STORAGE`: For `chooseVideo` on page 2700, `takeVideo` on page 2712, `choosePhoto` on page 2699, `takePhoto` on page 2712 front calls.

For a complete list of Android™ permissions, see [Android's Manifest permissions](https://developer.android.com/guide/topics/manifest/). 

```java
--build-app-permissions android.permission.READ_CALENDAR,...
```
**Define app's color theme**

To customize your Android™ app, it can be created with a color theme defined by a list of colors. The colors are specified as a comma-separated list of RGB colors with the --build-app-colors option.

**Note:** This feature is only available with Android™ 5.0 / SDK 21 and higher. With older versions of Android™, the colors specified with the --build-app-colors option will not take effect.

The value provided to the --build-app-colors option must be a comma-separated list of four hexadecimal RGB colors.

The position of the RGB value in the color list defines its purpose:

1. colorPrimary: The main color used in the app.
2. primaryDark: The color used for the status bar and the navigation bar.
3. accent: The accent color used for widgets and table lines.
4. actionbarText: The foreground color for the texts in the action bar.
5. primaryText: The text color for items in the whole application.
6. windowBackground: The window background color.
7. navigationBarBackground: The background color of the bottom bar.

By default, the color theme is the Genero purple color.

For example, to define a red color theme, use the following combination:

```
gmabuildtool build \
  ... \
  --build-app-colors \
  "#F44336,#B71C1C,#EF9A9A,#FFFFFF,#FF0000,#00AA00,#DDAA00" \
  ...
```

For more details, see Android material theme and Android color palette.

**Debug and release versions**

Android™ apps can be generated in a debug or release version. Release versions meet the requirements for distribution on Google Play, while debug versions are used in development. In debug mode, the app installed on the device will listen on the debug TCP port to allow fgldb -m connections.

Debug or release mode can be controlled with the --build-mode option of the gmbuildtool command:

```
gmabuildtool build \
  --build-mode debug \
  ...
```

By default the app is built in release mode.

**Building an Android™ app with gmbuildtool**

Follow the next steps to setup a GMA app build directory in order to create an Android™ app, based on the default directory structure:

1. Create the root distribution directory (top-dir)
2. Copy compiled program files (.42m, .42f, fglprofile, application images, web component files, etc) under top-dir.
3. Copy the default English .42s compiled string resource file under top-dir.
4. Create non-English language directories (fr, ge, ...) under top-dir and copie the corresponding .42s files.
5. Copy default application data files (database file for ex) under top-dir.
6. Create the top-dir/gma directory.
7. Copy Android™ app resources (icons) under top-dir/gma.
Once the build directory is prepared, issue the following commands to build the APK:

```bash
$ cd top-dir
$ gmabuildtool build \
   --android-sdk /home/mike/android/sdk \
   --build-force-scaffold-update \
   --build-apk-outputs /home/mike/work/example/outputs \
   --build-output-apk-name MyApp \
   --build-app-name MyApp \
   --build-app-package-name com.example.myapp \
   --build-app-version-name "10.02" \
   --build-jarsigner-alias android_alias \
   --build-jarsigner-keystore /home/mike/work/example/sign/android.keystore \
   --build-mode release \
   --build-app-permissions android.permission.ACCESS_WIFI_STATE,android.permission.CALL_PHONE
```

**Building an app with GMA custom extensions**

The `gmabuildtool build` command supports APK creation for applications using GMA custom extensions written in Java.

Before building the APK package, create the custom GMA binary archive with your extensions, as described in [Packaging custom Java extensions for GMA](#) on page 2223.

When your custom GMA binary archive is complete, build the APK package with the `gmabuildtool build` command. Use the `--build-project` option to specify the path to the Android™ Studio project that was used to build your custom GMA binary archive:

```bash
$ gmabuildtool build \
   ... \
   --build-project /home/mike/android_project/mycustgma \
   ...
```

**Note:** Other options have to be specified as for a regular build using the original standard GMI binary archive.

**Deploy and launch the app**

After building the APK package, for testing purposes, you can deploy and launch your app from the command line with the `gmabuildtool test` command.

**Note:** The `test` command is provided for development only. To deploy your app in production for several devices, use the regular publication channel of Android™ apps.

In order to deploy and launch the app, you must provide:

1. the path to the APK file

There must be only one Android™ device connected or running Android™ emulator.

```bash
$ gmabuildtool test \
   --test-apk /home/mike/work/example/outputs/MyApp-arm-debug.apk
```

**Related concepts**

[Genero Mobile common front calls](#) on page 2697

This section describes common front calls provided by all mobile front-ends.

[Color and theming](#) on page 3581
Mobile applications must follow the platform colors and theming.

**gmabuildtool**
The gmabuildtool is a utility to create and test applications for an Android™ device.

**Syntax**

```
 gmabuildtool command [option [...]]
```

1. *command* can be one of the following:
   - *updatesdk*: updates the Android™ SDK, to download packages required by GMA.
   - *scaffold*: manages scaffold archives.
   - *build*: builds an APK package.
   - *test*: deploys and launches an app on the device or emulator.

2. *option* can be a general or command-specific option, as described in Options on page 3599.

**Options**

**Table 754: General gmabuildtool options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Short option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--android-sdk <em>path</em></td>
<td>-as</td>
<td>The path to the Android™ SDK installation directory.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If not specified, it defaults to the ANDROID_SDK_ROOT environment variable.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If ANDROID_SDK_ROOT is not defined, defaults to ANDROID_HOME.</td>
</tr>
<tr>
<td>--help</td>
<td>-h</td>
<td>Display options for the tool.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Path to the file containing gmabuildtool options.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Define all options in a file and pass the file to the gmabuildtool command</td>
</tr>
<tr>
<td></td>
<td></td>
<td>with the --input-options argument.</td>
</tr>
<tr>
<td>--input-options <em>path</em></td>
<td>-i</td>
<td>The options file must use the following format:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>command *option-name* *option-value*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>--java-home <em>path</em></td>
<td>-jh</td>
<td>Java home path.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default is JAVA_HOME.</td>
</tr>
<tr>
<td>--proxy-host <em>host</em></td>
<td>-ph</td>
<td>Defines the proxy host.</td>
</tr>
<tr>
<td>--proxy-host <em>port</em></td>
<td>-pp</td>
<td>Defines the proxy port.</td>
</tr>
<tr>
<td>--verbose-fine</td>
<td>-v</td>
<td>Verbose mode (level 1)</td>
</tr>
<tr>
<td>--verbose-finer</td>
<td>-vv</td>
<td>Verbose mode (level 2)</td>
</tr>
<tr>
<td>--verbose-finest</td>
<td>-vvv</td>
<td>Verbose mode (level 3) - shows all possible logs.</td>
</tr>
<tr>
<td>--version</td>
<td>-V</td>
<td>Displays version information.</td>
</tr>
</tbody>
</table>
Table 755: gmabuildtool updatesdk options

<table>
<thead>
<tr>
<th>Option</th>
<th>Short option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--accept-licenses</td>
<td>-al</td>
<td>Silently accept Android™ SDK licenses when the Android™ SDK is updated.</td>
</tr>
<tr>
<td>--no-install-extras</td>
<td>-uN</td>
<td>Avoid installation of extra SDK modules.</td>
</tr>
</tbody>
</table>

Table 756: gmabuildtool scaffold options

<table>
<thead>
<tr>
<th>Option</th>
<th>Short option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--install-plugins</td>
<td>-ip</td>
<td>Install the specified plugins in the scaffold archive.</td>
</tr>
<tr>
<td>plugin-list</td>
<td></td>
<td>The plugin-list must be a comma-separated list of plugins.</td>
</tr>
<tr>
<td>--list-plugins</td>
<td>-lp</td>
<td>List the plugins available in the scaffold archive.</td>
</tr>
</tbody>
</table>

Table 757: gmabuildtool build options

<table>
<thead>
<tr>
<th>Option</th>
<th>Short option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--clean</td>
<td>-c</td>
<td>Cleans the intermediate build files before a rebuild.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Use the --clean option if the previous build was interrupted or has failed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> The --clean option does not remove and replace the scaffold, as done by the --build-force-scaffold-update option.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Define the Android™ color theme for the app (Android™ 5.0+ / SDK 21+)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The value must be a comma-separated list of four hexadecimal RGB colors: #F44336,#B71C1C,#EF9A9A,...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The position of the RGB value in the color list defines its purpose:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1. colorPrimary: The main color used in the app.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. primaryDark: The color used for the status bar and the navigation bar.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3. accent: The accent color used for widgets and table lines.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4. actionBarText: The foreground color for the texts in the action bar.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5. primaryText: The text color for items in the whole application.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6. windowBackground: The window background color.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7. navigationBarBackground: The background color of the bottom bar.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>By default, the color theme is the Genero purple color.</td>
</tr>
</tbody>
</table>
### Option

<table>
<thead>
<tr>
<th>Option</th>
<th>Short option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--build-app-genero-program-main path</td>
<td>-bgpm</td>
<td>Relative path to the main module of the application (can be .42m or .42r). Defaults to main.42m</td>
</tr>
<tr>
<td>--build-app-genero-program path</td>
<td>-bgp</td>
<td>Defines the path to the application program files (.42m, .42f, etc) The contents of this directory will be zipped and bundled inside APKs. This option can handle an already zipped Genero program archive. If not specified, defaults to the current working directory. <strong>Note</strong>: The path defined by this option is used as base directory for other options such as --build-project and application icon resources options.</td>
</tr>
<tr>
<td>--build-app-icon-hdpi path</td>
<td>-bih</td>
<td>Defines the path to application icon in hdpi. Default is top-dir/gma/ic_app_hdpi.png, where top-dir is defined by the --build-app-genero-program option.</td>
</tr>
<tr>
<td>--build-app-icon-mdpi path</td>
<td>-bim</td>
<td>Defines the path to application icon in mdpi. Default is top-dir/gma/ic_app_mdpi.png, where top-dir is defined by the --build-app-genero-program option.</td>
</tr>
<tr>
<td>--build-app-icon-xhdpi path</td>
<td>-bixh</td>
<td>Defines the path to application icon in xhdpi. Default is top-dir/gma/ic_app_xhdpi.png, where top-dir is defined by the --build-app-genero-program option.</td>
</tr>
<tr>
<td>--build-app-icon-xxhdpi path</td>
<td>-bixxh</td>
<td>Defines the path to application icon in xxhdpi. Default is top-dir/gma/ic_app_xxhdpi.png, where top-dir is defined by the --build-app-genero-program option.</td>
</tr>
<tr>
<td>--build-app-name app-name</td>
<td>-bn</td>
<td>Application name. If not specified, the application name defaults to the current working directory.</td>
</tr>
<tr>
<td>--build-app-package-name name</td>
<td>-bpn</td>
<td>APK package name. It is recommended to format the package name as &quot;com.organization-name.app-name&quot;. If not specified, the application package name defaults to com.example.current-working-directory Android™ application permissions. The list of permissions is provided as a comma separated list of android.permission.* identifiers. For more details, see Android permissions on page 3595.</td>
</tr>
<tr>
<td>--build-app-permissions permissions</td>
<td>-ba</td>
<td></td>
</tr>
</tbody>
</table>

| Description |

Relative path to the main module of the application (can be .42m or .42r).

Defaults to main.42m

Defines the path to the application program files (.42m, .42f, etc)

The contents of this directory will be zipped and bundled inside APKs. This option can handle an already zipped Genero program archive.

If not specified, defaults to the current working directory.

**Note**: The path defined by this option is used as base directory for other options such as --build-project and application icon resources options.

Defines the path to application icon in hdpi.

Default is top-dir/gma/ic_app_hdpi.png, where top-dir is defined by the --build-app-genero-program option.

Defines the path to application icon in mdpi.

Default is top-dir/gma/ic_app_mdpi.png, where top-dir is defined by the --build-app-genero-program option.

Defines the path to application icon in xhdpi.

Default is top-dir/gma/ic_app_xhdpi.png, where top-dir is defined by the --build-app-genero-program option.

Defines the path to application icon in xxhdpi.

Default is top-dir/gma/ic_app_xxhdpi.png, where top-dir is defined by the --build-app-genero-program option.

Application name. If not specified, the application name defaults to the current working directory.

APK package name. It is recommended to format the package name as "com.organization-name.app-name".

If not specified, the application package name defaults to com.example.current-working-directory Android™ application permissions.

The list of permissions is provided as a comma separated list of android.permission.* identifiers. For more details, see Android permissions on page 3595.
<table>
<thead>
<tr>
<th>Option</th>
<th>Short option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--build-app-version-code version-code</td>
<td>-bvc</td>
<td>Application version code. For example: 100915 The value of this option must be an integer (do not use decimal numbers).</td>
</tr>
<tr>
<td>--build-app-version-name version-name</td>
<td>-bvn</td>
<td>Application version name. For example: 10.09.15 This will be the actual app version visible on devices.</td>
</tr>
<tr>
<td>--build-apk-outputs path</td>
<td>-bo</td>
<td>Defines the destination folder where the APK packages must be created. Defines Cordova plugins to be embedded in the app package. When specifying multiple cordova plugins, use the comma (,) as separator.                                                                                                       The name of the plugin must match the Git repository name. It is case-sensitive. Note: To get the list of available Cordova plugins, use the gmabuildtool scaffold --list-plugins command. For further information, see Cordova plugins on page 3644.</td>
</tr>
<tr>
<td>--build-cordova cordova-plugin-names</td>
<td>-bco</td>
<td>The name of the plugin must match the Git repository name. It is case-sensitive. Note: To get the list of available Cordova plugins, use the gmabuildtool scaffold --list-plugins command. For further information, see Cordova plugins on page 3644.</td>
</tr>
<tr>
<td>--build-force-scaffold-update</td>
<td>-bfsu</td>
<td>Forces to re-create the app project directory with the original GMA scaffold directory (defined by --build-project) Defines the GBC to be used for Universal Rendering. gbc-archive is the GBC ZIP archive. For more details, see the Create a runtime zip topic in the Genero Browser Client User Guide.</td>
</tr>
<tr>
<td>--build-gbc-runtime gbc-archive</td>
<td>-bgr</td>
<td>Jarsigner alias. This is the alias provided to the keystore utility to build the keystore file to sign the app. Used when APK artifacts are signed. Jarsigner keypass. Specifies the password used to protect the private key of the keystore entry addressed by the alias specified in the --build-jarsigner-alias option. The password is required when using jarsigner to sign a JAR file. Used when APK artifacts are signed. Jarsigner keystore path. This is the path to the keystore file generated by the keystore utility to sign the app. Used when APK artifacts are signed.</td>
</tr>
<tr>
<td>--build-jarsigner-alias alias</td>
<td>-bja</td>
<td>Jarsigner alias. This is the alias provided to the keystore utility to build the keystore file to sign the app. Used when APK artifacts are signed. Jarsigner keypass. Specifies the password used to protect the private key of the keystore entry addressed by the alias specified in the --build-jarsigner-alias option. The password is required when using jarsigner to sign a JAR file. Used when APK artifacts are signed. Jarsigner keystore path. This is the path to the keystore file generated by the keystore utility to sign the app. Used when APK artifacts are signed.</td>
</tr>
<tr>
<td>--build-jarsigner-keypass keypass</td>
<td>-bjk</td>
<td>Jarsigner keypass. Specifies the password used to protect the private key of the keystore entry addressed by the alias specified in the --build-jarsigner-alias option. The password is required when using jarsigner to sign a JAR file. Used when APK artifacts are signed. Jarsigner keystore path. This is the path to the keystore file generated by the keystore utility to sign the app. Used when APK artifacts are signed.</td>
</tr>
<tr>
<td>--build-jarsigner-keystore path</td>
<td>-bjks</td>
<td>Jarsigner keypass. Specifies the password used to protect the private key of the keystore entry addressed by the alias specified in the --build-jarsigner-alias option. The password is required when using jarsigner to sign a JAR file. Used when APK artifacts are signed. Jarsigner keystore path. This is the path to the keystore file generated by the keystore utility to sign the app. Used when APK artifacts are signed.</td>
</tr>
<tr>
<td>Option</td>
<td>Short option</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>--build-jarsigner-storepass storepass</code></td>
<td><code>-bjs</code></td>
<td>Jarsigner storepass. Specifies the password that is required to access the keystore. Used when APK artifacts are signed.</td>
</tr>
<tr>
<td>`--build-mode release</td>
<td>debug`</td>
<td><code>-bm</code></td>
</tr>
<tr>
<td><code>--build-output-apk-name name</code></td>
<td><code>-ban</code></td>
<td>Defines the prefix for the APK packages names. By default, this prefix is &quot;app&quot;. The file name of the APK package is formed from: 1. The APK file name prefix defined by the <code>--build-output-apk-name</code> option (by default, &quot;app&quot;). 2. When building a debug version, the <code>-debug</code> suffix, 3. The <code>.apk</code> file extension. For example, if the APK file name prefix is <code>MyApp</code> and is a debug package, the resulting APK file name will be: <code>MyApp-debug.apk</code>.</td>
</tr>
<tr>
<td><code>--build-project path</code></td>
<td><code>-bp</code></td>
<td>Defines the path to the directory containing the original GMA binary archive files (i.e. scaffolding), or the directory containing the Android™ Studio project, when building a customized GMA. Default is <code>top-dir/gma/project</code>, where <code>top-dir</code> is defined by the <code>--build-app-genero-program</code> option. Forces a silent build, by answering yes to all questions asked during the build process. By default, the user must answer to the build questions by yes/no. Status icon path for hdpi (high dots per inch) size. The default path is <code>top-dir/gma/ic_status_hdpi.png</code>, where <code>top-dir</code> is defined by the <code>--build-app-genero-program</code> option. If this option is not specified, yet you provide default files under the gma directory named like those defined for the default path, your package will use these files. If you don't provide any status icon files, the default files are used.</td>
</tr>
</tbody>
</table>
## Mobile applications

**Option** | **Short option** | **Description**
--- | --- | ---
`--build-status-icon-mdpi path` | `-bsm` | Status icon path for mdpi (medium dots per inch) size.
The default path is `top-dir/gma/ic_status_mdpi.png`, where `top-dir` is defined by the `--build-app-genero-program` option. If this option is not specified, yet you provide default files under the gma directory named like those defined for the default path, your package will use these files. If you don't provide any status icon files, the default files are used.

`--build-status-icon-xhdpi path` | `-bsxh` | Status icon path for xhdpi (extra-high dots per inch) size.
The default path is `top-dir/gma/ic_status_xhdpi.png`, where `top-dir` is defined by the `--build-app-genero-program` option. If this option is not specified, yet you provide default files under the gma directory named like those defined for the default path, your package will use these files. If you don't provide any status icon files, the default files are used.

**Table 758: gmabuiltool test options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Short option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--test-apk path</code></td>
<td><code>-ta</code></td>
<td>Path to the APK file to deploy and launch for testing.</td>
</tr>
</tbody>
</table>

**Related concepts**

- [Building Android apps with Genero](#) on page 3591

Generero provides a command-line tool to create applications for Android™ devices.

## Deploying mobile apps on iOS devices

This section contains information to create a mobile application to be deployed on iOS devices.

### Directory structure for GMI apps

Platform-specific rules need to be considered when deploying on iOS devices (GMI).

### The application sandbox

On iOS devices, program interactions with the file system are limited to the directories inside the app's sandbox.

### Directory structure for a GMI application

Inside its application sandbox, an iOS app uses the following directory structure:

```
appdir/
 |-- main.42m
 |-- *.42m
 |-- *.42f
 |-- fglprofile
 |   ...
 |-- defaults/*.42s
 |-- de/
 |   |-- *.42s
```

Program files
Program files directory (*appdir*)

Application program files (.42m, .42f, as well as other program resources) need to be deployed in the *appdir* directory.

**Important:** On iOS, the application program directory is read-only. Only the "Documents" directory is writable.

The program files directory can be found in programs with the `base.Application.getProgramDir` on page 2408 method.

The `FGLAPPDIR` environment variable is automatically set to the *appdir* directory.

Program name (**MAIN**)

When deploying on mobile devices, the name of the program file must be `main.42m` or `main.42r`.

**Note:** When using the command-line app build scripts, the name of the program file must be `main.42?`. When using Genero Studio, the packaging script takes care of renaming this file, if you have not named it `main`.

As with other program files, the "MAIN" module must be located under the *appdir* application program directory.

Working directory

The current working directory for an iOS application is typically a writable "Documents" directory, in the private folder of the app. For example, the path to the working directory can be "'/private/var/mobile/.../Documents".

The current working directory can be found in program with the `os.Path.pwd` on page 2806 method.

**Note:** Any file access without an absolute path will be relative to the current working directory.

Files that need to be writable (such as SQLite database files) must be created or copied from the program files directory into the working directory. Copy must be done by the app at first execution, by using `base.Application.getProgramDir` on page 2408, to find the program files directory, and `os.Path.pwd()`, to find the working directory.

Temporary directory (**tmpdir**) 

A temporary directory is available for the application.

In order to find the temporary directory for the app, use the `standard.fInfo` front call, with the "dataDirectory" parameter.

To create a temporary file name, use the `os.Path.makeTempName()` method.
Language directories for localized strings

When the app starts, the appropriate .42s string files will be loaded from the directory corresponding to the current language settings of the mobile device. String files to be loaded can be defined in the app's fglprofile, or you can use the main program name to avoid fglprofile settings.

For each language supported by your application, a directory must exist under appdir, with a name including the locale codes. Default string files (in English for example) can be provided under appdir/defaults, in case the regional settings of the device do not match one of the locale directories of the app, otherwise the application will stop with error -8006.

For example:

```
appdir/defaults/mystrings.42s
appdir/fr/mystrings.42s
appdir/de/mystrings.42s
```

For more details, see Localized string files on mobile devices on page 546.

Deploying a custom fglprofile file

If you need to set fglprofile entries for your mobile application, create a file with the name fglprofile, and deploy it under the appdir directory, along with the other program files.

See Understanding FGLPROFILE on page 255 for more details about fglprofile settings.

Creating the initial database file

When a mobile application starts for the first time, it typically creates a new database, or copies a existing database template file from the appdir program file directory (base.Application.getProgramDir on page 2408) to the working directory (os.Path.pwd on page 2806).

For more details about database creation on mobile devices, see Creating a database from programs on page 608.

Building iOS apps with Genero

Genero provides a command-line tool to build applications for iOS devices.

Basics

Genero mobile apps for iOS are distributed as IPA packages, like any other iOS app. Genero provides a command line tool to build the .ipa package for your mobile application, or the .app directory for simulators.

Note: This documentation section implies that you are familiar with iOS app programming concepts and requirements. In order to build your apps, you must have an Apple® developer account, as well as certificates and provisioning profiles to deploy your apps. For more details, visit the Apple developer site.

Prerequisites

Before starting the command line tool to build or deploy the app, fulfill the following prerequisites:

1. The Genero BDL development environment (FGLDIR) must be installed on the Mac computer to compile your program files.
2. The GMI archive (and gmibuildtool) must be installed and available.
   
   The GMI archive is provided as a ZIP archive (fjs-fglgmi-*.*.zip). Extract the archive into FGLDIR.
   
   Warning: If the GMI archive is not extracted into FGLDIR, the Xcode® project of the $FGLDIR/demo/MobileDemo/userfrontcall demo does not work.

For test purposes, the GMI zip archive can be installed elsewhere in a location of your choice. You may choose to do this to resolve problems resulting from several GMI versions. If you install elsewhere, define an environment variable GMIDIR on page 284 with the install location and add $GMIDIR/bin to the path.
**Note:** The GMI buildtool does not need GMIDIR; However, all the supplied GMI demos need GMIDIR in the Makefiles.

Check that the gmibuildtool command is available (add GMI-install-dir/bin to PATH for convenience).

**Important:** When re-installing a new GMI archive, remove all "build" directories created by the gmibuildtool.

3. Get an Apple® developer account, device identifiers (UDID) and corresponding identifiers to sign your iOS app (certificate, bundle id, provisioning profile).

**Important:** The UDID is the identifier of your physical device, it can be found with the `instruments -s` command when the device is plugged to the Mac. When deploying on a physical device, make sure that the UDID of the device is listed in the Apple® Developer account that is used to generate the provisioning profiles.

4. Xcode® must be installed on your Mac® OS X computer (utilities from Xcode® toolchain are required).

**Note:** Make sure that the installed Xcode® version supports the iOS versions of your mobile devices. As a general rule, update the Xcode® and iOS to the latest versions.

5. iOS app resources such as icons and launch images (in all required sizes).

### Finding the UDID of the plugged device

In order to find the UDID of the device plugged to your Mac, execute the `instruments -s` command, and identify the line describing your physical device:

```bash
$ instruments -s
Known Devices:
  fraise [55D6D6C1-DE87-52F0-865E-3C6DC79F13D7]
  Fourjs2 iPod touch (9.1) [78b7452fa9462c98c3bc7047da344314fd032004]
  iPad 2 (9.0) [19CDA827-CA55-46F1-9376-BF61E2ECEFDDBB]
  iPad Air (9.0) [F55E1207-C42B-472E-BD76-5B5AE46DE77A]
  iPad Air 2 (9.0) [A0E8C4C4-67CD-42CB-84DF-9C75AC773293]
...
Known Templates:
  "Activity Monitor"
  "Allocations"
  ...
```

In the above output, the UDID of the iPod® is 78b7452fa9462c98c3bc7047da344314fd032004.

### Environment settings

Before starting the command-line buildtool, make sure that Xcode® tools are available. Try `xcodebuild -version` from the command line:

```bash
$ xcodebuild -version
Xcode 11.3.1
Build version 11C504
```

### The gmibuildtool

The gmibuildtool command line tool can build IPA packages of iOS apps written in Genero.

In order to identify the exact product version number of each GMI component, use the `--version` option of the gmibuildtool:

```bash
$ gmibuildtool --version
GMI version:1.40.04
VM version:3.20.05
GWS version:3.20.05
```
Cordova ver:gm_1.30.14

**Manage GMI plugins**

In order to get the list of plugins installed in the GMI environment, use the `--list-plugins` option:

```
$ gmibuildtool --list-plugins
...
cordova-plugin-device-motion
cordova-plugin-media
...
```

To install additional plugins in the GMI installation directory, use the `--install-plugins` option, for example:

```
gmibuildtool --install-plugins path-to-plugin-sources
```

For more usage examples, see Cordova plugins on page 3644.

**Creating the GMI front-end for development purpose**

A self-made GMI front-end can be created with the `gmibuildtool` command. For more details, see Mobile development mode on page 3568.

**Specifying the target to build and deploy the iOS app**

The `gmibuildtool` command can build and install iOS apps for the simulator or for physical devices.

The build and/or install action is controlled by the `--device` option:

- By default, when not specifying the `--device` option, a GMI.app directory is created for the simulator.
- When specifying the `--device booted` option, the GMI.app directory is created and the app is installed on the booted simulator.
- When specifying the `--device phone` option, the GMI.app directory and .ipa file are created.
- When specifying the `--device physical-device-name` option (with a real physical device name plugged on your Mac), the GMI.app directory and .ipa file are created and the app is installed on the device.

By default, the generated GMI.app directory and .ipa archive can be found in $PWD/build sub-directories; however, you can specify the destination IPA file with the `--output` option.

**Specifying the GBC for Universal Rendering**

When your app uses Universal Rendering mode, use the `--gbc` option to specify which GBC has to be bundled in the iOS app package:

```
gmibuildtool ... --gbc gbc-archive ...
```

The `gbc-archive` parameter is the ZIP archive of the GBC front-end, to be created as described in the Create a runtime zip topic in the Genero Browser Client User Guide.

If this option is used, the embedded applications will implicitly be displayed with Universal Rendering. There is no need to set the `gui.rendering` option in the FGLPROFILE; however, if `gui.rendering="native"` is set, the native GUI mode will be used, even if the GBC option is specified during the build.

**Elements used to build the iOS app**

The `gmibuildtool` command builds the iOS app package from the following:

- The GMI binary archive, containing the GMI front-end and the FGLGWS runtime system library,

  **Note:** These files are provided in the `fjs-fglgmi-*`.zip archive.
Important: When re-installing a new GMI archive, remove all "build" directories created by the gmibuildtool.

- The GBC to be used for Universal Rendering (--gbc option),
- The compiled application program and resource files (.42m, .42f, etc),

Note: The application program files must include a main.42m or main.42r module.

- The display name of the app (--app-name option),
- The version of the app (--app-version option),
- The debug or release mode (--mode option),
- The certificate (to sign the app) (--certificate option),
- The bundle Identifier (--bundle-id option),
- The app provisioning profile (.mobileprovision file) (--provisioning option),
- iOS app specific resources:
  - App icons (--icons option),
  - Launch images (--launch-images option) or launch storyboard file (--storyboard option).

For a complete description of command options, see gmibuildtool on page 2102.

Default build directory structure

For convenience, the buildtool supports a default directory structure to find all files required to build the app:

```
top-dir
|-- main.42m and other program files, as described in Directory structure for GMI apps on page 3604
|-- gmi
    |-- Info.plist
    |  |-- LaunchScreen.storyboard or (default launch images)
    |  |    |-- Default@2x.png
    |    |-- Default-568h@2x.png
    |    |-- Default-Landscape.png
    |    |-- Default-Landscape-667h@2x.png
    |    |-- Default-Landscape-736h@3x.png
    |    |-- Default-Landscape@2x.png
    |    |-- Default-Portrait.png
    |    |-- Default-Portrait-736h@3x.png
    |    |-- Default-Portrait-667h@2x.png
    |    |-- Default-Portrait@2x.png
    |    |    ...
    |    |-- icon_29x29.png
    |    |-- icon_40x40.png
    |    |-- icon_57x57.png
    |    |-- icon_58x58.png
    |    |-- icon_72x72.png
    |    |-- icon_76x76.png
    |    |-- icon_80x80.png
    |    |-- icon_120x120.png
    |    |-- icon_152x152.png
    |    |    ...
```

In the above directory structure:

1. `top-dir` is the top directory of the default structure. It will typically hold your application program files. A different program files directory can be specified with the `--program-files` option.
2. `top-dir/gmi` is the default directory containing the app resource files such as icons:
a. **Info.plist** is the Information Property List File that will be used to build the app. Some properties will be overwritten by **gmibuildtool** options like --app-name and --app-version.

b. Provide either a launch screen storyboard or default launch images:

- **LaunchScreen.storyboard** is the default storyboard file for the app launch screen. This file can be specified with the **gmibuildtool --storyboard** option.
- **Default-*.*.png** are the app launch image files. The directory to find launch images can be specified with the **gmibuildtool --launch-images** option.

c. **icon_*.png** are the app icon files. The directory to find icons can be specified with the **gmibuildtool --icons** option.

**Debug and release versions**

iOS apps can be generated in a debug or release version. Release version are prepared for distribution on the App Store, while debug versions are used in development.

In debug mode, the app installed on the device can listen on the debug TCP port to allow **fgldb -m** connections, after enabling the debug port in the app settings.

Debug or release mode must be specified in the command line with the **--mode debug** or **--mode release** option. Additionally, if you want to deploy on a physical device, you need to use a provisioning profile corresponding to the debug or release mode:

- In debug mode, the certificate must be a development certificate.
- In release mode, the certificate must be a distribution certificate.

**Defining the app version and build number**

Apple distinguishes the app version number of a bundle (visible to the end user), from the build version number of a bundle (called a release version number in Apple docs).

You specify the app version number with the **--app-version** option of the **gmibuildtool** command. This option sets the **CFBundleVersion** property of the **Info.plist** file, and must match the version specified in iTunes® Connect.

In order to distinguish multiple builds (Apple's term is "releases") of the same app version number, define the build version number of your app with the **--build-number** option. This option sets the **CFBundleShortVersionString** property of the **Info.plist** file. For a given app version, you need to increase this build number, to be able to upload a new binary on iTunes® Connect.

**Note:** If you do not specify the **--build-number** option, the build version number defaults to the app version specified with the **--app-version** option.

**Defining app properties in the ./gmi/Info.plist file**

iOS app are created with a set of properties that are essential configuration information for a bundled executable. These properties are defined in the "Information Property List File", an XML formatted file named **Info.plist** by convention.

Most important **Info.plist** properties are defined with **gmibuildtool** options such as --app-name and --app-version. However, you may need to define other properties that are out of the scope of the buildtool. For example: background modes, device capabilities, screen orientations, permanent wifi, and so on.

In order to define specific app properties, create an **Info.plist** file in the **top-dir/gmi** directory before executing the **gmibuildtool**. Properties covered by the buildtool will be overwritten, while any other property defined in the **top-dir/gmi/Info.plist** file will be left untouched.

For more details about the **Info.plist** file structure, see Apple developer site page about Information Property List File.
Building an iOS app with gmibuildtool

Follow the next steps to setup a GMI app build directory in order to create an iOS app, based on the default directory structure:

1. Create the root distribution directory (`top-dir`)
2. Copy compiled program files (.42m, .42f, fgprofilie, application images, web component files, etc) under `top-dir`.
3. Copy the default English .42s compiled string resource file under `top-dir`.
4. Create non-English language directories (fr, ge, ...) under `top-dir` and copy the corresponding .42s files.
5. Copy default application data files (database file for ex) under `top-dir`.
6. Create the `top-dir/gmi` directory.
7. Copy iOS app resources (icons, launch screen, storyboard) under `top-dir/gmi`.
8. If needed, create an `top-dir/gmi/Info.plist` file, to define specific iOS app properties.

Once the build directory is prepared, issue the following commands:

```bash
$ cd top-dir
$ gmibuildtool \
   --output myapp.ipa \
   --app-name "My App" \
   --app-version "v3.1.6" \
   --bundle-id "com.example.mycompany.myapp" \
   --mode release \
   --certificate "iPhone Developer" \
   --provisioning "~/Downloads/myapp.mobileprovision" \
   --device phone
```

Building a GMI app with C extensions or custom front calls

In order to create an iOS app using C extensions written in Objective-C as in Implementing C-Extensions for GMI on page 2247, proceed as follows (the same technique can be used to build apps that include custom front calls):

1. Build a static library from your Objective-C sources, by using the `staticlib` target of the `$GMIDIR/lib/Makefile-gmi` generic makefile (specify the library file name with the `USER_LIBNAME` variable). The `staticlib` makefile target will produce a `.a` library file, by using all .m and .c files found in the current directory. For example:

   ```bash
   $ make -f $GMIDIR/lib/Makefile-gmi USER_LIBNAME=mylib.a staticlib
   ```

2. When building the app, specify the additional libraries with the `--extension-libs` option. For example:

   ```bash
   $ gmibuildtool ... --extension-libs "-lz libBPush.a mylib.a" ...
   ```

Regardless of whether the `--extension-libs` option is used, the gmibuildtool looks to see if `./gmi/*.a` exists. If they exist, it adds these static libs to the link list.

If a file called `./gmi/link_flags.sh` exists, this file is read by gmibuildtool to set additional link flags. For example:

```bash
#set LINK_FLAGS to add additional system libs or frameworks
LINK_FLAGS="-bz2 -framework MapKit"
```

In most cases the *.a files will be sufficient, for system frameworks like MapKit, as long as the following hint is specified in your .m source file:

```c
/* this avoids using -framework MapKit and instructs the linker to link with MapKit*/
@ import MapKit;
```
Only pure C libs such as `libbz2.a` actually need the `link_flags.sh` file, if no `--extension-libs` option is used and one of the static extension libs needs a system C library.

For complete examples, see `$GMIDIR/demo/MobileDemo/userextension` and `$GMIDIR/demo/MobileDemo/userfrontcall`

**Related concepts**

*Mobile development mode* on page 3568
Set up a development environment to display app forms on a mobile front-end.

**gmbuildtool**
The `gmbuildtool` is a utility to create and test applications for an iOS devices.

**Syntax**

```
gmbuildtool [options]
```

1. *options* are described in *Options* on page 3612.

**Options**

**Table 759: gmbuildtool options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--app-name application-name</code></td>
<td>Display name of the mobile app. This option can be specified to define the display name of the app, it sets the <code>CFBundleDisplayName</code> property in the <code>Info.plist</code> file. If not specified, the name defaults to &quot;Noname&quot;.</td>
</tr>
<tr>
<td><code>--app-version application-version</code></td>
<td>Defines app version visible to the users on the App Store. This option is mandatory and sets <code>CFBundleVersion</code> properties in the <code>Info.plist</code> file. <strong>Note:</strong> If the <code>--build-number</code> option is not used, <code>--app-version</code> will also set the both the <code>CFBundleShortVersionString</code> property. In iTunes® Connect, you define the version of your app, that must match the <code>CFBundleVersion</code> property in the <code>Info.plist</code> file of the app. If these versions do not match, the app cannot be published. Once the app is visible on App Store, the version specified in iTunes® Connect shows up in the &quot;Version&quot; section of the application page. The recommendation for the app version number is that it is a string comprised of three period-separated integers. For example: &quot;1.4.2&quot;</td>
</tr>
<tr>
<td><code>--bundle-id bundle-identifier</code></td>
<td>Defines the Bundle Identifier (a.k.a. App Id) for the app. This option is mandatory and sets the <code>CFBundleIdentifier</code> property in the <code>Info.plist</code> file. A bundle identifier is the unique identifier of your app, to let iOS recognize new app versions. When developing for the simulator, you can choose your own identifier. When creating an application for the App Store, the bundle identifier must be registered with Apple. If not specified, the name defaults to &quot;noname&quot; (for prototyping).</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>--build-cordova cordova-plugin-names</code></td>
<td>Defines Cordova plugins to be embedded in the app package. When specifying multiple cordova plugins, use the comma (,) as separator. The name of the plugin must match the Git repository name. It is case-sensitive. <strong>Note:</strong> To get the list of available Cordova plugins, use the <code>gmibuildtool --list-plugins</code> command.</td>
</tr>
<tr>
<td><code>--build-number build-number</code></td>
<td>Defines the build number used to upload a new binary of the same app version. This option must be used to distinguish different builds for the same app version. It sets the <code>CFBundleShortVersionString</code> property in the <code>Info.plist</code> file. The build number needs to be incremented in order to upload a new binary version of the same app version in iTunes Connect. If this option is not used, the build number defaults to the version specified with the <code>--app-version</code> option. The build number is a string comprised of three period-separated integers. For example: &quot;1.4.2&quot;</td>
</tr>
<tr>
<td><code>--certificate identity</code></td>
<td>Name of a certificate to sign the app. This option is mandatory to build apps for a physical device or for the app store. The certificate can be found in the Keychain® access program, in the &quot;Common Name&quot; field of the certificate panel. The command <code>security find-identity -v</code> can be used to list all available certificates.</td>
</tr>
<tr>
<td>`--crypto {yes</td>
<td>no}`</td>
</tr>
</tbody>
</table>
| `--device device-name` | Defines the name of a device or simulator.  
- By default, when not specifying the `--device` option, a `GMI.app` directory is created for the simulator.
- When specifying the `--device booted` option, the `GMI.app` directory is created and the app is installed on the booted simulator.
- When specifying the `--device phone` option, the `GMI.app` directory and `.ipa` file are created.
- When specifying the `--device physical-device-name` option (with a real physical device name plugged on your Mac), the `GMI.app` directory and `.ipa` file are created and the app is installed on the device. **Note:** Use the `instruments -s Xcode®` command to find the list of available devices (simulators or connected devices). |
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--extension-libs</td>
<td>Specify the libraries to use when compiling and linking the app. This option is used when you want to provide your own C extension or custom front calls.</td>
</tr>
<tr>
<td>--help</td>
<td>Displays options for the tool.</td>
</tr>
<tr>
<td>--gbc gbc-archive</td>
<td>Defines the GBC to be used for Universal Rendering.</td>
</tr>
<tr>
<td>--icons icons-dir</td>
<td>Provides the directory where the application icons are located. By default, the application icons directory is current-working-dir/gmi.</td>
</tr>
<tr>
<td>--install [yes</td>
<td>no]</td>
</tr>
<tr>
<td>--install-plugins github-url</td>
<td>This option installs additional plugins in the GMI installation directory.</td>
</tr>
<tr>
<td>--launch-images launch-images-dir</td>
<td>The directory where launch images are located. By default, the launch images directory is current-working-dir/gmi.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> This option is ignored if the --storyboard option is provided.</td>
</tr>
<tr>
<td></td>
<td>The name of the image files must be: Default.png, <a href="mailto:Default@2x.png">Default@2x.png</a> <a href="mailto:Default-568h@2x.png">Default-568h@2x.png</a>, <a href="mailto:Default-Landscape-667h@2x.png">Default-Landscape-667h@2x.png</a>, <a href="mailto:Default-Portrait-667h@2x.png">Default-Portrait-667h@2x.png</a>, <a href="mailto:Default-Landscape-736h@3x.png">Default-Landscape-736h@3x.png</a>, <a href="mailto:Default-Portrait-736h@3x.png">Default-Portrait-736h@3x.png</a>, Default-Landscape.png <a href="mailto:Default-Portrait@2x.png">Default-Portrait@2x.png</a>, <a href="mailto:Default-Landscape@2x.png">Default-Landscape@2x.png</a>. Each file name corresponds to a device type (you may not need to provide all files if you target only recent iOS devices), see Apple® Developer documentation for more details about launch images.</td>
</tr>
<tr>
<td>--list-plugins</td>
<td>This option lists the shipped plugins and additional plugins installed in the GMI installation directory.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>--mode (debug|release)</strong></td>
<td>Controls the debug or release mode for the app. &lt;br&gt;By default, the mode is debug.  &lt;br&gt;Note that the provisioning profile must correspond:  &lt;br&gt;- <strong>--mode debug</strong>: Development provisioning profile.  &lt;br&gt;- <strong>--mode release</strong>: Distribution provisioning profile.</td>
</tr>
<tr>
<td><strong>--output (ipa-file-name)</strong></td>
<td>Path to output IPA and APP files to be generated. &lt;br&gt;By default, a &quot;build&quot; directory is created, with subdirectories containing the .ipa and .app files. &lt;br&gt;An IPA file is created when building an application for a physical device and the App Store. The IPA file is not needed and will not be created when building for the simulator.</td>
</tr>
<tr>
<td><strong>--program-files (program-dir)</strong></td>
<td>Path to Genero BDL program files (.42m, .42f, etc). &lt;br&gt;By default, the program files directory is the current work directory. &lt;br&gt;Following files are automatically excluded: *.4gl, *.per, *.msg, *.str, *.sch, [Mm]akefile, *.42d, [Mm]akefile, *. [chdmo], *.xib, build/ (the build directory), gmi/ (this folder is the default location of LaunchScreens and AppIcons). &lt;br&gt;If the file gmiignore exists, then this file contains additional files to be ignored.</td>
</tr>
<tr>
<td><strong>--provisioning (provisioning-file)</strong></td>
<td>Path to the provisioning profile (.mobileprovision). &lt;br&gt;The provisioning profile is mandatory to build apps for a physical device or for the app store. &lt;br&gt;Provisioning profiles can be found in $HOME/Library/MobileDevice/Provisioning\ Profiles/</td>
</tr>
<tr>
<td><strong>--storyboard (storyboard-file)</strong></td>
<td>Path to the storyboard file, to get a splash screen to be displayed when the app starts. &lt;br&gt;This file is an alternative for Launch Screens (--launch-images option). This option is mandatory if you do not provide launch images with the <strong>--launch-images</strong> option. &lt;br&gt;The default storyboard is showing an empty navigation bar and an empty toolbar. If the storyboard references images, gmibuildtool searches for the images in the same directory the storyboard is in, and bundles the images with the application.</td>
</tr>
<tr>
<td><strong>--verbose (yes|no)</strong></td>
<td>Enable the verbose mode.</td>
</tr>
<tr>
<td><strong>--version</strong></td>
<td>Displays version information.</td>
</tr>
</tbody>
</table>

**Related concepts**

Building iOS apps with Genero on page 3606
Genero provides a command-line tool to build applications for iOS devices.

**Running mobile apps on an application server**

From the mobile device, programs can be started remotely on an application server, and displayed on the device.

**Purpose of remote application execution for mobile devices**

Remote applications displayed on a mobile device allow the use of the processor, memory, storage and software resources available on a server, for mobile users.

**Note:** Executing remote/server applications for display on a mobile device requires a reliable and constant network connection. If the network connection fails, the application will stop, as with other client/server Genero front-ends.

Server applications can only be started through the Genero Application Server (GAS), by using the UA protocol available since version 3.00. You must set up and configure the GAS for the programs you want to start remotely. See the GAS documentation for more details.

**Note:** Applications executed on the GAS server must use the UTF-8 encoding. Mobile front-ends will reject any attempt to display forms of an application using an encoding other than UTF-8.

**Implementing the embedded mobile app**

Create a small application to be deployed on the mobile device, which then starts the application(s) on an GAS server.

The server application is started from the embedded application through the `runOnServer` front call. The embedded mobile application can be a very simple `MAIN / END MAIN` program, only performing the "runOnServer" front call.

For example, this is the very minimal embedded application, starting a program on the GAS:

```plaintext
MAIN
   CALL ui.interface.frontcall("mobile","runOnServer",
      ["http://myappserver:6394/ua/r/myapp"],[])
END MAIN
```

When the remote application starts, the graphical user interface displays on the mobile device.

The `runOnServer` front call returns when the called application ends, control goes back to the initial application executing on the mobile device.

**Note:** In development context, it is possible to execute the parent starter app on a server, display on a mobile device with `FLGSERVER` on page 280 set properly, and use the `runOnServer` front call. Because starting remote GAS applications is done with a front call, this configuration mimics an embedded starter app running on the device.

**Using the runOnServer front call**

The application executed on the server-side is identified by the first parameter of the `runOnServer` front call. This application must be delivered by the Genero Application Server. The parameter must contain an "ua/r" URL syntax (the UA protocol introduced with the GAS 3.00).

For example: `http://myappserver:6394/ua/r/myapp`

The URL may contain a query string, with parameters for the application to be executed by the GAS.

If needed, you can add a second argument to define a timeout as a number of seconds. The embedded application will wait for the remote application to start, until the timeout expired. If no timeout parameter is specified, or when zero is passed, the timeout is infinite.

In case of failure (application not found, timeout expired), the front call raises the runtime error -6333 and the HTTP status code of the request can be found in the error message details. Use a `TRY/CATCH` block to check if the execution the server application was successful:

```plaintext
MAIN
```
TRY
   CALL ui.interface.frontcall("mobile","runOnServer",
   ["http://myappserver:6394/ua/r/myapp"],[[]])
CATCH
   ERROR err_get(STATUS)
END TRY
END MAIN

Subsequent server-side application runs are allowed; the last active application will display on the device. However, it is not possible to navigate between started applications. Therefore, an application started with the runOnServer front call must only use the RUN instruction to start sub-programs. RUN WITHOUT WAITING is not supported.

**Passing parameters to the server application**

If needed, the embedded app can pass arguments to the server application by using parameter specification in the URL string, with the ?Arg=value1&Arg=value1&... notation:

```
DEFINE params, base, complete_url STRING
LET params = "Arg=verbose&Arg=5677"
LET url = "http://myappserver:6394/ua/r/myapp"
LET complete_url = base || "?" || params
```

The remote program can retrieve the parameters with the arg_val() built-in function.

**Note:** It is not needed to URL-encode the string passed to the runOnServer front call.

See the GAS documentation (AllowUrlParameters attribute) about passing parameters in the application URL.

This is an example of an embedded application to be deployed on the mobile device, which passes parameters to a server-side application:

```
IMPORT util
MAIN
   DEFINE arr DYNAMIC ARRAY OF STRING, x INT
   MENU "test"
   COMMAND "runOnServer"
      CALL arr.clear()
      LET arr[1] = "first argument"
      LET arr[2] = "second argument"
      LET x = do_run("http://10.0.40.29:6394/ua/r/test1", 10, arr)
   COMMAND "exit"
   EXIT MENU
END MENU
END MAIN

FUNCTION do_run(url,timeout,params)
   DEFINE url STRING,
      timeout SMALLINT,
      params DYNAMIC ARRAY OF STRING
   DEFINE i, r INTEGER, tmp STRING
   LET r = 0
   LET tmp = url
   FOR i=1 TO params.getLength()
      LET tmp = tmp || IIF(i==1,"?","&") || "Arg=" || params[i]
   END FOR
   TRY
      CALL ui.interface.frontcall("mobile","runOnServer",[tmp,timeout],[[]])
   CATCH
      ERROR err_get(STATUS)
      LET r = -1
   END TRY
   RETURN r
```
A sample server-side application:

```plaintext
MAIN
  MENU "Prog1"
    COMMAND "arg1" MESSAGE "Arg 1 = ", arg_val(1)
    COMMAND "arg2" MESSAGE "Arg 2 = ", arg_val(2)
    COMMAND "arg3" MESSAGE "Arg 3 = ", arg_val(3)
    COMMAND "Quit" EXIT MENU
  END MENU
END MAIN
```

Sharing files between embedded and server app

If files need to be shared between the embedded application and the server application, the application running on the GAS can only access the `data-directory` directory, in the sandbox of the embedded application that executes the "runOnServer" front call.

This matters when using file handling APIs such as `fgl_putfile()` and `fgl_getfile()` or front calls like `takePhoto` and `launchURL`.

The `data-directory` on the mobile device can be found with the `feInfo/dataDirectory` front call. In both the embedded app and the app running on the server, this front call will return the same directory.

The following workflow can be used:

1. Before starting the server application with a `runOnServer` front call, the embedded app must copy files to the `data-directory`.
2. While executing, the server application can retrieve files from the `data-directory` with `fgl_getfile()`, and send its own files to the `data-directory`, with `fgl_putfile()`.
3. When the server application terminates, the embedded app can read files the server application left in the `data-directory`.

**Note:** If several remote applications are started successively on the server with a `RUN` instruction, make sure to not overwrite files written by other server programs.

In order to write code for the embedded app, that can be executed in development mode (running on a server) and on the mobile device, you can adapt to the execution context: Make a simple file copy when executing on the mobile device, or do an `fgl_putfile()` call, when running on the development server. Check the execution context with the `base.Application.isMobile()` method.

This example, in the embedded app on the mobile device, copies a file from the device private directory to the `data-directory`:

```plaintext
IMPORT os
...
  CALL mobile_copy_to_data_dir("myfile.txt")
...
FUNCTION mobile_copy_to_data_dir(fn)
  DEFINE fn, dd, dst STRING, r INT
  CALL ui.interface.frontcall("standard","feInfo",["dataDirectory"],[dd])
    -- Always use / as path sep for Android/iOS dirs.
  LET dst = dd || "/" || os.Path.basename(fn)
  IF base.Application.isMobile() THEN
    -- Executing on device: make a simple copy to data-dir
    LET r = os.Path.copy(fn, dst)
    MESSAGE SFMT("COPY status = %1", r)
  ELSE
    -- Executing on dev server: make a file transfer to data-dir
    CALL fgl_putfile(fn, dst)
  END IF
```
Note: We do not use the `os.Path.join()` method here because it would add the path separator for the operating system where the application is executed. This would not be a problem when executing on the mobile device or Unix-like platforms. However, when running on a Windows® platform, the `os.Path.join()` method would join the directory and the file name with a backslash, and the resulting path would not fit Android™ or iOS directory path specification for the `data-directory`.

In the server application, use the `fgl_getfile()` function, to transfer a file from the mobile device `data-directory` to the local server disk:

```python
IMPORT os
...
    CALL server_get_from_data_dir("myfile.txt", "/tmp/server_file.txt")
...
FUNCTION server_get_from_data_dir(fn, dst)
    DEFINE fn, dst, dd, src STRING
    CALL ui.interface.frontcall("standard","feInfo",['dataDirectory'],[dd])
    -- Use / as path sep for Android/iOS dirs!
    LET src = dd || "/" || fn
    CALL fgl_getfile(src, dst)
END FUNCTION

```

Similarly, in the server application, use the `fgl_putfile()` function, to copy a file from the server application to the `data-directory` of the embedded app:

```python
IMPORT os
...
    CALL server_put_to_data_dir("/tmp/server_file.txt", "myfile.txt")
...
FUNCTION server_put_to_data_dir(src, fn)
    DEFINE src, fn, dd, dst STRING
    CALL ui.interface.frontcall("standard","feInfo",['dataDirectory'],[dd])
    -- Use / as path sep for Android/iOS dirs!
    LET dst = dd || "/" || fn
    CALL fgl_putfile(src, dst)
END FUNCTION

```

**Related concepts**

**feInfo** on page 2651
Queries general front-end properties.

**runOnServer** on page 2710
Run an application from the Genero Application Server using the specified URL.

**Debugging on a mobile device** on page 2145
It is possible to remotely start the debugger for an app running on a mobile device.

**Push notifications**

This section describes how to implement push notification with Genero.

A push notification is a short message sent by a central server entity to an app installed on a mobile device. In order to be notified, the app+device must register itself to a push service (a global service such as Firebase Cloud Messaging), and register also to a push provider (part of the custom application). To indicate that fresh information is available, notifications are sent by push providers to the push service, which broadcasts notifications to registered devices. The apps can then get details about the notification and display a little hint to the end user. Enterprise mobile applications can use push notifications to produce urgent and important updates for users.
Figure 153: This figure describes the workflow for a push notification (items in yellow are the components that can be implemented with Genero BDL)

Workflow:
1. The app registers to the push service.
2. The push service generates a unique token to identify the device+app and returns this token to the app.
3. The app transmits the token to the token maintainer.
4. The token maintainer stores the new token in a database.
5. Some event occurs in the global application workflow that requires a push notification to warn all registered devices/apps.
6. The push provider reads the database for registered tokens.
7. The push provider sends push notification requests to the push service.
8. The push service broadcasts the notification messages to all registered devices.

There are several push notification mechanisms available. This chapter covers the Firebase Cloud Messaging (FCM) and Apple Push Notification services (APNs).

Common components can be implemented on the same code base for both FCM and APNs push notification mechanisms: The mobile app and the token maintainer.

The complete source code is available of the FourJSGenero GitHub: https://github.com/FourjsGenero/ex_push_notification
Firebase Cloud Messaging (FCM)

Follow this procedure to implement push notification with FCM.

Introduction to FCM push notification

The push notification solution described in this section is based on the Firebase Cloud Messaging service. Familiarize yourself with FCM by visiting the Firebase Cloud Messaging web site.

Firebase Cloud Messaging services allow push servers to send notification message data to registered Android™ or iOS devices.

The system involves the following actors:

- The Firebase Cloud Messaging service (FCM):
  
  FCM provides push server and client identification. It also handles all aspects of queuing of messages and delivery to the target application running on registered devices.

- The registration tokens maintainer:
  
  A Web services server program maintaining the database of registration tokens with application user information. This program must listen to new device registration events and store them in a database. The push server program can then query this database to build the list of registration tokens to identify the devices to be notified.

- The push server program:
  
  Implemented by a third-party service or as a Genero BDL program using the Web services API. This push server program will send notification messages to FCM with two connection servers (HTTP and XMPP).

- Devices running the Genero app registered to the push notification server:
  
  Registered devices use the push notification client API to register, get notifications data and unregister from the service.

Note: The database used to store registration tokens must be a multi-user database (do not use SQLite for example), since two distinct programs will use the database.

Creating a FCM project

To initiate a push notification service dedicated to your applications, you must first create a Firebase Cloud Messaging project in the FCM console. Creating an FCM project will provide you:

- The google-services.json configuration file, to be bundled with your app.
- The "Server Key" is the authentication key to access Google services.

Steps to setup your FCM project:

1. Go to the FCM console and login with your Google developer account.
2. Add a new FCM project.
3. Add an (Android) app to your FCM project: Specify the same package name specified when building your app with the --build-app-package-name option of gmabuildtool.
4. Download the google-services.json configuration file. This file needs to be added to the appdir, beside other program files.
5. Skip other FCM project creation steps.
6. In the project overview page, go to the settings of the new created project.
7. Select the "Cloud Messaging" panel.
8. Copy the "Server Key" and save it to a file. This is the key to be send with the "Authorization:key=server-key" HTTP header when posting a message to the FCM server endpoint fcm.googleapis.com/fcm/send.

For more details about FCM project creation, visit the Firebase Cloud Messaging web site.
**Implementing the registration tokens maintainer**

To handle device registrations on the server side of your application, the same code base can be used for FCM and other token-based frameworks.

For more details, see Implementing a token maintainer on page 3631.

**Implementing the push server**

The push server will produce application notification messages that will be transmitted to the FCM service. The FCM service will then spread them to all mobile devices registered to the service with the Sender ID.

**Important:** The size of an FCM notification content cannot exceed 4 Kilobytes. If more information needs to be passed, after receiving the push message, apps must contact the server part to query for more information. However, this is only possible when network is available.

The push server will use RESTful HTTP POST requests to send notifications through the FCM service to the following URL:

"https://fcm.googleapis.com/fcm/send".

The HTTP POST header must contain the following attributes:

```
Content-Type: application/json
Authorization: key=server-key
```

where *server-key* is the Server Key obtained from the FCM project settings.

The push server program can be implemented with the Web Services API to make RESTful requests as follows:

```java
IMPORT com
IMPORT util

FUNCTION fcm_send_notif_http(server_key, notif_obj)
  DEFINE server_key STRING,
   notif_obj util.JSONObject
  DEFINE req com.HTTPRequest,
   resp com.HTTPResponse,
   req_msg, res STRING
  TRY
    LET req = com.HTTPRequest.Create("https://fcm.googleapis.com/fcm/send")
    CALL req.setHeader("Content-Type", "application/json")
    CALL req.setHeader("Authorization", SFMT("key=%1", server_key))
    CALL req.setMethod("POST")
    LET req_msg = notif_obj.toString()
    IF req_msg.getLength() > 4096 THEN
      LET res = "ERROR : GCM message cannot exceed 4 kilobytes"
      RETURN res
    END IF
    CALL req.doTextRequest(req_msg)
    LET resp = req.getResponse()
    IF resp.getStatusCode() != 200 THEN
      LET res = SFMT("HTTP Error (%1) %2",
                     resp.getStatusCode(),
                     resp.getStatusDescription())
      ELSE
        LET res = "Push notification sent!"
      END IF
    CATCH
      LET res = SFMT("ERROR : %1 (%2)", STATUS, SQLCA.SQLERRM)
    END CATCH
    RETURN res
  END TRY
END FUNCTION
```
The body of the HTTP POST request must be a JSON formatted record using a structure similar to the following example:

```json
{
  "collapse_key": "stock_update",
  "time_to_live": 108,
  "delay_while_idle": true,
  "data": {
    "stock_change": {
      "stock_id": "STK-034",
      "timestamp": "2015-02-24 15:10:34.18345",
      "item_count": 15023
    },
    "registration_ids": [ "APA91b...", "Hun4MxP...", "5ego..." ]
  }
}
```

**Note:** This notification message uses the "registration_ids" attribute to provide a list of devices to be notified. If you want to notify a single device, use the "to" attribute instead of "registration_ids", and pass a single registration token instead of a JSON array.

For more details about the JSON request structure in a FCM HTTP POST, visit the Firebase Cloud Messaging web site.

By convention, if the "data" member of the JSON request defines a "genero_notification" member, the front-end will show graphical notification (pop-up hint) with the "title", "content" and the "icon" values.

**Note:** The recommendation for GMA is that the icon is packaged in the APK and is accessible by name (as the gma_ic_genero.png in the drawable folders).

For example:

```json
...
  "data": {
    "genero_notification": {
      "title": "Stock has changed",
      "content": "New stock information will be retrieved from the backend server...",
      "icon": "stock_update"
    },
    ...
  },
  "registration_ids": [ "APA91b...", "Hun4MxP...", "5ego..." ]
}
```

The next code example implements a function that creates the JSON object, which can be passed to the `fcm_send_notif_http()` function described above. The only purpose of this notification message is to test the "genero_notification" pop-up hint. The function takes an array of registration tokens as a parameter, which will be used to set the "registration_ids" attribute:

```java
FUNCTION fcm_simple_popup_notif(reg_ids, notif_obj, popup_msg, user_data) 
DEFINE reg_ids DYNAMIC ARRAY OF STRING,
    notif_obj util.JSONObject, 
    popup_msg, user_data STRING
DEFINE data_obj, popup_obj util.JSONObject
CALL notif_obj.put("registration_ids", reg_ids)
```
The `fcm_simple_popup_notif()` and `fcm_send_notif_http()` functions can then be used as follows:

```java
import com
import util

main
constant server_key = "xyz..."
define reg_ids dynamic array of string,
    notif_obj util.JSONObject

let reg_ids[1] = "APA91bHun..."
let reg_ids[2] = "B4AA2q7xa..."

let notif_obj = util.JSONObject.create()
call fcm_simple_popup_notif(reg_ids, notif_obj, "This is my message!")
call fcm_send_notif_http(server_key, notif_obj)
end main

In order to use the tokens database maintained by a token maintainer program, your FCM push server can collect registration tokens as shown in the following example:

```java
function fcm_collect_tokens(reg_ids)
define reg_ids dynamic array of string
define rec record
    id integer,
    notification_type varchar(10),
    registration_token varchar(250),
    badge_number integer,
    app_user varchar(50),
    reg_date datetime year to fraction(3)
end record
declare c1 cursor for
    select * from tokens
    where notification_type = "FCM"
call reg_ids.clear()
foreach c1 into rec.*
    call reg_ids.appendElement()
    let reg_ids[reg_ids.getLength()] = rec.registration_token
end foreach
end function

The above function can then be used by another function to send the push message to all registered devices:

```java
function fcm_send_text(server_key, msg_title, user_data)
define server_key, msg_title, user_data string
define reg_ids dynamic array of string,
    notif_obj util.JSONObject,
info_msg STRING
CALL fcm_collect_tokens(reg_ids)
IF reg_ids.getLength() == 0 THEN
    RETURN "No registered devices..."
END IF
LET notif_obj = util.JSONObject.create()
CALL fcm_simple_popup_notif(reg_ids, notif_obj, msg_title, user_data)
LET info_msg = fcm_send_notif_http(server_key, notif_obj)
RETURN info_msg
END FUNCTION

Handle push notifications in mobile apps

To handle push notifications in mobile apps, the same code base can be used for FCM and other token-based frameworks.

For more details see Handling notifications in the mobile app on page 3637.

Related concepts

Apple Push Notification Service (APNs) on page 3625
Follow this procedure to implement push notification with APNs.

Apple Push Notification Service (APNs)

Follow this procedure to implement push notification with APNs.

Introduction to APNs push notification

The push notification solution described in this section is based on the Apple Push Notification Service. Familiarize yourself with APNs by visiting the Apple Push Notification Service web site.

Apple Push Notification service allows push servers to send notification message data to registered iOS (and OS X) devices.

The APNs service transports and routes a remote notification from a given provider to a given device. A notification is a short message built from two pieces of data: the device token and the payload.

Note: Each device needs to be identified by its device token, and the provider must send individual notification messages for each registered device.

The system involves the following actors:

- The Apple Push Notification Service (APNs):
  
  APNs provides push server and client identification. It also handles all aspects of message queuing and delivery to the target applications running on registered devices. The APNs system includes a feedback service that can be queried to check for devices that have unregistered and no longer need to be notified.

- The device tokens maintainer:

  A Web Services server program maintaining the database of device tokens, with application user information. This program must listen to new device registration events, store them in a database, and from time to time query the APNs feedback service to check for unregistrations.

- The push provider:

  This program will send notification messages to the APNs server by using the com.APNS class and TCP request API. The push provider program will query the device token database to know which devices need to be notified.

- Devices running the Genero app registered to the push notification server:

  Registered devices use the push notification client API to register, get notifications data and unregister from the service.

Note: The database used to store device tokens must be a multi-user database (do not use SQLite for example), since two distinct programs will use the database.
**APNs push notification security**

iOS apps must be created with an Apple certificate for development or distribution, linked to an App ID (or Bundle ID) with push notification enabled. The provisioning profile used when building the IPA must be linked to the App ID with push enabled. Certificate, provisioning and bundle id must be specified to the GMI buildtool.

To create the push provider linked to your app, usually you need to create two Apple Push Notification certificates linked to your App ID (you select the App ID when you create a push certificate in the Apple member center): One certification for development and another for distribution. For more details about the push provider certificates, see [APNs SSL/TLS certificate](#) on page 2923.

Check also Apple Push Notification documentation for more details about certificate requirements for push notifications.

**Identifying target devices**

Each APNs client device is identified by a *device token*. A device token is an opaque identifier of a device that APNs gives to the device when an app registers itself for push notification. It enables APNs to locate in a unique manner the device on which the client app is installed. The device shares the device token with the push provider. The push provider must produce notification messages for each device by including the device token in the message structure.

**Important**: The mobile app obtains its device token by registering to the APNs service with the `registerForRemoteNotifications` on page 2708 front call. It is then in charge of sending its device token to the push provider; typically through a RESTful request. The push provider must collect and store the device tokens, as they need to be specified in a push notification message send by the push provider.

**Notification content (payload)**

In a notification message, the *payload* is a JSON-defined property list that specifies how the user of an app on a device is to be alerted.

**Important**: The size of an APNs notification payload cannot exceed 2 Kilobytes. Make sure that the resulting BYTE variable does not exceed this size limitation. If more information needs to be passed, after receiving the push message, apps must contact the server part to query for more information. However, this is only possible when network is available.

The payload must contain a list of "aps" records. Each "aps" record represents a notification message to be displayed as a hint on the device (for example, by adding a badge number to the app icon). The "aps" records can also contain custom data in a separate set of JSON attributes.

In the Genero mobile app, the notification messages are obtained by using the `getRemoteNotifications` on page 2703 front call, after a `notificationpushed` action was detected with an ON ACTION handler.

**Important**: When an iOS app is in background, silent push notifications can occur, but notification message data (i.e. the payload) may not be available. In such case, GMI is able to detect that a notification arrived (i.e. when the app badge number is greater than zero) and raise the `notificationpushed` action, but the `getRemoteNotifications` front call will return no message data (*data* return param is NULL). If such case, implement a fallback mechanism (based on RESTful web services for example), to contact the push notification provider and retrieve the message information.

Example of notification record list (JSON array) returned by the `getRemoteNotifications` front call:

```json
[
  {
    "aps" : {
      "alert" : "My first push",
      "badge" : 1,
      "sound" : "default",
      "content-available" : 1
    }
  },
]```
Badge number handling

With APNs, badge number handling is in charge of the application code: The push provider sends a badge number in the payload records, the app can check the message content, and must communicate with a server component, to indicate that the notification message has been consumed. The server program can then maintain a badge number for each registered device, decrementing the badge number.

In order to set or query the badge number for your app, use the following front calls:

- setBadgeNumber (iOS) on page 2719
- getBadgeNumber (iOS) on page 2719

In this tutorial, badge numbers are stored on the server database. The token maintainer handlers requests from apps to sync the badge number for a given device token, and the push provider program reads the database to set the badge number in the notification payload. When the app consumes messages, it queries and resets the app badge number with the getBadgeNumber/setBadgeNumber front calls, and informs the token maintainer to sync the badge number in the central database.

Communication channels

A provider communicates with Apple Push Notification service over a binary network interface, using a streaming TCP socket design in conjunction with binary content:

- The binary interface of the APNs development environment is available through the URL gateway.sandbox.push.apple.com on port 2195.
- The binary interface of the APNs production environment is available through the URL gateway.push.apple.com on port 2195.
- The binary interface of the APNs feedback service is available through the URL feedback.push.apple.com on port 2196.

For each interface, use TLS (or SSL) to establish a secured communication channel. The SSL/TLS certificate required for these connections is obtained from Apple's Member Center.

To establish a TLS session with APNs, an Entrust Secure CA root certificate must be installed on the provider's server. If the server is running OS X, this root certificate is already in the keychain. On other systems the certificate might not be available.

Creating an APNs certificate for the app

The Apple Push Notification Certificate identifies the push notification service for a given mobile app. This certificate will be created from an App ID (a.k.a. Bundle ID) and is used by the APNs system to dispatch the notification message to the registered devices.

For more details, see APNs SSL/TLS certificate on page 2923.
Implementing the device tokens maintainer

To handle device registrations on the server side of your application, the same code base can be used for APNs and other token-based frameworks.

For more details, see Implementing a token maintainer on page 3631.

Implementing the push provider

The push provider will produce application notification messages that will be transmitted to the APNs service. The APNs service will then spread them to all registered mobile devices, identified by their device token.

To send notification messages, the push provider must build binary messages by using the com.APNS API, provided by the Web Services library, and send TCP message requests over SSL/TLS to the following URLs:

- "tcps://gateway.sandbox.apple.com:2195" (for development)
- "tcps://gateway.push.apple.com:2195" (for production)

**Note:** In order to establish a secure connection to the APNs framework an SSL/TLS certificate needs to be defined in FGLPROFILE, as described in APNs SSL/TLS certificate on page 2923.

To send a notification message, the push provider must know the device tokens of the registered devices/applications.

**Note:** A distinct notification message must be sent for each registered device.

The following example demonstrates how to implement a function to send an APNs notification message. The function takes a device token and a JSON object as parameters. First, build the binary data with the com.APNS.EncodeMessage() method, then POST the data with a com.TCPRequest.doDataRequest() method. In case of success, the TCP request timeout will occur (APNs service only responds immediately in case of error), then use the com.TCPResponse.getDataResponse() method, to get status information. See com.APNS.EncodeMessage() for more details about notification message creation.

```java
IMPORT com
IMPORT security
IMPORT util

FUNCTION apns_send_notif_http(deviceTokenHexa, notif_obj)
    DEFINE deviceTokenHexa STRING,
        notif_obj util.JSONObject
    DEFINE req com.TCPRequest,
        resp com.TCPResponse,
        uuid STRING,
        ecode INTEGER,
        dt DATETIME YEAR TO SECOND,
        exp INTEGER,
        data, err BYTE,
        res STRING
    LOCATE data IN MEMORY
    LOCATE err IN MEMORY
    LET dt = CURRENT + INTERVAL(10) MINUTE TO MINUTE
    LET exp = util.Datetime.toSecondsSinceEpoch(dt)
    TRY
        --LET req = com.TCPRequest.create( "tcps://
gateway.push.apple.com:2195" )
        LET exp = util.Datetime.toSecondsSinceEpoch(dt)
        CALL req.setKeepConnection(true)
        CALL req.setTimeout(2) # Wait 2 seconds for APNs to return error code
        LET uuid = security.RandomGenerator.createRandomString(4)
```
DISPLAY "PUSH MESSAGE: ", deviceTokenHexa, "/", notif_obj.toString()
CALL com.APNS.EncodeMessage(
    data,
    security.HexBinary.ToBase64(deviceTokenHexa),
    notif_obj.toString(),
    uuid,
    exp,
    10
)
IF LENGTH(data) > 2000 THEN
    LET res = "ERROR : APNS payload cannot exceed 2 kilobytes"
    RETURN res
END IF
CALL req.doDataRequest(data)
TRY
    LET resp = req.getResponse()
    CALL resp.getDataResponse(err)
    CALL com.APNS.DecodeError(err) RETURNING uuid, ecode
    LET res = SFMT("APNS result: UUID: %1, Error code: %2", uuid, ecode)
    CATCH
        CASE STATUS
            WHEN -15553 LET res = "Timeout Push sent without error"
            WHEN -15566 LET res = "Operation failed ": SQLCA.SQLERRM
            WHEN -15564 LET res = "Server has shutdown"
            OTHERWISE LET res = "ERROR : ": STATUS
        END CASE
    CATCH
        LET res = SFMT("ERROR : %1 (%2)", STATUS, SQLCA.SQLERRM)
    END TRY
END TRY
RETURN res
END FUNCTION

The next code example implements a function that creates the JSON object defining notification content (payload). That object can be passed to the apns_send_notif_http() function described above:

FUNCTION apns_simple_popup_notif(notif_obj, msg_title, user_data, badge_number)
    DEFINE notif_obj util.JSONObject,
    msg_title, user_data STRING,
    badge_number INTEGER
    DEFINE aps_obj, data_obj util.JSONObject
    LET aps_obj = util.JSONObject.create()
    CALL aps_obj.put("alert", msg_title)
    CALL aps_obj.put("sound", "default")
    CALL aps_obj.put("badge", badge_number)
    CALL aps_obj.put("content-available", 1)
    CALL notif_obj.put("aps", aps_obj)
    LET data_obj = util.JSONObject.create()
    CALL data_obj.put("other_info", user_data)
    CALL notif_obj.put("custom_data", data_obj)
END FUNCTION

The apns_simple_popup_notif() and apns_send_notif_http() functions can then be used as follows:

IMPORT com
IMPORT util

MAIN

DEFINE reg_ids DYNAMIC ARRAY OF STRING,
    notif_obj util.JSONObject,
    i INTEGER

LET notif_obj = util.JSONObject.create()
CALL gcm_simple_popup_notif(notif_obj, "This is my message!", 1)

LET reg_ids[1] = "APA91bHun..."
LET reg_ids[2] = "B4AA2q7xa..."
...
FOR i=1 TO reg_ids.getLength()
    DISPLAY gcm_send_notif_http(reg_ids[i], notif_obj)
END FOR

END MAIN

In order to use the tokens database maintained by a token maintainer program, your APNs push provider can collect device tokens as shown in the example below. Note that the dynamic array contains token ids and badge numbers:

FUNCTION apns_collect_tokens(reg_ids)

DEFINE reg_ids DYNAMIC ARRAY OF RECORD
    token STRING,
    badge INTEGER
END RECORD

DEFINE rec RECORD
    id INTEGER,
    notification_type VARCHAR(10),
    registration_token VARCHAR(250),
    badge_number INTEGER,
    app_user VARCHAR(50),
    reg_date DATETIME YEAR TO FRACTION(3)
END RECORD,

x INTEGER
DECLARE c1 CURSOR FOR
    SELECT * FROM tokens
WHERE notification_type = "APNS"
CALL reg_ids.clear() 
FOREACH c1 INTO rec.*
    LET x = reg_ids.getLength() + 1
    LET reg_ids[x].token = rec.registration_token
    LET reg_ids[x].badge = rec.badge_number
END FOREACH
END FUNCTION

In order to handle badge numbers for each registered device, implement a function to update badge numbers in database:

FUNCTION save_badge_number(token, badge)

DEFINE token STRING,
    badge INT
UPDATE tokens SET
    badge_number = badge
WHERE registration_token = token
END FUNCTION

The above functions can then be used to send a push message to all registered devices:

FUNCTION apns_send_message(msg_title, user_data)

DEFINE msg_title, user_data STRING
DEFINE reg_ids DYNAMIC ARRAY OF RECORD
    token STRING,
    badge INTEGER
  END RECORD,
  notif_obj util.JSONObject,
  info_msg STRING,
  new_badge, i INTEGER
CALL apns_collect_tokens(reg_ids)
IF reg_ids.getLength() == 0 THEN
  RETURN "No registered devices..."
END IF
LET info_msg = "Send:
FOR i=1 TO reg_ids.getLength()
  LET new_badge = reg_ids[i].badge + 1
  CALL save_badge_number(reg_ids[i].token, new_badge)
  LET notif_obj = util.JSONObject.create()
  CALL apns_simple_popup_notif(notif_obj, msg_title, user_data, new_badge)
  LET info_msg = info_msg, "\n",
    apns_send_notif_http(reg_ids[i].token, notif_obj)
END FOR
RETURN info_msg
END FUNCTION

Handle push notifications in mobile apps
To handle push notifications in mobile apps, the same code base can be used for APNs and other token-based frameworks.

For more details see Handling notifications in the mobile app on page 3637.

Related concepts
Firebase Cloud Messaging (FCM) on page 3621
Follow this procedure to implement push notification with FCM.

Implementing a token maintainer
The token maintainer is a BDL Web Services server program that handles push token registration from mobile apps.

Basics
In order to implement a push notification mechanism, you need to set up a server part (token maintainer and push notification server), based on a push notification framework such as Firebase Cloud Messaging (FCM) or Apple Push Notification service (APNs). In addition, you need to handle notification events in your mobile app. This section describes how to implement the token maintainer, the server program that maintains the list of registered devices (i.e. registration tokens for FCM or device tokens for APNs).

Note: The max length of a push client token can vary depending on the push framework provider. If you need to store registration tokens in a database, check the max size for a token and consider using a large column type such as VARCHAR(250).

The same code base can be used for Android™ (using FCM) and iOS (using APNs) applications: The token maintainer will basically handle RESTful HTTP requests coming from the internet for token registration and token unregistration. For each of these requests, the program will insert a new record or delete an existing record in a dedicated database table.

Note: The database used to store tokens must be created before starting the token maintainer program. By default, the push demo program uses SQLite (dbmsqt) and the name of the database is "tokendb". To create this SQLite database, simply create an empty file with this name.

The push provider/server program can then query the tokens table to build the list of target devices for push notifications.
In the context of APNs, the token maintainer must also handle badge numbers for each registered device: When consuming notification messages, the iOS app must inform the token maintainer that the badge number has changed. This function is implemented with the "badge_number" command.

The token maintainer is a Web Services server program which must be deployed behind a GAS to handle load balancing. You can, however, write code to test your program in development without a GAS.

The act of registering/unregistering push tokens is application specific: When registering tokens, you typically want to add application user information. Genero BDL allows you to implement a token maintainer in a simple way.

**Note:** When executing this token maintainer program with APNs, you must pass the "APNS" command line argument to execute APNs feedback queries.

**MAIN block and database creation**

Start with the MAIN block, and the connection to a database. In this tutorial, we use SQLite as the database. The program will automatically create the database file and the tokens table if it does not yet exist.

```plaintext
IMPORT util
IMPORT com
IMPORT os

CONSTANT DEFAULT_PORT = 9999

MAIN
   CALL open_create_db()
   CALL handle_registrations()
END MAIN

FUNCTION open_create_db()
   DEFINE dbsrc VARCHAR(100), x INTEGER
   IF NOT os.Path.exists("tokendb") THEN
      CALL create_empty_file("tokendb")
   END IF
   LET dbsrc = "tokendb+driver='dbmsqt'"
   CONNECT TO dbsrc WHENEVER ERROR CONTINUE
   SELECT COUNT(*) INTO x FROM tokens WHENEVER ERROR STOP
   IF SQLCA.SQLCODE<0 THEN
      CREATE TABLE tokens ( id INTEGER NOT NULL PRIMARY KEY,
                          notification_type VARCHAR(10) NOT NULL,
                          registration_token VARCHAR(250) NOT NULL UNIQUE,
                          badge_number INTEGER NOT NULL,
                          app_user VARCHAR(50) NOT NULL, -- UNIQUE
                          reg_date DATETIME YEAR TO FRACTION(3) NOT NULL )
   END IF
END FUNCTION

FUNCTION create_empty_file(fn)
   DEFINE fn STRING, c base.Channel
   LET c = base.Channel.create()
   CALL c.openFile(fn, "w")
   CALL c.close()
END FUNCTION
```
Handling registration and unregistration requests

The next function is typical Web Service server code using the Web Services API to handle RESTful requests. Note that the TCP port is defined as a constant that is used to set FGLAPPSERVER automatically when not running behind the GAS:

```sql
IMPORT util
IMPORT com

CONSTANT DEFAULT_PORT = 9999

MAIN
...
CALL handle_registrations()
END MAIN

FUNCTION handle_registrations()
DEFINE req com.HTTPServiceRequest,
   url, method, version, content_type STRING,
   reg_data, reg_result STRING
IF LENGTH(fgl_getenv("FGLAPPSERVER"))==0 THEN
   -- Normally, FGLAPPSERVER is set by the GAS
   DISPLAY SFMT("Setting FGLAPPSERVER to %1", DEFAULT_PORT)
   CALL fgl_setenv("FGLAPPSERVER", DEFAULT_PORT)
END IF
CALL com.WebServiceEngine.Start()
WHILE TRUE
   TRY
      LET req = com.WebServiceEngine.getHTTPServiceRequest(20)
      CATCH
         IF STATUS==-15565 THEN
            DISPLAY "TCP socket probably closed by GAS, stopping
process..."
            EXIT PROGRAM 0
         ELSE
            DISPLAY "Unexpected getHTTPServiceRequest() exception: ",
            STATUS
            DISPLAY "Reason: ", SQLCA.SQLERRM
            EXIT PROGRAM 1
         END IF
      END TRY
      IF req IS NULL THEN -- timeout
         DISPLAY SFMT("HTTP request timeout...: %1", CURRENT YEAR TO
         FRACTION)
         CALL check_apns_feedback()
         CALL show_tokens()
         CONTINUE WHILE
      END IF
      LET url = req.getURL()
      LET method = req.getMethod()
      IF method IS NULL OR method != "POST" THEN
         IF method == "GET" THEN
            CALL req.sendTextResponse(200,NULL,"Hello from token
            maintainer...")
         ELSE
            DISPLAY SFMT("Unexpected HTTP request: %1", method)
            CALL req.sendTextResponse(400,NULL,"Only POST requests
            supported")
         END IF
         CONTINUE WHILE
      END IF
      LET version = req.getRequestVersion()
      IF version IS NULL OR version != "1.1" THEN
```
PROCESSING REGISTRATION AND UNREGISTRATION COMMANDS

The next function is called when a RESTful request is to be processed. The URL will define the type of command to be executed by the server:

- If the URL contains "/token_maintainer/register", a new token must be inserted in the database.
- If the URL contains "/token_maintainer/unregister", an existing token must be deleted from the database.

FUNCTION process_command(url, data)
DEFINE url, data STRING
DEFINE data_rec RECORD
    notification_type VARCHAR(10),
    registration_token VARCHAR(250),
    badge_number INTEGER,
    app_user VARCHAR(50)
END RECORD,
DEFINE p_id INTEGER,
DEFINE p_ts DATETIME YEAR TO FRACTION(3),
DEFINE result_rec RECORD
    status INTEGER,
    message STRING
END RECORD,
DEFINE result STRING
LET result_rec.status = 0
TRY
    CASE
    WHEN url MATCHES "*/token_maintainer/register"
        CALL util.JSON.parse( data, data_rec )
        SELECT id INTO p_id FROM tokens
            WHERE registration_token = data_rec.registration_token
        IF p_id > 0 THEN
            LET result_rec.status = 1
            LET result_rec.message = SFMT("Token already registered:
                [%1]",
                data_rec.registration_token)
        END IF
        GOTO pc_end
    END CASE
END TRY

IF p_id IS NULL THEN LET p_id=1 END IF
LET p_ts = util.Datetime.toUTC(CURRENT YEAR TO FRACTION(3))
WHENEVER ERROR CONTINUE
INSERT INTO tokens
VALUES( p_id, data_rec.notification_type,
data_rec.registration_token, 0, data_rec.app_user,
p_ts )
WHENEVER ERROR STOP
IF SQLCA.SQLCODE==0 THEN
LET result_rec.message = SFMT("Token is now registered:
[ %1 ]", data_rec.registration_token)
ELSE
LET result_rec.status = -2
LET result_rec.message = SFMT("Could not insert token in
database: [ %1 ]", data_rec.registration_token)
END IF
WHEN url MATCHES "token_maintainer/unregister"
CALL util.JSON.parse( data, data_rec )
DELETE FROM tokens
WHERE registration_token = data_rec.registration_token
IF SQLCA.SQLERRD[3]==1 THEN
LET result_rec.message = SFMT("Token unregistered:
[ %1 ]", data_rec.registration_token)
ELSE
LET result_rec.status = -3
LET result_rec.message = SFMT("Could not find token in
database: [ %1 ]", data_rec.registration_token)
END IF
WHEN url MATCHES "token_maintainer/badge_number"
CALL util.JSON.parse( data, data_rec )
WHENEVER ERROR CONTINUE
UPDATE tokens
SET badge_number = data_rec.badge_number
WHERE registration_token = data_rec.registration_token
WHENEVER ERROR STOP
IF SQLCA.SQLCODE==0 THEN
LET result_rec.message = SFMT("Badge number updated for Token:
[ %1 ]\n New value:
[ %2 ]\n", data_rec.registration_token, data_rec.badge_number)
ELSE
LET result_rec.status = -4
LET result_rec.message = SFMT("Badge update failed for token:
[ %1 ]", data_rec.registration_token)
END IF
END CASE
CATCH
LET result_rec.status = -1
LET result_rec.message = SFMT("Failed to register token:
[ %1 ]", data_rec.registration_token)
END TRY
LABEL pc_end:
DISPLAY result_rec.message
LET result = util.JSON.stringify(result_rec)
RETURN result
END FUNCTION
Showing the current registered tokens

The following function is called after a WebServiceEngine timeout, when no request is to be processed. Its purpose is just to show the current list of registered tokens in a server log (stdout):

```favorite
FUNCTION show_tokens()
    DEFINE rec RECORD -- Use CHAR to format
        id INTEGER,
        notification_type CHAR(10),
        registration_token CHAR(250),
        badge_number INTEGER,
        app_user CHAR(50),
        reg_date DATETIME YEAR TO FRACTION(3)
    END RECORD
    DECLARE c1 CURSOR FOR SELECT * FROM tokens ORDER BY id
    FOREACH c1 INTO rec.*
        DISPLAY "   ", rec.id, ": ",
        rec.notification_type, ": ",
        rec.app_user[1,10], "/ ",
        "(" , rec.badge_number USING "<<<<&", " ") ",
        rec.registration_token[1,20],想起来..."
    END FOREACH
    IF rec.id == 0 THEN
        DISPLAY "No tokens registered yet..."
    END IF
END FUNCTION
```

APNs feedback checking

When using Apple Push Notification service, the device token maintainer can also handle device unregistration by querying the APNs feedback service. The APNs feedback service will provide the list of device tokens that are no longer valid because the app on the devices has unregistered.

The token maintainer can use this service to clean up the token database.

**Note:** When using the APNs feedback service, an SSL/TLS certificate needs to be defined in FGLPROFILE as described in APNs SSL/TLS certificate on page 2923.

To get the list of unregistered devices, send an HTTP POST to the following URL:
tcps://feedback.push.apple.com:2196

The next function is called after a timeout when no request needs to be processed by the token maintainer:

```favorite
FUNCTION check_apns_feedback()
    DEFINE req com.TCPRequest,
         resp com.TCPResponse,
         feedback DYNAMIC ARRAY OF RECORD
            timestamp INTEGER,
            deviceToken STRING
    END RECORD,
    IF arg_val(1) != "APNS" THEN RETURN END IF
    DISPLAY "Checking APNs feedback service..."
    LOCATE data IN MEMORY
    TRY
        LET req = com.TCPRequest.create( "tcps://
            feedback.push.apple.com:2196"
```
CALL req.setKeepConnection(true)
CALL req.setTimeout(2)
CALL req.doRequest()
LET resp = req.getResponse()
CALL resp.getDataResponse(data)
CALL com.APNS.DecodeFeedback(data,feedback)
FOR i=1 TO feedback.getLength()
  LET timestamp =
    util.Datetime.fromSecondsSinceEpoch(feedback[i].timestamp)
  LET timestamp = util.Datetime.toUTC(timestamp)
  LET token = feedback[i].deviceToken
  DELETE FROM tokens
  WHERE registration_token = token
    AND reg_date < timestamp
END FOR
CATCH
  CASE STATUS
    WHEN -15553 DISPLAY "APNS feedback: Timeout: No feedback message"
    WHEN -15566 DISPLAY "APNS feedback: Operation failed ":",
    SQLCA.SQLERRM
    WHEN -15564 DISPLAY "APNS feedback: Server has shutdown"
    OTHERWISE DISPLAY "APNS feedback: ERROR ":",STATUS
END CASE
END TRY
END FUNCTION

For more details about APNs feedback service, see the Apple Push Notification Service documentation.

**Related concepts**

- Firebase Cloud Messaging (FCM) on page 3621
  Follow this procedure to implement push notification with FCM.

- Apple Push Notification Service (APNs) on page 3625
  Follow this procedure to implement push notification with APNs.

**Handling notifications in the mobile app**

This topic describes how to handle push notification in the app running on mobile devices.

**Basics**

In order to implement a push notification mechanism, you need to set up a server part (token maintainer and push notification server), based on a push notification framework such as Firebase Cloud Messaging (FCM) or Apple® Push Notification service (APNs). In Addition, you need to handle notification events in your mobile app. This section describes how to implement push notification in the app with the push notification API available in Genero BDL.

The same code base can be used to handle push notifications for Android™ (using FCM) and iOS (using APNs) devices. Only the content of the notification message will have to be processed with specific code, as the structure of the message differs depending on standards defined by the push notification framework.

**Genero API for push notifications**

Genero BDL provides an API to handle push notification on mobile apps. Dedicated front calls are available to register to a push server, fetch push notification data, and unregister:

- registerForRemoteNotifications on page 2708
- getRemoteNotifications on page 2703
- unregisterFromRemoteNotifications on page 2713
To detect when a notification message arrives from the push server, a specific action called `notificationpushed` must be used by app code on a `ON ACTION` handler. This special action is referenced as a predefined action.

**Android™ app permissions for FCM push notifications**

Android apps using push notification services need specific permissions (Android manifest), such as:

- `android.permission.INTERNET`
- `android.permission.GET_ACCOUNTS`
- `android.permission.WAVE_LOCK`
- `com.google.android.c2dm.permission.RECEIVE`
- `application-package-name.permission.C2D_MESSAGE` where `application-package-name` is the Android™ package name of your app (for example, `com.mycompany.pushclient`)

Permissions are automatically set for Android™ APK packages by the GMA buildtool. As some permissions need to be prefixed with the package name, they are applied with the `--build-app-package-name` option.

See the FCM documentation for more details about required permissions for push notifications.

**iOS app certificates for APNs push notifications**

iOS apps must be created with an Apple® certificate for development or distribution, linked to an App ID (or Bundle ID) with push notification enabled. The provisioning profile used when building the IPA must be linked to the App ID with push enabled. Certificate, provisioning and bundle id must be specified to the GMI buildtool.

**Handling push notification in the app**

To handle push notifications in your mobile app, perform the following steps:

1. Register to the push service and get the registration token
2. Send the push notification token to your token maintainer
3. Handle notification events with the `notificationpushed` action
4. Eventually un-register from the push servers

**1 - Registering to the push service and to the push provider**

Register the app to the push notification service with the `registerForRemoteNotifications` front call.

- When using FCM, you must provide Sender ID to identify the FCM project.
- When using APNs, you can set the Sender ID to NULL.

**Note:** The app does not need to register for notification each time it is restarted. Even if the app is closed, the registration is still active until the `unregisterFromRemoteNotifications` front call is performed. At first execution, an app will typically ask if the user wants to get push notifications and register to the push service if needed. To disable push notification, apps usually implement an option that can be disabled (to unregister) and re-enabled (to register again) by the user. On Android™, the app must register for notification each time it is upgraded.

**Important:**

When an app restarts, if notifications are pending and the app has already registered for push notification in a previous execution, the `notificationpushed` action will be raised as soon as a dialog with the corresponding `ON ACTION` handler activates. The app then performs a `getRemoteNotifications` on page 2703 front call as in the usual way, to get the pending notifications pushed to the device while the app was off.

However, special consideration needs to be given to iOS devices. When push notification arrives for an iOS app that has not started, there is no mechanism to wake up the app and get the push data. Therefore, when the user starts the app from the springboard, there will never be any push data available. Depending on the context, implement the following programming patterns to solve this problem:
1. If the push notification contains a badge number, the app can verify if the badge is greater than 0 (with the `getBadgeNumber` front call) in order to perform a `getRemoteNotifications` front call. Even if there is no data available with the front call, it is recommended that the app sends a request directly to the server push provider to get last push data.

2. If the push notification does not contain badge numbers, it is still recommended that the app performs a `getRemoteNotification` front call when it starts. If there is no push data available from the front call, the recommendation is that the app sends a request to the server push provider to see if there is push data available. This is by the way also recommended when receiving a `notificationpushed` action during application life time.

3. If the user starts the app from the Notification Center, the app is launched with push data transmitted from the system, and the `notificationpushed` action is sent. It is recommended that the app perform a `getRemoteNotifications` front call and get the push data.

The `registerForRemoveNotifications` front call will return a registration token for the app which will be used by the push server (a.k.a push provider).

- When using FCM, the returned identifier is the FCM "registration token".
- When using APNs, the returned identifier is the APNs "device token".

```plaintext
DEFINE rec RECORD
   tm_host STRING,
   tm_port INTEGER,
   notification_type STRING,
   user_name STRING,
   registration_token STRING
END RECORD
...
LET rec.tm_host = "https://pushreg.example.orion"
LET rec.tm_port = 4930
LET rec.app_user = "mike"
LET rec.notification_type = "FCM"
...
DIALOG ATTRIBUTES(UNBUFFERED)
   INPUT BY NAME rec.tm_host,
      rec.tm_port,
      rec.notification_type,
      rec.user_name,
      rec.registration_token
   ATTRIBUTES(WITHOUT DEFAULTS)
END INPUT
...
ON ACTION register
   LET rec.registration_token = register(rec.notification_type,
      rec.user_name)
...
FUNCTION register(notification_type, app_user)
   DEFINE notification_type STRING,
      app_user STRING
   DEFINE registration_token STRING
   TRY
      CALL ui.Interface.frontCall(  
         "mobile", "registerForRemoteNotifications",
         [], [ registration_token ] )
      IF tm_command( "register", notification_type,
         registration_token, app_user, 0 ) < 0 THEN
          RETURN NULL
      END IF
   END TRY
```
2 - Sending a push notification token to your token maintainer

Once registered to the FCM or APNs service, the app must also register to the push server or push provider by sending the token obtained in step 1.

This is typically done by using a RESTful HTTP POST, sending the token (along with additional application user information) to a dedicated server program that maintains the list of registered devices/tokens.

The device token maintainer can be implemented in BDL as a Web Service program, as described in Implementing a token maintainer on page 3631.

In this tutorial, the `tm_command()` function implements token registration (as well as badge number handling for APNs):

```bdb
IMPORT com
IMPORT util
...
LET rec_tm_host = "https://pushreg.example.orion"
LET rec_tm_port = 4930
...
FUNCTION tm_command( command, notification_type, registration_token,
                      app_user, badge_number )
DEFINE command STRING,
    notification_type STRING,
    registration_token STRING,
    app_user STRING,
    badge_number INTEGER
DEFINE url STRING,
    json_obj util.JSONObject,
    req com.HTTPRequest,
    resp com.HTTPResponse,
    json_result STRING,
    result_rec RECORD
    status INTEGER,
    message STRING
END RECORD
TRY
    LET url = SFMT( "http://%1:%2/token_maintainer/%3",
                    rec_tm_host, rec_tm_port, command )
    LET req = com.HTTPRequest.create(url)
    CALL req.setHeader("Content-Type", "application/json")
    CALL req.setMethod("POST")
    CALL req.setConnectionTimeOut(5)
    CALL req.setTimeOut(5)
    LET json_obj = util.JSONObject.create()
    CALL json_obj.put("notification_type", notification_type)
    CALL json_obj.put("registration_token", registration_token)
    CALL json_obj.put("app_user", app_user)
    CALL json_obj.put("badge_number", badge_number)
    CALL req.doTextRequest(json_obj.toString())
    LET resp = req.getResponse()
    IF resp.getStatusCode() != 200 THEN
        MESSAGE SFMT("HTTP Error (%1) %2",
                      resp.getStatusCode(), resp.getStatusMessage())
    ELSE
        CALL json_result = resp.getData()
        IF json_result.isSuccess() THEN
            LET result_rec = json_result.getData()
            IF result_rec.status = 0 THEN
                MESSAGE SFMT("Registration succeeded (token=%1)",
                              registration_token)
                RETURN registration_token
            ELSE
                MESSAGE SFMT("Registration failed.")
                RETURN NULL
            END IF
        ELSE
            MESSAGE SFMT("Registration failed.")
            RETURN NULL
        END IF
    END IF
END TRY
MESSAGE SFMT("Registration succeeded (token=%1)", registration_token)
RETURN registration_token
END FUNCTION
```
When the app is declared as push notification client to the push server, continue with the normal program flow.

3 - Handling push notification events

To get and handle notification events, the current active dialog must implement the notificationpushed special action.

To control action view rendering defaults and current field validation behavior when the notificationpushed action is used, consider setting action default attributes for this action in your .4ad file as follows:

```
<ActionDefaultList>
  ...
  <ActionDefault name="notificationpushed" validate="no" defaultView="no"
    contextMenu="no"/>
  ...
</ActionDefaultList>
```

Another option is to define these action defaults attributes in the ON ACTION handler:

```
ON ACTION notificationpushed (VALIDATE=NO, DEFAULTVIEW=NO)
  ...
```

In the ON ACTION block for this action, query for notification messages by using the "getRemoteNotifications" front call, (passing the Sender ID as parameter when using FCM, for APNs the Sender ID must be NULL). This front call returns a JSON string containing a list of notification messages to be processed:

```
DEFINE notifs DYNAMIC ARRAY OF RECORD
  info STRING,
  ts DATETIME YEAR TO FRACTION(3)
END RECORD

... DEFINE x INTEGER

DIALOG ...
  DISPLAY ARRAY notifs TO sr.*
  END DISPLAY
  ...
  ON ACTION notificationpushed
    LET x=handle_notification()
```
CALL DIALOG.setCurrentRow("sr",x)

... END DIALOG ...

FUNCTION handle_notification()

DEFINE notif_list STRING,
    notif_array util.JSONArray,
    notif_item util.JSONObject,
    notif_data util.JSONObject,
    aps_record util.JSONObject,
    gcm_data_s STRING,
    gcm_genero_notification_s STRING,
    gcm_genero_notification util.JSONObject,
    info, other_info STRING,
    i, x INTEGER

CALL ui.Interface.frontCall(
    "mobile", "getRemoteNotifications",
    [ ], [ notif_list ] )

TRY

LET notif_array = util.JSONArray.parse(notif_list)
IF notif_array.getLength() > 0 THEN
    CALL setup_badge_number(notif_array.getLength())
END IF
FOR i=1 TO notif_array.getLength()
    LET info = NULL
    LET other_info = NULL
    LET notif_item = notif_array.get(i)
    -- Try APNs msg format
    LET aps_record = notif_item.get("aps")
    IF aps_record IS NOT NULL THEN
        LET info = aps_record.get("alert")
        LET notif_data = notif_item.get("custom_data")
        IF notif_data IS NOT NULL THEN
            LET other_info = notif_data.get("other_info")
        END IF
    END IF
ELSE
    -- Try GCM msg format
    LET gcm_data_s = notif_item.get("data")
    IF gcm_data_s IS NOT NULL THEN
        LET notif_data = util.JSONObject.parse(gcm_data_s)
        IF notif_data IS NOT NULL THEN
            LET gcm_genero_notification_s = notif_data.get("genero_notification")
            LET gcm_genero_notification = util.JSONObject.parse(gcm_genero_notification_s)
            IF gcm_genero_notification IS NOT NULL THEN
                LET info = gcm_genero_notification.get("content")
            END IF
            LET other_info = notif_data.get("other_info")
        END IF
    END IF
END IF
IF info IS NULL THEN
    LET info = "Unexpected message format"
END IF
MESSAGE SFMT("Notification message:
%1
%2", info, other_info)
CALL notifs.appendElement()
LET x = notifs.getLength()
LET notifs[x].info = SFMT("%1 (%2)", info, other_info)
LET notifs[x].ts = CURRENT
END FOR
CATCH
When using APNs, the app must handle the badge numbers attached to the device token. The app must:

1. Query the current badge number with the `getBadgeNumber` front call.
2. Compute the new badge number based on the number of notifications consumed.
3. Reset the badge number with the `setBadgeNumber` front call.
4. Inform the token maintainer to sync the badge number in the central database.

The following function handles badge numbers for the app:

```plaintext
FUNCTION setup_badge_number(consumed)
DEFINE consumed INTEGER
DEFINE badge_number INTEGER
TRY -- If the front call fails, we are not on iOS...
   CALL ui.Interface.frontCall("ios", "getBadgeNumber", [],
   [badge_number])
CATCH
   RETURN
END TRY
IF badge_number>0 THEN
   LET badge_number = badge_number - consumed
END IF
CALL ui.Interface.frontCall("ios", "setBadgeNumber", [badge_number], [])
IF tm_command( "badge_number", "APNS", rec.registration_token, 
   rec.user_name, badge_number) < 0 THEN
   ERROR "Could not send new badge number to token maintainer."
RETURN
END IF
END FUNCTION
```

4 - Unregistering the app from push notification

If the app no longer wants to get push notifications, unregister from the push provider (using a RESTful POST, in the `regunreg_token()` function), and unregister from the push service by using the "unregisterFromRemoteNotifications" front call.

- When using FCM, you must pass the FCM Sender ID as parameter.
- When using APNs, the parameter must be NULL.

```plaintext
... 
LET rec.tm_host = "https://pushreg.example.orion"
LET rec.tm_port = 4930
CALL unregister(rec.registration_token, rec.app_user)
...
```

```plaintext
FUNCTION unregister(notification_type, registration_token, app_user)
DEFINE notification_type STRING,
registration_token STRING,
app_user STRING
IF tm_command( "unregister", notification_type, 
   registration_token, app_user, 0 ) < 0 THEN
   RETURN
END IF
TRY
```
CALL ui.Interface.frontCall(
    "mobile", "unregisterFromRemoteNotifications",
    [ ], [ ]
)
CATCH
    MESSAGE "Un-registration failed (broadcast service)."
RETURN
END TRY
MESSAGE "Un-registration succeeded"
END FUNCTION

Related concepts
Firebase Cloud Messaging (FCM) on page 3621
Follow this procedure to implement push notification with FCM.

Apple Push Notification Service (APNs) on page 3625
Follow this procedure to implement push notification with APNs.

Cordova plugins
This section describes how to use Cordova plugins.

Understanding Cordova plugins
Cordova plugins allow you to access specific mobile device functionalities.

What are Cordova plugins in Genero?
Cordova plugins are based on the Apache Cordova cross-platform mobile development framework, to access mobile device functionality such as the accelerometer, the camera, the compass, the microphone, and more.

GMI and GMA act as a plugin container and provide a plugin API widely compatible with the original Cordova API. The implementation is Genero-specific and makes the native plugin interfaces available in Genero BDL through Cordova plugin front calls.

Cordova plugins with device simulator or in client/server GUI mode
Depending on the configuration used when executing your app, some Cordova plugins or plugin APIs may not be available and may cause a malfunction.

• When using a device simulator, some Cordova plugin will not work because the feature is not available on the simulator.
• When executing the app on a server in development mode or in "runOnServer" mode, the front calls are slowed down by the network roundtrips.

Installing Cordova plugins into the development environment
In order to be used, Cordova plugins have to be downloaded from the FOURJS Cordova GitHub or from the Apache Cordova GitHub, and must be installed by using the --install-plugin option of GMA or GMI build tools.

A Cordova plugin installation into GMA/GMI installation directories must be done by using a local clone of the GitHub repository.

For more details, see Installing Cordova plugins on page 3645.

Building apps with Cordova plugins
Installed Cordova plugins need to be specified in the build process when creating your app.

The Cordova plugins to be bundled with the app are specified with the --cordova-plugin option of GMA and GMI build tools:
• Embed Cordova plugins in a GMA app on page 3649
• Embed Cordova plugins in a GMI app on page 3649

Using Cordova plugin APIs (wrapper functions)
Access to the native code can be achieved using native Cordova APIs through Cordova plugin front calls.
To ease the usage of a Cordova plugin, a library of BDL functions can encapsulate the Cordova front calls. Most Cordova plugins provided by FOURJS are shipped with a BDL wrapper library, available from the FOURJS Cordova GitHub.

Installing Cordova plugins
Before usage, Cordova plugins need to be installed in the GMA or GMI development environment.

Requirements for Cordova plugins installation
The following tools and utilities are required to install Cordova plugins:

1. Cordova plugins are available on a GitHub repository. Therefore, the git command line tool is required to install Cordova plugins.
2. The gmabuildtool uses the make utility to compile FGL wrappers and install Cordova plugins. Therefore, the make utility must be installed in the computer.

Cordova plugin installation basics
GitHub contains a large number of Cordova plugins that can be used in your app, like those from the Apache GitHub for example, or the FOURJS Cordova GitHub.
Before installing the Cordova plugins in the GMA / GMI installation directories, you must first clone the GitHub repository to your local disk.

Upgrading GMA, GMI or FGL with GMI with installed Cordova plugins
Cordova plugins need to be re-installed in the following cases:

• With iOS/GMI: after upgrading the GMI package or the FGL package (the GMI package can be installed into the FGLDIR installation directory, or in a separate GMIDIR installation directory)
• With Android/GMA: after upgrading the GMA package (the GMA package is always to be installed in a dedicated GMADIR installation directory)

Important: In all cases, after upgrading an FGL, GMI or GMA environment where plugins have been installed, you need to re-execute the plugin installation with a gmabuildtool scaffold --install-plugin or gmibuildtool --install-plugin command.

Installing a Cordova plugin from a local clone of the GitHub repository
First clone the GitHub repository into a local directory:

```
$ cd /usr/dev/myplugins
$ git clone https://github.com/apache/cordova-plugin-network-information.git
```

Then install the plugin:

• For GMI / iOS:

```
$ gmibuildtool --install-plugins=/usr/dev/myplugins/cordova-plugin-network-information
```
- For GMA / Android™ (note that you need to use the "scaffold" command):

```
$ gmabuildtool scaffold \
  --install-plugins /usr/dev/myplugins/cordova-plugin-network-information
```

**Note:** The gmabuildtool compiles cordova plugins FGL wrappers at installation with the make command. Compiled .42m modules will be bundled inside the scaffold archive. Before building an apk, FGL wrapper binaries will be copied inside the application directory.

### Checking out a particular version from GitHub

If you want to test a particular version of a Cordova plugin, make a clone of the repository, and check out that version, then install the plugin:

```
$ cd cordova-plugin-network-information
$ git checkout specific-version
$ cd -
```

It is now possible to use the gmibuildtool or gmabuildtool to install this specific version.

**Note:** If a plugin with the same name is already installed, it is replaced by the new installation.

### Checking Cordova plugin installation

Use the --list-plugins option of the GMA or GMI build tool, to check that the plugin is properly installed:

- For GMI / iOS:

```
$ gmibuildtool --list-plugins
... cordova-plugin-network-information ...
```

- For GMA / Android™ (note that you need to use the "scaffold" command):

```
$ gmabuildtool scaffold --list-plugins
... cordova-plugin-network-information ...
```

The **cordova-plugin-network-information** name can now be used for bundling an application using that plugin.

### Finding Cordova plugin wrapper libs in development environment

After installing the Cordova plugin, if the plugin repository includes a BDL wrapper library, the fglrun runtime system needs to know where the .42m p-code modules can be found.

If there is a local clone of the GitHub repository, search for *.4gl files (typically prefixed with "fglcdv"), make sure they are compiled to .42m modules, and set the FGLLDPATH to the directory where the .42m files are located.

For example (on Unix/Linux), with the **cordova-plugin-media** repository clone:

```
$ cd cordova-plugin-media
$ find . -name "fglcdv*.4gl"
./fgl/fglcdvMedia.4gl
$ cd fgl
$ fglcomp fglcdvMedia.4gl (or make, if makefile is present)
$ export FGLLDPATH=$PWD (or export FGLLDPATH=$PWD:$FGLLDPATH)
Using a Cordova plugin API

Cordova plugin features can be used by invoking cordova front calls.

How to access Cordova plugin APIs

Cordova plugin APIs are available to the front-end and therefore need to be called through Cordova plugin front calls. Before calling a Cordova plugin function, you need to identify what native APIs are available. If provided, check the Cordova plugin API documentation for available functions.

It is a good practice to implement BDL wrapper functions that encapsulate the Cordova front calls specific to this plugin. BDL wrapper libraries are provided with the plugins on the FOURJS Cordova GitHub.

Identifying and verifying the version of available Cordova plugins at runtime

At runtime, the code of an app can query available Cordova plugins with the cordova.listPlugins and get details of the plugin with cordova.getPluginInfo front calls.

To verify the git version in the wrapper modules for usage in direct mode and GAS mode, implement an initialization function in the wrapper module, to check that the GIT_VERSION string defined in the git_version.txt file (found nearby the wrapper library sources) corresponds to the current available plugin version returned by the cordova.getPluginInfo front call:

```javascript
#include "git_version.txt"  -- defines GIT_VERSION
...
DEFINE initialized BOOLEAN
...
PUBLIC FUNCTION initialize()
  IF initialized THEN
    -- exclusive library usage
    DISPLAY "The library is already in use."
    EXIT PROGRAM 1
  END IF
  CALL checkGitVersion(GIT_VERSION)
  LET initialized = TRUE
END FUNCTION
...
PRIVATE FUNCTION checkGitVersion(version STRING)
  DEFINE rec RECORD
    git_version STRING
  END RECORD
  CALL ui.interface.frontcall("cordova","getPluginInfo",["Accelerometer"],
    [rec])
  IF NOT version.equals(rec.git_version) THEN
    DISPLAY sfmt("fglcdnMotion.4gl version mismatch, git version:%1,
      expecting:%2.",
      rec.git_version, version)
    EXIT PROGRAM 1
  END IF
END FUNCTION
```

Identifying Cordova plugin native APIs in JS sources

Cordova plugins are primarily targeted for JavaScript. Each plugin source directory has a www directory with JavaScript wrapper code calling into the native code.

For example, with the cordova-plugin-network-information plugin, the www/network.js file implements the following *getInfo* JavaScript function, that performs an exec() call of the "NetworkStatus.getConnectionInfo" native Cordova function:

```javascript
NetworkConnection.prototype.getInfo =
```
function(successCallback, errorCallback) {
    exec(successCallback, errorCallback,
        "NetworkStatus", "getConnectionInfo", []);
}

The BDL equivalent of the above exec() call is done with the generic Cordova front call "cordova.call":
CALL ui.Interface.frontCall( "cordova", "call",
    ["NetworkStatus","getConnectionInfo"], [result] )

Note: In the README.md files of the plugins, the available methods for the plugins are documented for the JavaScript wrappers and not the underlying native calls.

In order to find available native Cordova APIs, such as getConnectionInfo, search for the exec() calls in the js files of the www directory. The native Cordova function name is the fourth parameter of the exec() function.

Implementing an app that uses the Cordova plugin

After installing a plugin, you can create a Genero program using a function of this plugin, by using the generic front call for Cordova APIs.

For example:

IMPORT FGL fgldialog
MAIN
DEFINE result STRING
MENU "network"
    COMMAND "Info"
    CALL ui.Interface.frontCall( "cordova", "call",
        ["NetworkStatus","getConnectionInfo"], [result] )
    CALL fgl_winMessage("Result",result,"info")
    COMMAND "Exit"
    EXIT MENU
END MENU
END MAIN

Compile and build your app as usual (the installed Cordova plugins will be included).

Cordova plugin front calls overview

Cordova plugin front calls are generic functions that give access to the native Cordova APIs, in conjunction with the predefined cordovacallback action.

The cordova.call function issues a synchronous Cordova function call. In this case, the underlying native function is executed and the program waits until it returns. Results are provided in the front call output parameters.

The cordova.callWithoutWaiting function performs an asynchronous Cordova function call. The program continues while the Cordova function executes in parallel. The program can implement a trigger for the cordovacallback predefined action, to detect when results are available. To retrieve results of asynchronous calls, use the cordova.getCallbackDataCount, cordova.getCallbackData and cordova.getAllCallbackData front calls.

See these front calls in action in the demos provided on the FOURJS Cordova GitHub.

Implement BDL wrappers on top of Cordova front calls

For better code readability and maintenance, we strongly suggest you implement BDL functions that encapsulate Cordova front calls, as done with plugins available on the FOURJS Cordova GitHub.

Cordova plugin API internals

The IOS code is located in the src/ios directory of a plugin.
The Android™ code is located in the src/Android directory of a plugin.
The IOS code in src/ios/CDVConnection.m contains a method called

```
(void)getConnectionInfo:(CDInvokedUrlCommand*)command
```

This is the native objective-C method called by the front call.
The Android™ code in src/android/NetworkManager.java, contains an execute method for handling the getConnectionInfo string:

```
public boolean execute(String action, JSONArray args, CallbackContext callbackContext) {
if (action.equals("getConnectionInfo")) {
```

## Embed Cordova plugins in a GMA app
To be included in your app, Cordova plugins need to be specified in the build process.

### Including Cordova plugins in your Android APK package
Use the `gmabuildtool --build-cordova` command-line option, to specify which Cordova plugins to embed in your Android™ APK package:

```
gmabuildtool build
    --build-cordova cordova-plugin-contacts,cordova-plugin-device
```

### The plugin.xml file
A plugin can contain several assets (images, sounds, native forms) which are listed in the `plugin.xml` file that is bundled by gmabuildtool.

### Shipping the Cordova wrapper library with the app package
If the Cordova plugin front calls are called from BDL wrapper functions, the .42m p-code module of the wrapper library needs to be included in the APK package.

When building the APK package for an app using Cordova plugins, the `gmabuildtool` automatically includes the corresponding .42m wrapper module(s).

**Related concepts**
- [Deploying mobile apps on Android devices on page 3589](#)
  This section contains information to create a mobile application to be deployed on Android™ devices.

## Embed Cordova plugins in a GMI app
To be included in your app, Cordova plugins need to be specified in the build process.

### Including Cordova plugins in your iOS IPA package
Use the `gmibuildtool --build-cordova` command-line option, to specify which Cordova plugins to embed in your iOS IPA package:

```
$ gmibuildtool --build-cordova=cordova-plugin-media,GeneroTestPlugin
```

### The plugin.xml file
A plugin can contain several assets (images, sounds, native forms) which are listed in the `plugin.xml` file that is bundled by gmibuildtool.
Shipping the Cordova wrapper library with the app package

If the Cordova plugin front calls are called from BDL wrapper functions, the .42m p-code module of the wrapper library needs to be included in the IPA package.

When building the API package for an app using Cordova plugins, the gmibuildtool automatically includes the corresponding .42m wrapper module(s).

Related concepts

Deploying mobile apps on iOS devices on page 3604
This section contains information to create a mobile application to be deployed on iOS devices.

Copyrights and Trademarks

The Copyright and trademark page is shared amongst all Four Js documentation. Appearance in this page does not guarantee a mention within the current document.

Copyright notice

© Copyright Four Js Development Tools, Inc. Use, duplication or disclosure restricted.

Trademarks

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated.

IBM and WebSphere are trademarks or registered trademarks of International Business Machines Corp., in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at Copyright and trademark information.

Linux is a registered trademark of Linus Torvalds.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

DEBIAN is a registered trademark owned by Software in the Public Interest, Inc. Debian trademark is a registered United States trademark of Software in the Public Interest, Inc., managed by the Debian project. Genero is not affiliated with Debian. Debian is a registered trademark owned by Software in the Public Interest, Inc.

Red Hat is a registered trademark of Red Hat, Inc. in the United States and other countries.

Red Hat Enterprise Linux is a registered trademark of Red Hat, Inc. in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.
# Index

## Numerics

<table>
<thead>
<tr>
<th>Version</th>
<th>Feature</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.00</td>
<td>upgrade notes web services upgrade</td>
<td>192</td>
</tr>
<tr>
<td>2.10</td>
<td>upgrade notes SQL Server DATETIME</td>
<td>190</td>
</tr>
<tr>
<td>2.20</td>
<td>upgrade notes web services upgrade</td>
<td>183</td>
</tr>
<tr>
<td>2.21</td>
<td>upgrade notes web services upgrade</td>
<td>181</td>
</tr>
<tr>
<td>2.30</td>
<td>upgrade notes style Form.resetFormSize</td>
<td>180</td>
</tr>
<tr>
<td>2.40</td>
<td>upgrade notes web services upgrade</td>
<td>172</td>
</tr>
<tr>
<td>2.50</td>
<td>upgrade notes web services upgrade</td>
<td>168</td>
</tr>
<tr>
<td>3.00</td>
<td>upgrade notes Built-in icons desupported</td>
<td>158</td>
</tr>
<tr>
<td></td>
<td>Current datetime in UTC</td>
<td>161</td>
</tr>
<tr>
<td></td>
<td>FGL_GETVERSION()</td>
<td>158</td>
</tr>
<tr>
<td></td>
<td>FGL-3893</td>
<td>156</td>
</tr>
<tr>
<td></td>
<td>fglcomp compilation time improvements</td>
<td>159</td>
</tr>
<tr>
<td></td>
<td>FreeTDS (SQL Server versions &gt; 2005)</td>
<td>157</td>
</tr>
<tr>
<td></td>
<td>Front call changes</td>
<td>159</td>
</tr>
<tr>
<td></td>
<td>Grid-based form layout for mobile apps</td>
<td>154</td>
</tr>
<tr>
<td></td>
<td>MariaDB support</td>
<td>157</td>
</tr>
<tr>
<td></td>
<td>MySQL DATETIME</td>
<td>156</td>
</tr>
<tr>
<td></td>
<td>MySQL VARCHAR</td>
<td>155</td>
</tr>
<tr>
<td></td>
<td>Oracle NUMBER</td>
<td>154</td>
</tr>
<tr>
<td></td>
<td>Oracle scroll cursors</td>
<td>155</td>
</tr>
<tr>
<td></td>
<td>PostgreSQL DATETIME</td>
<td>156</td>
</tr>
<tr>
<td></td>
<td>preprocessor behavior change</td>
<td>158</td>
</tr>
<tr>
<td></td>
<td>Presentation style changes</td>
<td>158</td>
</tr>
<tr>
<td></td>
<td>removed database drivers</td>
<td>154</td>
</tr>
<tr>
<td></td>
<td>SQL Server SERIAL emulation</td>
<td>159</td>
</tr>
<tr>
<td></td>
<td>Structured arrays in dialogs</td>
<td>162</td>
</tr>
<tr>
<td></td>
<td>version</td>
<td>158</td>
</tr>
<tr>
<td></td>
<td>web services upgrade</td>
<td>151</td>
</tr>
<tr>
<td>3.10</td>
<td>upgrade notes Application directory (FGLRESOURCEPATH search)</td>
<td>148</td>
</tr>
<tr>
<td></td>
<td>BUTTONEDIT action</td>
<td>151</td>
</tr>
<tr>
<td></td>
<td>C Extensions</td>
<td>134</td>
</tr>
<tr>
<td></td>
<td>dbmsnc_13</td>
<td>141</td>
</tr>
<tr>
<td></td>
<td>dbmsnc_17</td>
<td>141</td>
</tr>
<tr>
<td></td>
<td>Error box</td>
<td>150</td>
</tr>
<tr>
<td></td>
<td>FGL-2729</td>
<td>147</td>
</tr>
<tr>
<td></td>
<td>FGL-2953</td>
<td>142</td>
</tr>
<tr>
<td></td>
<td>FGL-3713</td>
<td>142</td>
</tr>
<tr>
<td></td>
<td>FGL-4192</td>
<td>147</td>
</tr>
<tr>
<td></td>
<td>FGL-4216</td>
<td>150</td>
</tr>
<tr>
<td></td>
<td>FGL-4239</td>
<td>148</td>
</tr>
<tr>
<td></td>
<td>FGL-4241</td>
<td>148</td>
</tr>
<tr>
<td></td>
<td>FGL-4556</td>
<td>143</td>
</tr>
<tr>
<td></td>
<td>FGL-4657 / GDC-3733</td>
<td>140</td>
</tr>
<tr>
<td></td>
<td>fgilhint_SQL hints</td>
<td>147</td>
</tr>
<tr>
<td></td>
<td>front calls changes</td>
<td>138</td>
</tr>
<tr>
<td></td>
<td>GMA changes</td>
<td>130</td>
</tr>
<tr>
<td></td>
<td>GMI changes</td>
<td>133</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Version</th>
<th>Feature</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.20</td>
<td>C Extensions</td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>case insensitive UI objects</td>
<td>126</td>
</tr>
<tr>
<td></td>
<td>Circular IMPORT FGL</td>
<td>126</td>
</tr>
<tr>
<td></td>
<td>database drivers changes</td>
<td>117</td>
</tr>
<tr>
<td></td>
<td>DATETIME type mapping</td>
<td>118</td>
</tr>
<tr>
<td></td>
<td>dynamic array assignment</td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>fglrun -m / fglrun -M</td>
<td>126</td>
</tr>
<tr>
<td></td>
<td>FreeTDS version</td>
<td>123</td>
</tr>
<tr>
<td></td>
<td>front calls changes</td>
<td>116</td>
</tr>
<tr>
<td></td>
<td>GLOBALS</td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>GMA changes</td>
<td>112</td>
</tr>
<tr>
<td></td>
<td>GMI changes</td>
<td>114</td>
</tr>
<tr>
<td></td>
<td>imageCache</td>
<td>115</td>
</tr>
<tr>
<td></td>
<td>PostgreSQL 12</td>
<td>124</td>
</tr>
<tr>
<td></td>
<td>record assignment</td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>SQL interruption</td>
<td>124</td>
</tr>
<tr>
<td></td>
<td>SQL Server driver performance</td>
<td>124</td>
</tr>
<tr>
<td></td>
<td>style Window.actionPanelPosition</td>
<td>115</td>
</tr>
<tr>
<td></td>
<td>style Window.ringMenuPosition</td>
<td>115</td>
</tr>
<tr>
<td></td>
<td>style Window.toolBarPosition</td>
<td>115</td>
</tr>
<tr>
<td></td>
<td>web component changes</td>
<td>117</td>
</tr>
<tr>
<td></td>
<td>web services upgrade</td>
<td>129</td>
</tr>
</tbody>
</table>

|          | Empty POST/PUT request                                                 | 110  |
A

action configuration
  accelerator keys 1774
  key modifiers 1775
  virtual keys 1773

action rendering
  mobile apps 3575

aggregate - reports
  AVG 2062
  COUNT 2061
  MAX 2064
  MIN 2063
  PERCENT 2061
  SUM 2062

Android
  JNI Java Native Interface 2194

Android apps (GMA)
  building APKs 3591
  directory structure 3589

app
  client-server 3565
  connected 3565
  development mode 3565
  partially-connected 3565
  standalone 3565

app types 3565

assignment
  LET instruction 402

attributes
  acceleratorName (.4ad) 1163
  acceleratorName2 (.4ad) 1163
  acceleratorName3 (.4ad) 1163
  acceleratorName4 (.4ad) 1163

  action defaults (.4ad) 1163
  actions 1744
  comment (.4ad) 1163
  contextMenu (.4ad) 1163
  defaultView (.4ad) 1163
  image (.4ad) 1163
  name (.4ad) 1163
  text (.4ad) 1163
  validate (.4ad) 1163

B

background mode
  mobile apps 3573

basics
  actions programming 1739
  dialog programming 1706
  Genero Web Services 3270
  JSON support 575
  user interface 1105

block
  GLOBALS 552
  MAIN 464

boolean expressions
  in form files 1265

bugs 28

built-in class
  base.Application 2406
  base.Channel 2412
  base.MessageServer 2469
  base.SqlHandle 2432
  base.StringBuffer 2447
  base.StringTokenizer 2462
  base.TypeInfo 2465
  introspection 2465
  om.DomDocument 2577
  om.DomNode 2584
  om.NodeList 2607
  om.SaxAttributes 2610
  om.SaxDocumentHandler 2617
  om.XmlReader 2625
  om.XmlWriter 2630
  ui.ComboBox 2561
  ui.Dialog 2512
  ui.DragBox 2570
  ui.Form 2496
  ui.Interface 2470
  ui.Window 2490

built-in function
  arg_val() 2271
  arr_count() 2271
  arr_curr() 2272
  downshift() 2272
  err_get() 2273
  err_print() 2274
  err_quit() 2274
  errorlog() 2275
  fgl_buffertouched() 2275
  fgl_db_driver_type() 2275
  fgl_decimal_exp() 2276
  fgl_decimal_logn() 2276
  fgl_decimal_power() 2277
  fgl_decimal_sqrt() 2276
  fgl_decimal_truncate() 2276
  fgl_dialog_getbuffer() 2277
  fgl_dialog_getbufferlength() 2280
  fgl_dialog_getbufferstart() 2280
  fgl_dialog_getcursort() 2281
  fgl_dialog_getfieldname() 2278
  fgl_dialog_getkeylabel() 2281
  fgl_dialog_getselectionend() 2281
  fgl_dialog_infeld() 2278
  fgl_dialog_setbuffer() 2277
  fgl_dialog_setcursortline() 2280
  fgl_dialog_setcursort() 2279
  fgl_dialog_setfieldorder() 2279
  fgl_dialog_setkeylabel() 2282
  fgl_dialog_setselection() 2282
  fgl_drawbox() 2283
  fgl_drawline() 2284
  fgl_evenloop() 2284
  fgl_getenv() 2285
  fgl_getfile() 2286
  fgl_gethelp() 2285
  fgl_getkey() 2286
  fgl_getkeylabel() 2287
  fgl_getpid() 2285
  fgl_getresource() 2287
C
C language
  compiler requirements 38
  c-extensions
    c-extensions for GMI 2247
    call BDL from C 2245
    call C from BDL 2244
    ESQL/C extensions 2232
    fglcapit_peekStackBufferSize 2235
    fglcapit_peekStackType 2235
    fgIExt.h 2231
    importing 481
    interface file 2233
    NULL handling 2242
    popbigint 2235
    popdub 2235
    popdtime 2235
    popflo 2235
    popint 2235
    popinv 2235
    poplong 2235
    poplocator 2235
    popquote 2235
    popshort 2235
    popstring 2235
    popvchar 2235
    pushbigint 2235
    pushdate 2235
    pushdub 2235
    pushdtime 2235
    pushflo 2235
    pushint 2235
    pushinv 2235
    pushlong 2235
    pushquote 2235
    pushshort 2235
    runime stack 2235
    types and structures 2239
  built-in web components
    fglgallery 1979
    fglrichtext 1972
    fglsvgcanvas 1988
  BY NAME
    dialog programming 1715
  canceling
    SQL statement 601
    cautions - xml.DomDocument 2964
  character set
    application locale 512
    character set units 517
  classes
    JSON 576
    Classes
      OOP 570
  client app configuration
    HTTPS server 3305, 3335, 3423
  code beautifier
    basics 2174
    syntax 2175
    usage 2177
  color and theming
    mobile apps 3581
    presentation styles 1173
    termcap 1125
  com package
    helper classes 2922
    HTTP classes 2871
    HTTPServiceRequest methods 2846
    IMPORT 2811
    specific classes 2923
    TCP classes 2912
    Web services classes 2811
    com.fourjs.fgl.lang.FglByteBlob
      toString 2206
      valueOf 2206
    com.fourjs.fgl.lang.FglDate
      toString 2201
      valueOf 2201
    com.fourjs.fgl.lang.FglDateTime
      encodeTypeQualifier 2202
      toString 2202
      valueOf 2202
    com.fourjs.fgl.lang.FgLDecimal
      encodeTypeQualifier 2205
      toString 2205
      valueOf 2205
    com.fourjs.fgl.lang.FgIInterface
      encodeTypeQualifier 2207
ring menus (MENU) 1467
runtime stack 493
scrollgrid views 1880
SOAP web services
  high-level WS APIs 3278
  low-level WS APIs 3278
split views 1900
SQL support 590
stack-based layout 1431
start menus 2019
static display 1452
Static SQL 694
structured variables (RECORD) 408
table views 1853
toolbars 1436
topmenus 1444
tree views 1889
two-pass report 2051
type conversions 311
ui.Interface.refresh 1710
user defined types 431
user interface 1105
user interface modes 1109
user interruption 1711
variables 390
viewport changes 1417
Web Component 1922
web services
  HTTP compression 3295
  password encryption 3290
  SOAP features 3280
  SOAP multipart style 3297
  stateful web services 3283
Web services
  Service Oriented Architecture (SOA) 3271
  standards (XML, SOAP, REST, WSDL, HTTP) 3272
window containers 2023
windows and forms 1131
XML support 573
concept - form file
  BUTTON item type 1266
  BUTTONEDIT item type 1267
  CHECKBOX item type 1269
  COMBOBOX item type 1270
  DATEEDIT item type 1272
  DATETIMEEDIT item type 1273
  EDIT item type 1274
  FOLDER item type 1275
  GRID item type 1276
  GROUP item type 1276
  HBOX item type 1278
  IMAGE item type 1279
  LABEL item type 1280
  PAGE item type 1282
  PROGRESSBAR item type 1283
  RADIOGROUP item type 1284
  SCROLLGRID item type 1285
  SLIDER item type 1286
  SPINEDIT item type 1287
  TABLE item type 1288
  TEXTEDIT item type 1289
  TIMEEDIT item type 1290
  TREE item type 1292
  VBOX item type 1293
  WEBCOMponent item type 1293, 1923
concept - form files
  action views 1241
  aggregate fields 1248
  database column fields 1243
  form fields 1242
  form item types 1266
  form items 1238
  form tags 1254
  formonly fields 1245
  hbox tags 1261
  item tags 1259
  layout items 1240
  layout tags 1254
  matrix 1259
  phantom fields 1246
  satellite items 1239
  screen array 1251
  screen record 1251
  stack items 1241
  static items 1239
concept - web services
  HTTP cookies 3287
  WS-Addressing 3283
conceptprogram
  coverage tool 2168
  dead code 2168
  execution trace 2171
  function call stack trace 2171
  introspection 2171
  performance 2168
  reachable code 2168
  unused code 2168
concepts
  locale settings 513
  module structure 462
  program structure 462
  understanding locale basics 516
conditional block
  WHILE loop 389
conditionnal block
  CASE block 381
  IF statement 386
configuration
  actions key labels 1775
  arabe 534
  collation / ordering 526
  date/time formatting 526
  database client locale 526
  environment variables 262
  FGLSERVER 1116
  Ming Guo date format 532
  Mingguo 532
  IPv6 39
  length semantics 521
  locale 519
  locale 519
  Ming Guo date format 532
  Mingguo 532
  n’ko 534
numeric formatting 526
reports 2064
semitic 534
terminal 1123
tifinar 534
urdu 534
configuration - apache for GWS
deploying web service server in GAS 3462
HTTP basic authentication 3462
HTTPS directives 3462
SSL/TLS root key 3461
SSL/TLS server authority list 3462
SSL/TLS server key 3461
configuration - FGLPROFILE
dbi.database.* 256
dbi.default.driver 256
Dialog.currentRowVisibleAfterSort 256
Dialog.fieldOrder 256
fglrun.arrayIgnoreRangeError 256
fglrun.decToCharScale2 256
fglrun.defaults 256
fglrun.floatToCharScale2 256
fglrun.ignoreDebuggerEvent 256
fglrun.ignoreLogoffEvent 256, 566
fglrun.localization.* 256
fglrun.mapAnyErrorToError 256
fglrun.mmapDisable 256
flm.* 256
gui.connection.timeout 256
gui.key.add_function 256
gui.programStoppedMessage 256
gui.protocol.format 256
gui.protocol.pingTimeout 256
gui.rendering 256
gui.server.autostart.* 256
mobile.environment.* 256
configuration - web services
FGLPROFILE settings 3489
configuration - web services FGLPROFILE
Basic or digest HTTP authentication
authenticate.idauth.login 3491
authenticate.idauth.password 3491
authenticate.idauth.realm 3491
authenticate.idauth.scheme 3491
HTTP
http.global.request.date 3495
HTTPS and password encryption
security.global.agent 3489
security.global.agent.gid 3489
security.global.ca 3489
security.global.ca.lookuppath 3489
security.global.certificate 3489
security.global.cipher 3489
security.global.keyssubject 3489
security.global.ocsp.enable 3489
security.global.ocsp.url 3489
security.global.privatekey 3489
security.global.protocol 3489
security.global.script 3489
security.global.systemca 3489
security.global.windowsca 3489
security.idsec.certificate 3489
security.idsec.keyssubject 3489
security.idsec.privatekey 3489
IPv6
ip.global.v6.interface.id 3493
ip.global.v6.interface.name 3493
ip.global.version 3493
Proxy
proxy.http.authenticate 3492
proxy.http.list 3492
proxy.http.location 3492
proxy.https.authenticate 3492
proxy.https.list 3492
proxy.https.location 3492
server
ws.idws.authenticate 3493
ws.idws.cipher 3493
ws.idws.security 3493
ws.idws.url 3493
ws.idws.verifyserver 3493
XML
xml.encryption.prefix 3494
xml.idxml.key 3494
xml.idxml.x509 3494
xml.keystore.caList 3494
xml.keystore.x509list 3494
xml.serializer.supportEmptyStrings 3494
xml.signature.prefix 3494
CONSTRUCT instruction
HELP attribute 1605, 1664
NAME attribute 1605, 1664
contact
mobile front call 2706
control block
AFTER DELETE 1548, 1623, 1679
AFTER DIALOG 1610
AFTER DISPLAY 1510, 1617, 1673
AFTER FIELD 1492, 1513, 1550, 1576, 1611, 1668
AFTER INPUT 1490, 1542, 1614, 1670
AFTER INSERT 1547, 1621, 1678
AFTER ROW 1511, 1545, 1619, 1675
BEFORE DELETE 1547, 1622, 1678
BEFORE DIALOG 1610
BEFORE DISPLAY 1509, 1617, 1673
BEFORE FIELD 1491, 1513, 1549, 1575, 1611, 1667
BEFORE INPUT 1489, 1541, 1613, 1669
BEFORE INSERT 1546, 1621, 1677
BEFORE MENU 1475, 1679
BEFORE ROW 1510, 1543, 1618, 1674
ON CHANGE 1492, 1549, 1575, 1612, 1668
ON ROW CHANGE 1544, 1619, 1675
conversion
BDL to JSON types 580
FGL to JSON types 581
JSON names 577
JSON Nulls and empty elements 578
JSON structures 576
json_name 577
json_null 578
copying
dynamic arrays 424
Cordova
  Cordova plugin API 3647
  GMA build 3649
  GMI build 3649
  Installing Cordova plugins 3645
  programming 3644, 3644
coverage tool
  basics 2168
  syntax 2168
  usage 2169
CREATE VIEW 708
create X.509 certificate
  WS client application configuration 3306, 3336, 3424

D

data block
  ON COLLAPSE 1508, 1607, 1666
  ON EXPAND 1508, 1607, 1666
  ON FILL BUFFER 1508, 1607, 1666
data conversions
  runtime stack 501
data type
  BIGINT 290
  BOOLEAN 293
  BYTE 290
  CHAR 293
  DATE 295
  DATETIME 296
  DECIMAL 298
  FLOAT 300
  INTEGER 300
  INTERVAL 301
  MONEY 303
  SMALLFLOAT 304
  SMALLINT 304
  STRING 305
  TEXT 307
  TINYINT 306
  VARCHAR 309
database
  client software 38
  IBM DB2 guide 758
  IBM Informix guide 742
  IBM Netezza guide 797
  login 664
  Microsoft SQL Server guide 827
  mobile apps 3583
  Oracle DB guide 908
  Oracle MySQL guide 876
  password 664
  PostgreSQL guide 964
  SAP ASE guide 1032
  SAP HANA guide 1069
  SQLite guide 1004
  user name 664
database client
  environment 658
  environment settings 266
database connection
  implicit 602
  multi-session 683
database driver
  dbmase 662
  dbmdh2 662
  dbmesm 662
  dbmfim 662
  dbmhdh 662
  dbmisx 662
  dbmmdb 662
  dbmmys 662
  dbmntz 662
  dbmodc 662
  dbmora 662
  dbmpgs 662
  dbmsnc 662
  dbmsqt 662
database schema
  file structure 470
database specification
  connection parameters 665
date/time specification
  connection parameters 665
date/time formatting 532
DB user
  CONNECT TO 679
  DATABASE 680
  DATABASE (callback) 680
  order of precedence 681
DB2 BOOLEAN 128
DB2 LUW
  database client environment 658
db2.prepare.deferred 128
DBCS 517
DBPRINT
  FGLSERVER 2034
debug
  mobile apps 3587
debugger
  basics 2142
  command reference 2147
  expressions 2147
  fgrun -d option 2143
  FGLSOURCEPATH 2143
  on mobile device 2145
  remote process 2143
  setting breakpoints 2146
debugger command
  bracktrace / where 2147
  break 2148
  call 2149
  clear 2149
  continue 2150
  delete 2150
  detach 2151
  disable 2151
  display 2151
  down 2152
  echo 2152
  enable 2153
  finish 2153
  frame 2154
  help 2154
  ignore 2154
  info 2155
list 2156
next 2156
output 2157
print 2157
ptype 2158
quit 2158
run 2158
set 2159
signal 2160
source 2159
step 2160
tbody 2161
undisplay 2162
until 2162
up 2163
watch 2163
what is 2164
debugging
  SQL statements 603
text mode 1130
Decimal to string conversion 170
Declare prepared
  SQL performances 653
default action views
  Android default decoration 1788
  Android rendering - action bar 1785
  Android rendering - FAB 1787
  iOS default decoration 1784
  iOS rendering 1779
  on mobile 1777
  Universal Rendering 1769
default driver
database 664
DEFAULT SAMPLE
  in form file 1350
definition - arrays
  DYNAMIC ARRAY 419
  ARRAY [size] 419
definition - constants
  CONSTANT 406
definition - dictionary
  DICTIONARY 427, 457
definition - functions 437
definition - records
  DEFINE ... RECORD 409
definition - reports 2039
definition - user types
  TYPE 432
definition - variables
  DEFINE 390
DELIMITERS
  in form file 1350
demos 28
deploying
  mobile apps 3588
deploying mobile apps
  Android (GMA) 3589
  iOS (GMI) 3604
  mobile apps on server 3616
deprecated 211
desupported
  database drivers removed in 1.33 200
  database drivers removed in 2.00 194
  database drivers removed in 2.40 173
  database drivers removed in 2.50 169
  database drivers removed in 2.51 162
  Informix database drivers removed in 2.00 193
development environment
  installing mobile clients 3568
  mobile development client for Android 3568
  mobile development client for iOS 3569
Dialog API
  action identification 2555
  current dialog 2554
  DIALOG 2554
  field identification 2556
  passing DIALOG object 2554
  screen-array identification 2557
DIALOG instruction
  FIELD ORDER FORM attribute 1602
  UNBUFFERED attribute 1602
dialog programming
  action binding 1765
  action deactivation 1763
  action handlers (ON ACTION) 1761
  action hiding 1764
  action view binding 1762
  action views 1759
  actions 1744
  autocompletion 1733
  automatic actions 1740
  close action 1772
COMBOBOX fields 1735
context menu configuration 1771
data validation 1762
field actions 1767
field changes 1724
field configuration 1718
field data type 1714
field disabling 1728
field dropdown window 1737
field initialization 1722
field input length 1716
field modification flag 1723
focus detection 1730
form validation rules 1727
INCLUDE 1727
input started 1725
interaction triggers 1732
NOT NULL 1727
predefined actions 1740
REQUIRED 1727
setting focus 1731
special actions 1740
sub-dialog identification 1594, 1729
tabbing order 1729
unbuffered mode 1720
variable to field binding 1715
window style position field 1737
dialog types
  dynamic dialogs 1707
  modal dialog 1707
parallel dialog 1707
dialogs
timeout trigger 1713
dialogtouched
dialog programming 1725
dictionary
  passed as parameter 497
  returned from functions 500
direct specification
database 667
DISPLAY ARRAY
  array variable binding 1831
  controlling number of rows 1834
DISPLAY ARRAY instruction
  ACCESSORYTYPE attribute 1603, 1662
  COUNT attribute 1603, 1662
  DETAILACTION attribute 1603, 1662
  DOUBLECLICK attribute 1603, 1662
  FOCUSONFIELD attribute 1603, 1662
  HELP attribute 1603, 1662
Doc conventions 25
documenting
  sources and programs 2126
DOM 573
double-byte 517
DOUBLECLICK
dialog programming 1872
dynamic arrays
  passed as parameter 497
  returned from functions 500
Dynamic Dialogs
  field definition array 2558
Dynamic SQL
  SQL performances 651
E
editing
  source files 2123
editing sources
  character set 2123, 2123
  Code completion (vim) 2123
  locale 2123, 2123
  Source highlighting (vim) 2123
  Source indentation (vim) 2123
email
  mobile front call 2701
environment
  variables 262
environment variable
  DBCENTURY 266
  DBDATE 267
  DBDELIMITER 268
  DBEDIT 268
  DBFORMAT 268
  DBMONEY 271
  DBPATH 272
  DBPRINT 272
  DBSCREENDUMP 272
  DBSCREENOUT 273
  DBTEMP 273
  DYLD_LIBRARY_PATH 264
  FGL_LENGTH_SEMANTICS 273
  FGLAPPPDIR 273
  FGLAPPSERVER 274
  FGLCOV 274
  FGLDBPATH 274
  FGLDIR 275
  FGLGBCDIR 275
  FGLGUI 275
  FGLGUIDEBUG 275
  FGLIMAGEPATH 276
  FGLLDPATH 278
  FGLPROFILE 279
  FGLRESOURCESPATH 279
  FGLSERVER 280
  FGLSOURCEPATH 280
  FGLSQLDEBUG 281
  FGLTRACE_EXCLUDE 282
  FGLTRACE_FUNCTIONS 282
  FGLWRTUMASK 283
  FGLWSDEBUG 283
  GMIDIR 284
  INFORMIXTERM 284
  LANG 264
  LC_ALL 264
  LD_LIBRARY_PATH 264
  LD_LIBRARY_PATH_64 264
  LIBPATH 264
  PATH 264
  SHLIB_PATH 264
  TERM 264
  TERMINFO 265
  TERMINO 265
  TMPDIR / TEMP / TMP 266
environment variables
  mobile apps 3571
  operating system 264
error -8063 1118
error handling
  exceptions 502
errors
  overflow 317
  type conversion 317
examples
  action defaults files (.4ad) 1164
  arrays 426
  base.Channel 2430
  base.MessageServer 2470
  base.SqlHandle 2445
  base.StringBuffer 2461
  base.StringTokenizer 2465
c-extensions 2246
Canvas 2017
com.HTTPPart 2910
com.HTTPRequest 2891
constants 408
CONSTRUCT 1583
DIALOG (declarative dialogs) 1699
DIALOG (multiple dialogs) 1644
DIALOG (parallel dialogs) 1705
dictionary 430
DISPLAY ARRAY 1526
drag & drop 1921
exception handling 510
fglgar 2191
form definition files (.per) 1414
functions 456
Genero Web Components
   fglgallery 1985
   fglrichtext 1978
   fglsvcanvas 1996
GICAPI Web Component 1954
globals 554
INPUT 1499
INPUT ARRAY 1560
insert cursors 735
interface 460
java interface 2225
localized strings 550
MENU 1481
message files (.msg) 1162
OAuthAPI
   mobile app 3521
om.DomDocument 2584
om.DomNode 2605
om.NodeList 2608
om.SaxAttributes 2616
om.SaxDocumentHandler 2623
om.XPath 2630
om.XmlWriter 2634
os.Path 2809
presentation styles (*.4st) 1234
profiler 2167
PROMPT 1467
records 417
scrollgrid views 1888
security.Digest 3147
SELECT ... FOR UPDATE 729
SOAP web services server
   complete server application 3348
   server application using third-party WSDL 3352
split views 1908
startmenus 2022
table views 1879
toolbars 1443
topmenus 1451
tree views 1897
ui.ComboBox 2569
ui.Dialog 2559
ui.Form 2511
ui.Interface 2488
ui.Window 2496
URL-based Web Component 1931
user defined types 434
util.JSON 2773
variables 404
web services
   FGLPROFILE (HTTPS + Proxy, HTTP + Proxy Authentication) 3496
   FGLPROFILE (XML crypto + fglpass) 3497
   UNIX password script (shell) 3496
   Windows password script (.BAT) 3496
   WS FGLPROFILE (wildcards in URL base) 3498
web services client
   BDL stub example 3331
WHERE CURRENT OF 729
XML reports 2030
xml.CryptoKey 3066
xml.CryptoX509 3075
xml.domdocument
   CDATASECTION 2962
   load and save methods 2963
   navigation methods 2960
   node creation methods 2961
xml.DomDocument 2967
xml.DomNode 2996
xml.Encryption 3114
xml.Signature 3099
xml.StaxReader 3032
xml.StaxWriter 3012
xml.XSLTTransformer 3124
examples - web services
   APNs feedback handler 2930
   APNs push notifications provider 2929
   Apple push notifications 2929
   com.APNS 2929
execution
   programs 488
execution trace
   basics 2171
   syntax 2172
   usage 2172
export
   SQL data 740
expressions
   boolean 331
date 335
datetime 335
integer 333
interval 336
numeric 334
string 334
extension class
   os.Path 2792
   util.Date 2726
   util.DateTime 2728
   util.Integer 2735
   util.Interval 2750
   util.JSON 2766
   util.JSONArray 2783
   util.JSONObject 2773
   util.Math 2759
   util.Strings 2752
extension class - http
   com.HTTPPart 2902
   com.HTTPRequest 2871
   com.HTTPResponse 2894
extension class - security
   security.Base64 3129
   security.BCrypt 3150
   security.Digest 3142
   security.HexBinary 3136
   security.PBKDF2 3148
   security.RandomGenerator 3127
extension class - web service
   APNs SSL/TLS certificate 2923
extension class - web services
  Apple push notifications 2923
  com.APNS 2923
  com.HTTPServiceRequest 2845
  com.TCPRequest 2912
  com.Util 2922
  com.WebOperation 2821
  com.WebServiceEngine 2828
  com.WebServices 2811
extension class - XML
  xml.CryptoKey 3047
  xml.CryptoX509 3068
  xml.DomDocument 2931
  xml.DomNode 2969
  xml.DomNodeList 2997
  xml.Encrytion 3104
  xml.KeyStore 3117
  xml.Serializer 3033
  xml.Signature 3076
  xml.StaxReader 3012
  xml.StaxWriter 2999
  xml.XSLTTransformer 3118
extract
database schema 476

F
FAQ 250
feature changes
  version list 109
features - xml.DomDocument 2964
feInfo - options
  dataDirectory 2651
  deviceId 2651
  deviceModel 2651
  dictionariesDirectory 2651
  feName 2651
  fePath 2651
  freeStorageSpace 2651
  iccid 2651
  imei 2651
  ip 2651
  numScreens 2651
  osType 2651
  osVersion 2651
  ppi 2651
  screenResolution 2651
  target 2651
  userPreferredLang 2651
  windowSize 2651
FGL_SQLDEBUG() 603
FGL-1670 175
FGL-2534 169
FGL-2647 171
FGL-3915 170
FGL-5165 127
FGL-5321 128, 128
FGL-5361 128
fgldbsch
database schema 476
fgldbsch (PostgreSQL namespace) 171
FGLGUIDEBUG
debugging 1120
FGLPROFILE
  aggregateZero 2064
  ase.logintime 678
  ase.prefetch.rows 678
  db2.schema 672
  dbi.default.userauth.callback 680
dialog configuration 1708
esm.logintime 677
  esm.prefetch.rows 677
  ftm.logintime 677
  ftm.prefetch.rows 677
  hdb.schema 678
  IBM DB2 parameters 672
  ifxemuil 668
  Informix emulation parameters 668
  MariaDB parameters 675
  mdb.config 675
  mys.config 675
  MySQL parameters 675
  ora.date.ifxfetch 673
  ora.logintime 673
  ora.prefetch.memory 673
  ora.prefetch.rows 673
  ora.schema 673
  ora.sid.command 673
  ora.temptables.private.prefix.command 673
  ora.temptables.private.prefix.source 673
  ora.temptables.private.prefix.value 673
  ora.temptables.schema.command 673
  ora.temptables.tablespace 673
Oracle DB parameters 673
  pgs.prefetch.tables 676
  pgs.schema 676
  PostgreSQL parameters 676
  SAP ASE parameters 678
  SAP HANA parameters 678
  snc.logintime 676
  snc.prefetch.tables 676
SQL Server Easysoft parameters 677
SQL Server FreeTDS parameters 677
SQL Server MS sODBC parameters 676
WS client application
  HTTPS server authority list 3308, 3338, 3426
  HTTPS server configuration 3309, 3339, 3427
security.global.ca 3308, 3338, 3426
  security.global.ca.lookuppath 3308, 3338, 3426
  security.idsec.certificate 3308, 3338, 3426
  security.idsec.privatekey 3307, 3337, 3425
  security.idsec.privatekey 3407, 3425
FGLSCREENOUT
debugging 1130
FIELD desupported 164
FIELD ORDER FORM
dialog programming 1729
FIELD_TOUCHED()
dialog programming 1723
file format
FGLPROFILE 255
localized string files (.str) 539
start menu files (.4sm) 2019
file format - database schema
column functional attributes (.val) 474
column types (.sch) 470
column video attributes (.att) 475
FLOAT/SMALLFLOAT to string conversion 127
focus
dialog programming 1730
font
fontFamily 1178
fontSize 1179
fontStyle 1180
fontWeight 1180
fonts
presentation styles 1178
foreground mode
mobile apps 3573
Form API
AUI element identification 2510
form inclusion 1265
formatting
DATE 321
DATETIME 322
INTERVAL 324
numeric 318
Four Js BDL
migration 231
front call
ui.Interface.frontCall() 583, 2473
front call - android
askForPermission 2714
showAbout 2715
showSettings 2715
startActivity 2716
startActivityForResult 2717
front call - cordova
call 2721
callWithoutWaiting 2722
getAllCallbackData 2723
getCallbackData 2725
getCallbackDataCount 2724
getPluginInfo 2725
listPlugins 2726
front call - ios
getBadgeNumber 2719
setBadgeNumber 2719
front call - localStorage
clear 2697
get 2696
keys 2697
removeItem 2697
setItem 2696
front call - mobile
chooseContact 2699
choosePhoto 2699
chooseVideo 2700
composeMail 2701
composeSMS 2701
connectivity 2702
geoLocation 2703
importContact 2706
isForeground 2707
newContact 2708
runOnServer 2710
scanBarcode 2711
takePhoto 2712
takeVideo 2712
front call - mobilepush notification
get notification data 2703
getRemoteNotifications 2703
register the device 2708
registerForRemoteNotifications 2708
unregister the device 2713
unregisterFromRemoteNotifications 2713
front call - monitor
GDC auto-update 2668
update 2668
front call - standard
cbAdd 2649
cbClear 2649
cbGet 2649
cbPaster 2650
cbSet 2650
clearFileCache 2650
execute 2651
fInfo 2651
getEnv 2656
getWindowId 2656
hardCopy 2656
launchURL 2657
mdClose 2658
openDir 2658
openFile 2659
openFiles 2660
playSound 2662
saveFile 2662
setReportFont 2663
setReportPrinter 2664
setWebComponentPath 2665
shellExec 2665
front call - theme
gGetCurrentTheme 2692
listThemes 2694
setTheme 2691
front call - webcomponent
call 2666
frontCallAPIVersion 2667
setTitle 2668
front call module
android 2713
cordova 2720
Genero Desktop Client 2668
ios 2718
localStorage 2695
mobile 2697
monitor 2668
standard 2647
theme 2691
webcomponent 2666
front calls
mobile apps 3581
front-end
  --gui-listen 1116
  connection 1116
front-end connection
  gui.connection.timeout 1118
  gui.protocol.pingTimeout 1118
front-end errors 1119
front-end protocol
  --run-guilog 1120
  --start-guilog 1120
  compression 1119
  debugging 1120
  replay for front-end 1120
front-end startup 1121
function
  calling 442
  calling a function 377
  local variable 441
  parameters 444
  references 449
  returning values 380, 446
  scope 440
FUNCTION 437
functions
  attributes 455
  dialog control 1709
  methods 452

G

GBC system colors
  presentation styles 1177
GDC system colors
  presentation styles 1176
general 25
General terms 27
Genero BDL
  concepts 30
  environment variables 266
  introduction 28
  overview 28
  syntax features 285
Genero web client hybrid mode
  software requirements 37
getCurrentItem()
  dialog programming 1730
getFieldTouched()
  dialog programming 1723
gICAPI WEBCOMPONENT
  action handling 1951
  deploying - direct mode 1948
  deploying - on mobile 1949
  deploying - summary 1947
  deploying - using GAS 1948
  form layout 1950
  gICAPIAction 1935
  gICAPIData 1937
  gICAPIFlushData 1938
  gICAPIFocus 1939
  gICAPIProperty 1940
  gICAPISetStateChanged 1944
  gICAPIData 1945
gICAPISetFocus 1946
  HTML document 1933
  image resources 1952
  interface script 1933
  onICHostReady 1933
  PROPERTIES (WEBCOMPONENT) 1940

GPS location
  mobile front call 2703
graphical mode 1110
grid-based layout
  automatic hbox vbox 1422
  empty grid cell size 1423
  form item dependency 1423
  grid positions example 1426
  hbox tags 1428
  packed grids 1420
  widget position and size 1423
GSM, Edge, 3G, 4G, Wifi
  mobile front call 2702
gui.server.autostart 1121
GUID 3128

H

help 28
high-level REST
  attributes
  context 3480
  error handling 3483
  file handling 3477
  HTTP return code 3479
  name, description, optional 3474
  operations 3465
  query, cookie, header 3471
  security handling 3486
  templating 3468
  WSAattachment 3477
  WSContext 3481
  WSCookie 3473
  WSDelete 3466
  WSDescription 3474
  WSError 3483
  WSGet 3467
  WSHeader 3472
  WSInfo 3488
  WSMedia 3478
  WSNamespace 3475
  WSOptional 3476
  WSParam 3469
  WSPath 3469
  WSPost 3467
  WSPut 3468
  WSSet 3471
  WSRefCode 3480
  WSScope 3486
high-level REST API
  attributes
    WSThrows 3484
how to
  web services 3445
howto
  web services 3359
HP-UX
  JNI Java Native Interface 2194
HTTPS configuration
  certificate authorities 3305
  GWS implicit client certificate 3305
HTTPS root certificate authority
  WS client application configuration 3306, 3336, 3424
HTTPS server
  client app configuration 3305, 3335, 3423

IBM AIX
  JNI Java Native Interface 2194
IBM DB2
  WHERE CURRENT OF 793
IBM DB2 LUW
  BIGINT 769
  BOOLEAN 767
  CHAR / VARCHAR 767
  CHECK 779
  CLOB / BLOB 778
  Constraints 779
  CREATE TEMP TABLE 785
  Data type conversions 766
  Database concepts 761, 793
  Database object names 780
  Database privileges 765
  Database users 765
  DATE 771
  DATETIME 771
  DECIMAL 769
  For update 762
  FOREIGN KEY 779
  GRAPHIC / VARGRAPHIC 767
  INSERT cursor 791
  INTEGER 769
  INTERVAL 773
  Isolation level 762
  LENGTH() 790
  LOAD / UNLOAD 794
  Lock mode 762
  MATCHES / LIKE 788
  MONEY 769
  NOT NULL 779
  OUTER join 782
  PRIMARY KEY 779
  Reserved words 782
  row limiting 790, 867
  ROWID 777
  SCROLL CURSOR 795
  SELECT ... FOR UPDATE 792
  SELECT ... INTO TEMP 785
  SELECT * (star) 789
  SERIAL / BIGSERIAL 774
  Single row SELECT 788
  SMALLINT 769
  SQL interrupt 794
  Storage concepts 761
  String delimiters 787
  Substrings 786
  TEXT / BYTE 778

Transactions 764, 784
  UNIQUE 779
  WITH HOLD 792

IBM Informix
  :: cast operator 757
  BIGSERIAL / SERIAL8 749
  CLOB / BLOB 754
  CREATE INDEX 752
  DataBlade modules 752
  DISTINCT 750
  LIST data type 755
  LVARCHAR 750
  MULTISET data type 755
  NCHAR / NVARCHAR 749
  OPAQUE data types 751
  Optimizer directives 751
  ROW data type 756
  SET data type 756
  Static SQL syntax in BDL 753
  Stored procedures 750
  Supported features 744
  Supported versions 742
  Table inheritance 757
  Triggers 751
  UTF-8 databases 745
  XML support 751

IBM Netezza
  BIGINT 805
  BOOLEAN 803
  CHAR / VARCHAR 804
  CHECK 812
  CLOB / BLOB 812
  Constraints 812
  CREATE INDEX 812
  CREATE TEMP TABLE 817
  Database concepts 798
  Database object names 813, 819
  Database type conversions 802
  Database users 801
  DATE 807
  DATETIME 807
  DECIMAL 805
  FOREIGN KEY 812
  INSERT cursors 823
  INTEGER 805
  INTERVAL 809
  ISOLATION LEVEL 798
  LENGTH() 819
  LOAD / UNLOAD 825
  LOCK MODE 798
  MATCHES / LIKE 821
  MONEY 805
  NCHAR / NVARCHAR 804
  NOT NULL 812
  OUTER join 814
  PRIMARY KEY 812
  Reserved words 814
  row limiting 822
  ROWID 811
  SCROLL CURSOR 826
  SELECT ... FOR UPDATE 798, 824
  SELECT ... INTO TEMP 817
SERIAL / BIGSERIAL 810
SMALLINT 805
SQL interrupt 826
String delimiters 820
Substrings 818
TEXT / BYTE 812
Transactions 800, 816
UNIQUE 812
UPDATE limitations 823
WHERE CURRENT OF 825
WITH HOLD 824

identifying
form items 1250
image resources
  FGLIMAGEPATH 1149
  FontAwesome 1149
  icon color 1149
  image URLs 1149
  image2font 1149
  TTF 1149
images and icons
  mobile apps 3575
import
  SQL data 736
importing
  modules 480
indirect specification
database 667
Informix
database client environment 658
informix 4GL
  migration 211
INPUT ARRAY
  array variable binding 1831
  controlling number of rows 1834
INPUT ARRAY instruction
  APPEND ROW attribute 1603, 1663
  AUTO APPEND attribute 1603, 1663
  COUNT attribute 1603, 1663
  DELETE ROW attribute 1603, 1663
  HELP attribute 1603, 1663
  INSERT ROW attribute 1603, 1663
  MAXCOUNT attribute 1603, 1663
INPUT instruction
  HELP attribute 1602, 1662
  NAME attribute 1602, 1662
  WITHOUT DEFAULTS attribute 1602, 1662
Install
  GMA 40
  GMI 42
  GMI multiple version 42
installation
  IBM DB2 LUW client 759
  IBM DB2 LUW server 758
  IBM Informix client 744
  IBM Informix DB 742
  IBM Informix server 742
  IBM Netezza client 797
  IBM Netezza server 797
  Microsoft SQL Server client 828
  Microsoft SQL Server DB 827
  Microsoft SQL Server Native Client 828
  Oracle DB client 910
  Oracle DB server 908
  Oracle MySQL/MariaDB client 877
  Oracle MySQL/MariaDB server 876
  PostgreSQL client 965
  PostgreSQL server 964
  SAP ASE 1032, 1035
  SAP HANA client 1070
  SAP HANA server 1069
  SQLite database 1004
  SQLite library 1005
installing
  Genero BDL 39
instruction
dialog instructions reference 1452
  EXIT PROGRAM 492
instruction - C Extensions
  IMPORT 481
instruction - classes
  DEFINE ... package class 570
instruction - CONSTRUCT
  ACCEPT CONSTRUCT 1579
  CONTINUE CONSTRUCT 1580
  EXIT CONSTRUCT 1580
instruction - database schema
  SCHEMA 468
instruction - debugging
  BREAKPOINT 493
instruction - dialog
  CLEAR field-list 1496, 1560, 1583, 1639, 1694
  DISPLAY TO / DISPLAY BY NAME 1639, 1694
  NEXT FIELD 1497, 1557, 1580, 1636, 1692
instruction - DIALOG
  ACCEPT DIALOG 1640, 1696
  CANCEL DIALOG 1641, 1697
  CONTINUE DIALOG 1639, 1695
  EXIT DIALOG 1640, 1695
instruction - dialogs
  CONSTRUCT 1566
  DIALOG (declarative) 1649
  DIALOG (procedural) 1588
  DISPLAY ARRAY 1502
  INPUT 1482
  INPUT ARRAY 1532
  MENU 1468
  PROMPT 1462
  START DIALOG 1702
  TERMINATE DIALOG 1703
instruction - display
  CLEAR field-list 1460
  CLEAR FORM 1458
  CLEAR SCREEN 1139
  CLEAR SCREEN ARRAY 1459
  DISPLAY (to stdout) 1453
  DISPLAY AT 1139
  DISPLAY BY NAME 1457
  DISPLAY TO 1455
  ERROR 1454
  MESSAGE 1453
  SCROLL screen-array 1461
instruction - DISPLAY ARRAY
  ACCEPT DISPLAY 1526
CONTINUE DISPLAY 1526
EXIT DISPLAY 1526

instruction - exceptions
TRY / CATCH 506
WHENEVER 503

instruction - flow control
CALL 377
CASE 381
CONTINUE CONSTRUCT 382
CONTINUE DIALOG 382
CONTINUE FOR 382
CONTINUE FOREACH 382
CONTINUE INPUT 382
CONTINUE MENU 382
CONTINUE WHILE 382
EXIT CONSTRUCT 383
EXIT DIALOG 383
EXIT DISPLAY 383
EXIT FOR 383
EXIT FOREACH 383
EXIT INPUT 383
EXIT MENU 383
EXIT REPORT 383
EXIT WHILE 383
FOR 384
GOTO 385
IF 386
LABEL 388
RETURN 380
SLEEP 388
WHILE 389

instruction - forms
CLOSE FORM 1139
DISPLAY FORM 1138
OPEN FORM 1137

instruction - functions
FUNCTION 451

instruction - INPUT
ACCEPT INPUT 1495
CONTINUE INPUT 1496
EXIT INPUT 1496

instruction - INPUT ARRAY
ACCEPT INPUT 1554
CANCEL DELETE 1555, 1642, 1697
CANCEL INSERT 1556, 1642, 1698
CONTINUE INPUT 1557
EXIT INPUT 1555

instruction - java
IMPORT JAVA 487

instruction - MENU
CONTINUE MENU 1480
EXIT MENU 1480
SHOW OPTION / HIDE OPTION 1479

instruction - messages
OPTIONS HELP FILE 1161

instruction - modules
IMPORT FGL 481

instruction - programs
RUN 490

instruction - records
THRU 417

instruction - reports
EXIT REPORT 2052
NEED 2056
PAUSE 2056
PRINT 2052
PRINTX 2055
SKIP 2057

instruction - SQL
ALTER INDEX 707
ALTER SEQUENCE 704
ALTER TABLE 706
BEGIN WORK 689
CLOSE (insert cursor) 734
CLOSE (result set cursor) 723
CLOSE DATABASE 683
COMMIT WORK 691
CONNECT TO 683
CREATE INDEX 707
CREATE SEQUENCE 704
CREATE SYNONYM 708
CREATE TABLE 705
CREATE VIEW 708
DATABASE 682
DECLARE (insert cursor) 732
DECLARE (result set) 717
DECLARE (SELECT ... FOR UPDATE) 728
DELETE 700
DELETE ... WHERE CURRENT OF 729
DISCONNECT 686
DROP INDEX 707
DROP SEQUENCE 705
DROP SYNONYM 708
DROP TABLE 707
DROP VIEW 708
EXECUTE 711
EXECUTE IMMEDIATE 713
FETCH (result set cursor) 721
fglhint comments 714
FLUSH (insert cursor) 734
FOREACH 724
FREE (insert cursor) 735
FREE (result set cursor) 723
FREE (SQL statement) 712
INSERT 696
LOAD 736
OPEN (insert cursor) 733
OPEN (result set cursor) 720
PREPARE 710
PUT (insert cursor) 733
RECOVER TABLE 709
RELEASE SAVEPOINT 692
RENAME 709
ROLLBACK WORK 691
SAVEPOINT 690
SELECT 700
SET CONNECTION 685
SET ISOLATION 692
SET LOCK MODE 693
SET SESSION AUTHORIZATION 685
SQL block 702
UNLOAD 740
UPDATE 698
UPDATE ... WHERE CURRENT OF 728
UPDATE STATISTICS 687

instruction - variables
FREE (TEXT/BYTE) 401
INITIALIZE 399
LET 402
LOCATE 400
VALIDATE 403

instruction - windows
CLEAR WINDOW 1136
CLOSE WINDOW 1135
CURRENT WINDOW 1135
OPEN WINDOW 1131

INT_FLAG
   interruption 1711

interaction block
   AFTER CONSTRUCT 1574, 1616, 1672
   BEFORE CONSTRUCT 1573, 1615, 1671
   COMMAND "option" 1476
   COMMAND KEY 1477
   ON ACTION 1478, 1493, 1514, 1551, 1577, 1623, 1680
   ON APPEND 1516, 1627, 1682
   ON DELETE 1519, 1629, 1685
   ON DRAG_ENTER 1522, 1633, 1688
   ON DRAG_FINISHED 1522, 1632, 1688
   ON DRAG_OVER 1523, 1634, 1689
   ON DRAG_START 1521, 1631, 1687
   ON DROP 1525, 1635, 1691
   ON IDLE 1465, 1494, 1514, 1551, 1577, 1624, 1681
   ON INSERT 1517, 1628, 1683
   ON KEY 1465, 1494, 1515, 1552, 1578, 1625, 1681
   ON SELECTION CHANGE 1520, 1630, 1686
   ON SORT 1520, 1553, 1630, 1686
   ON TIMER 1466, 1479, 1495, 1516, 1554, 1579, 1625, 1682
   ON UPDATE 1518, 1628, 1684

interaction block - DIALOG
   COMMAND "option" 1626

internationalization
   application locale 512
   localized strings 538

introduction
   action default (.4ad) 1162
   form files (.per) 1237
   Genero BDL 30
   message files (.msg) 1160
   presentation styles (.4st) 1165
   toolbars 1436
   topmenus 1444

introspection
   user preferred language 533

iOS 2194
iOS apps (GMI)
   building IPAs 3606
   directory structure 3604
   Info.plist 3606

iOS style attributes
   mobile apps 3582
   navigationbar colors 3582
   tabbar colors 3582
   toolbar colors 3582

J
Java
   runtime requirements 38
Java classes
      importing 487
java interface
      BDL to java data type mapping 2200
      BDL types in java code 2209
      calling java class methods 2196
      calling java object methods 2197
      case sensitivity 2198
      character set 2215
      creating java objects 2196
      defining java object reference 2196
      exception handling 2219
      formatting data in java code 2214
      garbage collection 2199
      ignorable returns 2199
      IMPORT JAVA class 2196
      java with GMA / Android 2219
      JVM options 2197
      locale 2215
      method overloading 2198
      packaging java for GMA 2223
      passing objects to functions 2198
      passing varargs 2217
      prerequisites 2192
      returning objects from methods 2199
      static java class fields 2200
      using BDL records in java code 2212
      using CAST operator 2218
      using INSTANCEOF operator 2218
      using java arrays 2215

Java interface
   FglByteBlob 2206
   FglDate 2201
   FglDateTime 2202
   FglDecimal 2205
   FglTextBlob 2205

Java runtime
   requirements 38
   JNI Java Native Interface 2194

K
KEEP CURRENT ROW attribute 1603, 1663
keep-alive 1118
keyboard type
   mobile apps 3579
keyboardHint
   mobile apps 3579

L
label
   goto 385
layout
   images 1148
library
   built-in functions 2268
built-in packages 2378
extension packages 2726
front calls 583, 2635
utility functions 2301
limitations
mobile apps 3571
linking
libraries (fgllink) 2113
programs (fgllink) 2115
list
DOUBLECLICK 1838
current row 1835
row choice 1838
rowActionTrigger 1838
visual / array index 1835
list dialogs
controlling a list 1831
list views
mobile apps 3579
literals
datetime literals 328
integer literals 325
interval literals 329
MDY() literals 328
numeric literals 326
text literals 326
LOAD/UNLOAD
SQL portability 639
local actions 1742
localization
device functions 3586
mobile apps 3572
locks (SQL) 624
log errors 508
login
database 679
Long transactions
SQL performances 652
loop
FOR loop 384
SQL result set 724

M
Mac OS X
JNI Java Native Interface 2194
MariaDB
database client environment 659
MBCS 517
MDI
window containers 2023
methods - base.Application
getArgument() 2407
getArgumentCount() 2408
getFglDir() 2408
getProgramDir() 2408
getProgramName() 2409
getResourceEntry() 2409
getStackTrace() 2409
isMobile() 2410
reloadResources() 2411
methods - base.Channel
close() 2414
create() 2414
dataAvailable() 2414
isEof() 2416
openClientSocket() 2416
openFile() 2417
openPipe() 2418
openServerSocket() 2418
read() 2419
readLine() 2420
readOctets() 2421
setDelimiter() 2422
stderr output 2417
write() 2423
writeLine() 2423
writeNoNL() 2424
methods - base.MessageServer
connect() 2469
send() 2469
methods - base.SqlHandle
close() 2434
create() 2434
execute() 2434
fetch() 2435
fetchAbsolute() 2435
fetchFirst() 2436
fetchLast() 2436
fetchPrevious() 2437
fetchRelative() 2437
flush() 2438
getResultCount() 2438
getResultName() 2439
getResultType() 2439
getResultValue() 2440
open() 2441
openScrollCursor() 2441
prepare() 2442
put() 2442
setParameter() 2443
methods - base.StringBuffer
append() 2450
clear() 2450
create() 2450
equals() 2451
equalsIgnoreCase() 2452
getCharAt() 2452
getIndexOf() 2453
getLength() 2454
insertAt() 2454
replace() 2455
replaceAll() 2455
subString() 2456
toLowerCase() 2457
toString() 2457
toUpperCase() 2458
trim() 2458
trimLeft() 2459
trimLeftWhiteSpace() 2460
trimRight() 2460
trimRightWhiteSpace() 2461
trimWhiteSpace() 2459
| Index | 3669 |

methods - base.StringTokenizer
- countTokens() 2464
- create() 2463
- createExt() 2463
- hasMoreTokens() 2464
- nextToken() 2465

methods - base.TypeInfo
- create() 2466
- describe() 2468

methods - BYTE
- readFile() 2379
- writeFile() 2379

methods - com.APNS
- DecodeError() 2925
- DecodeFeedback() 2926
- EncodeMessage() 2927

methods - com.HTTPPart
- clearHeaders() 2906
- CreateAttachment() 2905
- CreateFromData() 2905
- CreateFromDomDocument() 2905
- CreateFromString() 2904
- getAttachment() 2907
- getContentAsData() 2907
- getContentAsDomDocument() 2908
- getContentAsString() 2908
- getHeader() 2908
- removeHeader() 2909
- setHeader() 2909

methods - com.HTTPRequest
- addPart() 2874
- beginXmlRequest() 2875
- clearAuthentication() 2875
- clearHeaders() 2876
- Create() 2876
- doDataRequest() 2876
- doFileRequest() 2877
- doFormEncodedRequest() 2878
- doRequest() 2879
- doTextRequest() 2879
- doXmlRequest() 2879
- endXmlResponse() 2896
- getAsyncResponse() 2881
- getResponse() 2882
- methods 2871
- removeHeader() 2883
- setAuthentication() 2883
- setAutoCookies() 2884
- setAutoReply() 2885
- setCharset() 2885
- setConnectionTimeout() 2886
- setHeader() 2886
- setKeepConnection() 2887
- setMaximumResponseLength() 2887
- setMethod() 2888
- setMultipartResponseType() 2888
- setProxy() 2889
- setProxyAuthentication() 2889
- setTimeout() 2890
- setVersion() 2891

methods - com.HTTPResponse
- beginXmlResponse() 2896
- getAsyncResponse() 2896
- getFileResponse() 2897
- getHeader() 2898
- getHeaderCount() 2898
- getHeaderName() 2898
- getHeaderValue() 2899
- getMultipartType() 2899
- getPart() 2900
- getPartCount() 2899
- getPartFromContentID() 2900
- getServerCookies() 2900
- getStatusCode() 2901
- getStatusDescription() 2901
- getTextResponse() 2902
- getXmlResponse() 2902

methods - com.HTTPServiceRequest
- addResponsePart() 2850
- beginXmlRequest() 2850
- beginXmlResponse() 2851
- endXmlRequest() 2851
- endXmlResponse() 2852
- examples 2870
- findRequestCookie() 2852
- getMethod() 2853
- getRequestHeader() 2854
- getRequestHeaderCount() 2854
- getRequestHeaderValue() 2855
- getRequestMultipartType() 2855
- getRequestPart() 2856
- getRequestPartCount() 2856
- getRequestPartFromID() 2857
- getRequestVersion() 2853
- setURL() 2857
- getURLHost() 2858
- getURLPath() 2858
- getURLPort() 2858
- getURLQuery() 2859
- hasRequestKeepConnection() 2859
- readDataRequest() 2860
- readFileRequest() 2860
- readFormEncodedRequest() 2861
- readTextRequest() 2862
- readXmlRequest() 2862
- sendDataResponse() 2863
- sendFileResponse() 2863
- sendResponse() 2865
- setResponseCharset() 2867
- setResponseCookies() 2867
- setResponseHeader() 2868
- setResponseMultipartType() 2869
- setResponseVersion() 2869

methods - com.TCPRequest
- beginXmlRequest() 2914
- Create() 2915
- doDataRequest() 2915
- doRequest() 2915
- doTextRequest() 2917
- doXmlRequest() 2916
endXmlRequest()  2917
getAsyncResult()  2918
getResponse()  2917
setConnectionTimeOut()  2919
setKeepConnection()  2919
setMaximumResponseLength()  2919
setTimeout()  2918
methods - com.TCPResponse
beginXmlResponse()  2920
dataXmlResponse()  2920
dataTextResponse()  2921
dataXmlResponse()  2922
methods - com.Util
UniqueApplicationInstance()  2922
methods - com.WebOperation
addFault()  2823
addInputHeader()  2824
addOutputHeader()  2824
CreateDOCStyle()  2825
CreateOneWayDOCStyle()  2825
CreateOneWayRPCStyle()  2826
CreateRPCStyle()  2825
initiateSession()  2826
setComment()  2826
setInputAction()  2827
setInputEncoded()  2827
setOutputAction()  2827
setOutputEncoded()  2828
methods - com.WebService
createFault()  2813
createHeader()  2814
CreateStatefulWebService()  2815
CreateWebService()  2814
generateWSDL()  2816
publishOperation()  2816
registerInputHTTPVariable()  2816
registerInputRequestHandler()  2817
registerOutputHTTPVariable()  2818
registerOutputRequestHandler()  2818
registerWSDLHandler()  2819
saveWSDL()  2820
setComment()  2820
setFeature()  2820
methods - com.WebServiceEngine
Flush()  2831
GetHTTPServiceRequest()  2832
GetOption()  2833
HandleRequest()  2833
ProcessServices()  2834
RegisterRestService()  2835
RegisterService()  2835
SetFaultCode()  2836
SetFaultDetail()  2836
SetFaultString()  2837
SetOption()  2837
SetRestError()  2837
Start()  2839
methods - DICTIONARY
clear()  2401
contains()  2402
copyTo()  2402
getKeys()  2403
getLength()  2403
remove()  2404
methods - DYNAMIC ARRAY
addElement()  2394
clear()  2395
copyTo()  2395
deleteElement()  2396
getLength()  2396
insertElement()  2397
search()  2397
searchRange()  2398
sort()  2399
methods - Java Array
create()  2405
getLength()  2405
methods - OAuthAPI
CreateHTTPAuthorizationRequest()  3515
ExtractTokenFromHTTPRequest()  3519
FetchOpenIDMetadata()  3518
GetIDIssuer()  3512
GetIDScopes()  3513
GetIDSubject()  3513
GetMyAccessToken()  3514
GetOpenIDMetadata()  3512
init()  3510
InitService()  3510
RetrievePasswordToken()  3517
RetrieveServiceToken()  3520
RetryHTTPRequest()  3516
methods - om.DomDocument
copy()  2583
create()  2579
createChars()  2581
createElement()  2582
createEntry()  2582
createFromString()  2579
createFromXmlFile()  2580
getDocumentElement()  2580
generateWSDL()  2580
removeElement()  2584
methods - om.DomNode
appendChild()  2587
createChild()  2588
clear()  2590
getAttribute()  2590
getAttributeInteger()  2591
getAttributeName()  2592
getAttributesCount()  2591
getAttributeString()  2592
getAttributeValue()  2593
getFirstChild()  2594
getFirstChild()  2594
getFirstChild()  2594
getId()  2595
getLastChild()  2595
getNext()  2596
getParent()  2596
getPrevious()  2597
getTagName()  2597
insertBefore()  2589
loadXml()  2589
parse()  2590
removeAttribute() 2597
removeChild() 2598
replaceChild() 2599
selectByPath() 2601
selectByTagName() 2602
setAttribute() 2603
toString() 2604
write() 2604
writeXml() 2604
methods - om.NodeList
getLength() 2607
item() 2608
methods - om.SaxAttributes
addAttribute() 2611
clear() 2613
copy() 2612
create() 2612
getLength() 2613
getName() 2614
getValue() 2614
getValueByIndex() 2615
removeAttribute() 2615
setAttribute() 2616
methods - om.SaxDocumentHandler
characters() 2620
createForName() 2618
endDocument() 2620
endElement() 2620
processingInstructions() 2621
readXmlFile() 2621
setIndent() 2622
skippedEntity() 2623
startDocument() 2622
startElement() 2622
methods - om.XmlReader
createFileReader() 2626
getAttributes() 2627
getCharacters() 2627
getTagName() 2628
read() 2628
skippedEntity() 2629
methods - om.XmlWriter
createChannelWriter() 2631
createFileWriter() 2632
createPipeWriter() 2633
createSocketWriter() 2633
methods - os.Path
atime() 2795
baseName() 2795
chDir() 2796
chOwn() 2796
chRwc() 2796
chVolume() 2797
copy() 2795
delete() 2797
dirClose() 2797
dirFMask() 2797
dirName() 2798
dirNext() 2798
dirOpen() 2799
dirSort() 2799
executable() 2799
exists() 2800
extension() 2800
fullPath() 2800
gid() 2801
glob() 2801
homeDir() 2802
isDirectory() 2802
isFile() 2803
isHidden() 2803
isLink() 2803
isRoot() 2803
join() 2804
makeTempName() 2804
mkDir() 2805
mtime() 2804
pathSeparator() 2805
pathType() 2805
pwd() 2806
readable() 2806
rename() 2806
rootDir() 2807
rootName() 2807
rwx() 2808
separator() 2806
size() 2807
type() 2808
uid() 2808
volumes() 2809
writable() 2809
methods - security.Base64
FromByte() 3131
FromHexBinary() 3131
FromString() 3131
FromStringWithCharset() 3132
LoadBinary() 3132
SaveBinary() 3132
ToByte() 3133
ToHexBinary() 3133
ToString() 3133
ToStringWithCharset() 3134
Xor() 3134
methods - security.BCrypt
CheckPassword() 3152
methods - security.Digest
AddBase64Data() 3144
AddData() 3143
AddHexBinaryData() 3144
AddStringData() 3143
AddStringDataWithCharset() 3144
CreateDigest() 3145
CreateDigestString() 3146
DoBase64Digest() 3146
DoHexBinaryDigest() 3147
methods - security.HexBinary
FromBase64() 3138
FromByte() 3138
FromString() 3138
FromStringWithCharset() 3139
LoadBinary() 3139
SaveBinary() 3139
ToBase64() 3140
ToByte() 3140
ToString()  3140
ToStringWithCharset()  3141
Xor()  3141
methods - security.PBKDF2
  CheckKey()  3149
  GenerateKey()  3148
  GenerateSalt()  3151
  HashPassword()  3151
methods - security.RandomGenerator
  CreateRandomNumber()  3127
  CreateRandomString()  3128
  CreateUUIDString()  3128
methods - STRING
  append()  2383
  equals()  2383
  equalsIgnoreCase()  2384
  getCharAt()  2384
  getIndexOf()  2385
  getLength()  2385
  subString()  2386
  toLowerCase()  2387
  toUpperCase()  2387
  trim()  2387
  trimLeft()  2389
  trimLeftWhiteSpace()  2389
  trimRight()  2390
  trimRightWhiteSpace()  2390
  trimWhiteSpace()  2388
methods - TEXT
  getLength()  2391
  readFile()  2392
  writeFile()  2392
methods - ui.ComboBox
  addItem()  2565
  clear()  2565
  forName()  2564
  getColumnName()  2565
  getIndexOf()  2566
  getItemCount()  2566
  getName()  2567
  getNameText()  2567
  getTable()  2567
  getTag()  2568
  getText()  2568
  removeItem()  2568
  setDefaultInitializer()  2562
methods - ui.Dialog
  accept()  2520
  addConstructByContent()  2520
  addDisplayArrayTo()  2521
  addInputArrayFrom()  2522
  addInputByName()  2523
  addTrigger()  2524
  append()  2526
  appendRow()  2525
  arrayToVisualIndex()  2527
  cancel()  2527
  close()  2528, 2537
  createConstructByContent()  2515
  createDisplayArrayTo()  2515
  createInputArrayFrom()  2516
  createInputByName()  2517
createMultipleDialog()  2518
deleteAllDialogs()  2528
deleteNode()  2529
deleteRow()  2529
getArrayLength()  2530
getCurrent()  2519
getCurrentItem()  2531
getCurrentRow()  2531
getFieldBuffer()  2532
getFieldTouched()  2532
getFieldValue()  2533
getForm()  2534
getQueryFromField()  2534
setSortKey()  2534
insertNode()  2535
insertRow()  2535
isRowSelected()  2536
isSortReverse()  2537
nextField()  2539
selectionToString()  2540
setActionActive()  2541
setActionComment()  2541
setActionHidden()  2542
setActionImage()  2542
setActionText()  2543
setArrayAttributes()  2543
setArrayLength()  2545
setCellAttributes()  2546
setCompleterItems()  2548
setCurrentRow()  2548
setDefaultUnbuffered()  2519
setFieldActive()  2549
setFieldTouched()  2550
setFieldValue()  2550
setSelectionMode()  2552
setSelectionMode()  2552
validate()  2553
visualToArrayIndex()  2553
methods - ui.DragDrop
  addPossibleOperation()  2572
  drop()  2572
  getBuffer()  2572
  getLocationParent()  2573
  getLocationRow()  2573
  getOperation()  2574
  getSelectedMimeType()  2574
  selectMimeType()  2575
  setBuffer()  2575
  setFeedback()  2576
  setMimeType()  2576
  setOperation()  2577
methods - ui.Form
  displayTo()  2500
  ensureElementVisible()  2501
  ensureFieldVisible()  2502
  findNode()  2503
  getNode()  2502
  loadActionDefaults()  2504
  loadToolBar()  2504
  loadTopMenu()  2505
  setDefaultInitializer()  2499
  setElementComment()  2500
setElementHidden() 2506
setElementImage() 2507
setElementStyle() 2507
setElementText() 2508
setFieldComment() 2508
setFieldHidden() 2509
setFieldStyle() 2509

methods - ui.Interface
filenameToURI() 2475
frontCall() 583, 2473
 getChildCount() 2476
getChildInstances() 2476
getContainer() 2477
getDocument() 2477
getFrontEndName() 2477
getFrontEndVersion() 2478
getImage() 2479
getName() 2479
getRootNode() 2479
getText() 2480
getType() 2480
getUniversalClientName() 2480
getUniversalClientVersion() 2481
loadActionDefaults() 2481
loadStartMenu() 2482
loadStyles() 2482
loadToolBar() 2483
loadTopMenu() 2484
refresh() 2484
setContainer() 2485
setImage() 2485
setName() 2486
setSize() 2486
setText() 2487
setType() 2487

methods - ui.Window
createForm() 2492
findNode() 2494
forName() 2491
getCurrent() 2493
getForm() 2493
getImage() 2495
getNode() 2494
getText() 2495
setImage() 2495
setText() 2495

methods - util.Date
isLeapYear() 2728
parse() 2727

methods - util.Datetime
format() 2729
fromSecondsSinceEpoch() 2730
currentAsUTC() 2731
parse() 2731
toLocalTime() 2732
toSecondsSinceEpoch() 2733
toUTC() 2733

methods - util.Integer
abs() 2738
or() 2742
parseBinaryString() 2743
parseHexString() 2743
setBit() 2744
shiftLeft() 2745
shiftRight() 2746
testBit() 2747
toBinaryString() 2748
toHexString() 2747
xor() 2749

methods - util.Interval
format() 2750
parse() 2751

methods - util.JSON
format() 2767
parse() 2768
proposeType() 2769
.stringify() 2770
.stringifyOmitNulls() 2772

methods - util.Interval
create() 2785
fromFGL() 2786
get() 2787
getLength() 2788
type() 2788
parse() 2786
put() 2789
remover() 2790
toFGL() 2790
toString() 2791

methods - util.JSONArray
create() 2785
fromFGL() 2786
get() 2787
getLength() 2788
type() 2788
parse() 2786
put() 2789
remoyer() 2790
toFGL() 2790
toString() 2791

methods - util.JSONObject
create() 2775
fromFGL() 2776
get() 2777
getLength() 2778
type() 2779
has() 2780
name() 2780
parse() 2777
put() 2781
remoyer() 2782
toFGL() 2782
toString() 2783

methods - util.Math
acos() 2762
asin() 2762
atan() 2762
cos() 2762
exp() 2763
log() 2765
pi() 2763
pow() 2763
rand() 2763
sin() 2764
sqrt() 2764
srand() 2765
tan() 2765
toDegrees() 2765
toRadians() 2766

methods - util.Strings
base64Decode() 2754
<table>
<thead>
<tr>
<th>Method/Property</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>base64DecodeToHexString()</td>
<td>2755</td>
</tr>
<tr>
<td>base64DecodeToString()</td>
<td>2756</td>
</tr>
<tr>
<td>base64Encode()</td>
<td>2754</td>
</tr>
<tr>
<td>base64EncodeFromHexString()</td>
<td>2756</td>
</tr>
<tr>
<td>base64EncodeFromString()</td>
<td>2757</td>
</tr>
<tr>
<td>encodeURIComponent()</td>
<td>2759</td>
</tr>
<tr>
<td>methods - WSHelper</td>
<td></td>
</tr>
<tr>
<td>FindQueryStringValue()</td>
<td>3501</td>
</tr>
<tr>
<td>SplitQueryString()</td>
<td>3502</td>
</tr>
<tr>
<td>SplitUrl()</td>
<td>3503</td>
</tr>
<tr>
<td>methods - xml.CryptoKey</td>
<td></td>
</tr>
<tr>
<td>compareTo()</td>
<td>3051</td>
</tr>
<tr>
<td>computeKey()</td>
<td>3051</td>
</tr>
<tr>
<td>Create()</td>
<td>3052</td>
</tr>
<tr>
<td>CreateDerivedKey()</td>
<td>3052</td>
</tr>
<tr>
<td>CreateFromNode()</td>
<td>3053</td>
</tr>
<tr>
<td>deriveKey()</td>
<td>3053</td>
</tr>
<tr>
<td>generateKey()</td>
<td>3054</td>
</tr>
<tr>
<td>getFeature()</td>
<td>3054</td>
</tr>
<tr>
<td>getSHA1()</td>
<td>3054</td>
</tr>
<tr>
<td>getSize()</td>
<td>3055</td>
</tr>
<tr>
<td>getType()</td>
<td>3055</td>
</tr>
<tr>
<td>getUrl()</td>
<td>3055</td>
</tr>
<tr>
<td>getUsage()</td>
<td>3056</td>
</tr>
<tr>
<td>loadBIN()</td>
<td>3056</td>
</tr>
<tr>
<td>loadDER()</td>
<td>3057</td>
</tr>
<tr>
<td>loadFromHexString()</td>
<td>3057</td>
</tr>
<tr>
<td>loadPEM()</td>
<td>3057</td>
</tr>
<tr>
<td>loadPrivateKey()</td>
<td>3058</td>
</tr>
<tr>
<td>loadPublicKey()</td>
<td>3058</td>
</tr>
<tr>
<td>loadPublicKeyFromHexString()</td>
<td>3058</td>
</tr>
<tr>
<td>savePrivate()</td>
<td>3059</td>
</tr>
<tr>
<td>savePublicKey()</td>
<td>3059</td>
</tr>
<tr>
<td>savePublicKeyToHexString()</td>
<td>3059</td>
</tr>
<tr>
<td>saveToString()</td>
<td>3060</td>
</tr>
<tr>
<td>setFeature()</td>
<td>3060</td>
</tr>
<tr>
<td>methods - xml.CryptoX509</td>
<td></td>
</tr>
<tr>
<td>Create()</td>
<td>3070</td>
</tr>
<tr>
<td>CreateFromNode()</td>
<td>3070</td>
</tr>
<tr>
<td>createPublicKeyKey()</td>
<td>3070</td>
</tr>
<tr>
<td>getFeature()</td>
<td>3071</td>
</tr>
<tr>
<td>getIdentifier()</td>
<td>3071</td>
</tr>
<tr>
<td>getThumbprintSHA1()</td>
<td>3071</td>
</tr>
<tr>
<td>load()</td>
<td>3072</td>
</tr>
<tr>
<td>loadER()</td>
<td>3072</td>
</tr>
<tr>
<td>loadFromHexString()</td>
<td>3072</td>
</tr>
<tr>
<td>loadPEM()</td>
<td>3073</td>
</tr>
<tr>
<td>save()</td>
<td>3073</td>
</tr>
<tr>
<td>saveToString()</td>
<td>3073</td>
</tr>
<tr>
<td>setFeature()</td>
<td>3074</td>
</tr>
<tr>
<td>methods - xml.DomDocument</td>
<td></td>
</tr>
<tr>
<td>appendDocumentNode()</td>
<td>2938</td>
</tr>
<tr>
<td>clone()</td>
<td>2938</td>
</tr>
<tr>
<td>create()</td>
<td>2938</td>
</tr>
<tr>
<td>createAttribute()</td>
<td>2939</td>
</tr>
<tr>
<td>createAttributeNS()</td>
<td>2939</td>
</tr>
<tr>
<td>createCDATASection()</td>
<td>2940</td>
</tr>
<tr>
<td>createComment()</td>
<td>2940</td>
</tr>
<tr>
<td>createDocument()</td>
<td>2941</td>
</tr>
<tr>
<td>createDocumentFragment()</td>
<td>2941</td>
</tr>
<tr>
<td>createDocumentNS()</td>
<td>2941</td>
</tr>
<tr>
<td>createDocumentType()</td>
<td>2942</td>
</tr>
<tr>
<td>createElement()</td>
<td>2943</td>
</tr>
<tr>
<td>createElementNS()</td>
<td>2943</td>
</tr>
<tr>
<td>createEntityReference()</td>
<td>2944</td>
</tr>
<tr>
<td>createNode()</td>
<td>2944</td>
</tr>
<tr>
<td>createProcessingInstruction()</td>
<td>2944</td>
</tr>
<tr>
<td>createTextNode()</td>
<td>2945</td>
</tr>
<tr>
<td>declareNamespace()</td>
<td>2945</td>
</tr>
<tr>
<td>getDocumentElement()</td>
<td>2946</td>
</tr>
<tr>
<td>getDocumentNodeItem()</td>
<td>2946</td>
</tr>
<tr>
<td>getDocumentNodesCount()</td>
<td>2946</td>
</tr>
<tr>
<td>getElementById()</td>
<td>2947</td>
</tr>
<tr>
<td>getElementsByTagName()</td>
<td>2947</td>
</tr>
<tr>
<td>getElementsByTagNameNS()</td>
<td>2947</td>
</tr>
<tr>
<td>getErrorDescription()</td>
<td>2948</td>
</tr>
<tr>
<td>getErrorsCount()</td>
<td>2948</td>
</tr>
<tr>
<td>getFeature()</td>
<td>2949</td>
</tr>
<tr>
<td>getFirstDocumentNode()</td>
<td>2949</td>
</tr>
<tr>
<td>getLastDocumentNode()</td>
<td>2950</td>
</tr>
<tr>
<td>getXmlEncoding()</td>
<td>2950</td>
</tr>
<tr>
<td>getXmlVersion()</td>
<td>2950</td>
</tr>
<tr>
<td>importNode()</td>
<td>2950</td>
</tr>
<tr>
<td>insertAfterDocumentNode()</td>
<td>2951</td>
</tr>
<tr>
<td>insertBeforeDocumentNode()</td>
<td>2951</td>
</tr>
<tr>
<td>isXmlStandalone()</td>
<td>2952</td>
</tr>
<tr>
<td>load()</td>
<td>2952</td>
</tr>
<tr>
<td>loadFromPipe()</td>
<td>2953</td>
</tr>
<tr>
<td>loadFromString()</td>
<td>2953</td>
</tr>
<tr>
<td>normalize()</td>
<td>2954</td>
</tr>
<tr>
<td>prependDocumentNode()</td>
<td>2954</td>
</tr>
<tr>
<td>removeDocumentNode()</td>
<td>2954</td>
</tr>
<tr>
<td>saver()</td>
<td>2955</td>
</tr>
<tr>
<td>saveToPipe()</td>
<td>2955</td>
</tr>
<tr>
<td>saveToString()</td>
<td>2956</td>
</tr>
<tr>
<td>selectByXPath()</td>
<td>2956</td>
</tr>
<tr>
<td>setFeature()</td>
<td>2957</td>
</tr>
<tr>
<td>setXmlEncoding()</td>
<td>2957</td>
</tr>
<tr>
<td>setXmlStandalone()</td>
<td>2957</td>
</tr>
<tr>
<td>validate()</td>
<td>2959</td>
</tr>
<tr>
<td>validateOneElement()</td>
<td>2959</td>
</tr>
<tr>
<td>methods - xml.DomNode</td>
<td></td>
</tr>
<tr>
<td>addNextSibling()</td>
<td>2974</td>
</tr>
<tr>
<td>addPreviousSibling()</td>
<td>2974</td>
</tr>
<tr>
<td>appendChild()</td>
<td>2975</td>
</tr>
<tr>
<td>appendChildElement()</td>
<td>2975</td>
</tr>
<tr>
<td>appendChildElementNS()</td>
<td>2976</td>
</tr>
<tr>
<td>clone()</td>
<td>2976</td>
</tr>
<tr>
<td>getAttribute()</td>
<td>2976</td>
</tr>
<tr>
<td>getAttributeCount()</td>
<td>2978</td>
</tr>
<tr>
<td>getAttributeNode()</td>
<td>2977</td>
</tr>
<tr>
<td>getAttributeNodeItem()</td>
<td>2977</td>
</tr>
<tr>
<td>getAttributeNodeNS()</td>
<td>2977</td>
</tr>
<tr>
<td>getAttributeNS()</td>
<td>2978</td>
</tr>
<tr>
<td>getChildNodeItem()</td>
<td>2979</td>
</tr>
<tr>
<td>getChildrenCount()</td>
<td>2979</td>
</tr>
<tr>
<td>getElementByTagName()</td>
<td>2979</td>
</tr>
<tr>
<td>getElementsByTagNameNS()</td>
<td>2980</td>
</tr>
<tr>
<td>getFirstChild()</td>
<td>2980</td>
</tr>
<tr>
<td>getFirstChildElement()</td>
<td>2980</td>
</tr>
<tr>
<td>getLastChild()</td>
<td>2981</td>
</tr>
<tr>
<td>getLastChildElement()</td>
<td>2981</td>
</tr>
<tr>
<td>getLocalName()</td>
<td>2981</td>
</tr>
</tbody>
</table>
getNamespaceURI() 2981
getNextSibling() 2982
getNodeName() 2982
getNextSiblingElement() 2982
getNodeValue() 2982
getParentNode() 2983
hasAttribute( ) 2985
hasAttributes() 2985
hasChildNodes() 2986
insertAfterChild() 2986
insertBeforeChild() 2986
isAttached() 2987
isDefaultNamespace() 2987
lookupNamespaceURI() 2988
lookupPrefix() 2988
prependChild() 2988
prependChildElement() 2989
prependChildElementsNS() 2989
removeAllChildren() 2989
removeAttribute() 2990
removeAttributeNS() 2990
removeChild() 2990
replaceChild() 2991
setAttributeNode() 2992
setAttributeNodeNS() 2992
setAttributeNS() 2993
setIdAttribute() 2993
setIdAttributeNS() 2994
getNodeValue() 2994
toString() 2995
methods - xml.DomNodeList
getCount() 2998
getItem() 2998
methods - xml.Encryption
Create() 3107
decryptElement() 3107
decryptElementContent() 3107
decryptElementContentDetached() 3107
decryptElementDetached() 3108
decryptKey() 3108
DecrypString() 3109
encryptElement() 3109
encryptElementContent() 3109
encryptElementContentDetached() 3110
encryptElementDetached() 3110
encryptKey() 3110
EncryptString() 3111
getEmbeddedKey() 3111
RSAEncrypt() 3112
RSADecrypt() 3112
setCertificate() 3113
setKey() 3113
setKeyEncryptionKey() 3113
methods - xml.KeyStore
AddCertificate() 3117
AddKey() 3118
AddTrustedCertificate() 3118
methods - xml.Serializer
CreateXmlSchemas() 3036
DomToStax() 3036
DomToVariable() 3036
getOption() 3037
OptimizedDomToVariable() 3037
OptimizedSoapSection5ToVariable() 3038
OptimizedStaxToVariable() 3039
OptimizedVariableToDom() 3040
OptimizedVariableToSoapSection5() 3041
OptimizedVariableToStax() 3042
setOption() 3043
SoapSection5ToVariable() 3044
StaxToDom() 3044
StaxToVariable() 3045
VariableToDom() 3045
VariableToSoapSection5() 3045
VariableToStax() 3046
methods - xml.Signature
appendObjectData() 3080
appendReferenceTransformation() 3080
compute() 3081
Create() 3081
CreateFromNode() 3081
createObject() 3082
createReference() 3082
getCanonicalization() 3083
getDocument() 3083
getID() 3083
getObjectCount() 3084
getObjectId() 3084
getReferenceCount() 3084
getReferenceDigest() 3084
getReferenceID() 3085
getReferenceTransformation() 3085
getReferenceTransformationCount() 3085
getReferenceURI() 3086
getSignatureMethod() 3086
type() 3086
RetrieveObjectDataListFromSignatureNode() 3087
setCanonicalization() 3087
setCertificate() 3087
setId() 3088
setKey() 3088
setObjectId() 3088
setReferenceID() 3089
signString() 3089
verify() 3090
verifyString() 3090
methods - xml.StaxReader
close() 3017
Create() 3018
findAttributeValue() 3018
getAttributeCount() 3018
getAttributeLocalName() 3019
getAttributeNamespace() 3019
getAttributePrefix() 3019
getAttributeValue() 3020
methods - xml.StaxWriter
  attribute() 3001
  attributeNS() 3002
  cdata() 3002
  characters() 3003
  close() 3003
  comment() 3003
  create() 3003
  declareDefaultNamespace() 3004
  declareNamespace() 3004
  dd() 3005
  emptyElement() 3005
  emptyElementNS() 3005
  endDocument() 3006
  endElement() 3006
  entityRef() 3006
  getFeature() 3006
  processingInstruction() 3007
  setDefaultNamespace() 3007
  setFeature() 3007
  setPrefix() 3008
  startDocument() 3008
  startElement() 3009
  startElementNS() 3009
  writeTo() 3009
  writeToDocument() 3010
  writeToPipe() 3011
  writeToText() 3011

Microsoft SQL Server
  BIGINT 842
  BOOLEAN 838
  CHAR / VARCHAR 838
  Character case sensitivity 854
  CHECK 855
  Constraints 855
  CREATE TEMP TABLE 861
  Data type conversions 837
  Database concepts 831
  Database object names 856
  Database privileges 836
  Database users 836
  DATE 844
  DATETIME 844
  DECIMAL 842
  Errors on PREPARE 868
  FOREIGN KEY 855
  INSERT cursors 869
  INTEGER 842
  INTERVAL 846
  ISOLATION LEVEL 833
  LENGTH() 866
  LOAD / UNLOAD 872
  LOCK MODE 833
  MATCHES / LIKE 865
  MONEY 842
  NCHAR / NVARCHAR 838
  NOT NULL 855
  OUTER join 858
  PRIMARY KEY 855
  Reserved words 857
  ROWID 853
  ROWVERSION 852
  SCROLL CURSOR 873
  SELECT ... FOR UPDATE 833, 870
  SELECT ... INTO TEMP 861
  SERIAL / BIGSERIAL 847
  Single row SELECT 865
  SQL interrupt 873
  Storage concepts 831
  String concatenation 868
  String delimiters 863
  Substrings 862
  TEXT / BYTE 854
  Transactions 835, 860
  UNIQUE 855
  UNIQUEIDENTIFIER 852
  VARCHAR(MAX) / VARBINARY(MAX) 854
  WITH HOLD 869

Microsoft Windows
  JNI Java Native Interface 2194
migration
  database driver 163
  Four Js BDL 231
  Informix 4GL 211
  OM XML 3125

mobile
  app execution 3572
  app execution modes 3572

module
  importing 481
  multi-byte 517
  multibyte, See multi-byte
MySQL
  LENGTH() 901

N
Native Rendering 1111
methods - util.Strings
  urlDecode() 2758

new features
  version information 48
NEXT FIELD
  dialog programming 1731
  nextField()
    dialog programming 1731

O
OAuthAPI
  definitions 3505
  functions
    access tokens 3514
    authorization request 3515
    helpers 3517
    ID token 3512
    initialization 3510
    post-initialization 3511
library 3504
  mobile app 3505
  OAuthApp 3505
  OpenIDMetadataType 3505
  overview 3504
  RegisterRequestType 3505
  RegisterResponseType 3505
  service to service 3505
  with GAS 3505
Object reference definition 570
object references
  passed as parameter 498
OM XML
  migration 3125
ON ACTION close
  dialog programming 1772
ON ACTION cordovacallback 2722
ON ACTION INFIELD
  dialog programming 1767
ON CHANGE
  dialog programming 1724
operating system 37
Operating system notes 43

operator
  function ( [ ... ] ) 362
  (expr) 361
  AND 348
  ASCII() 353
  CAST 364
  CLIPPED 358
  COLUMN 354
  CURRENT 367
  DATE() 368
  DAY() 371
  expr - expr 350
  expr , expr 356
  expr != expr 344
  expr * expr 351
  expr ** expr 352
  expr / expr 351
  expr + expr 350
  expr < expr 345
  expr <= expr 345
  expr == expr 343
  expr > expr 346
  expr >= expr 346
  expr || expr 355
  EXTEND() 368
  FIELD_TOUCHED() 375
  GET_FLDBUF() 374
  IIF() 347
  INFIELD() 375
  INSTANCEOF 365
  IS NOT NULL 340
  IS NULL 340
  LIKE 340
  LSTR() 360
  MATCHES 341
  MDY() 372
  MOD 352
  MONTH() 370
  NOT 348
  NOT LIKE 340
  NOT MATCHES 341
  NVL() 347
  OR 349
  ORD() 359
  record.member 362
  variable[s,e] 357
  SFMT() 360
  SPACES 359
  SQLERRMESSAGE 364
  SQLSTATE 363
  TIME() 369
  TODAY 369
  UNITS 373
  USING 357
  variable := expr 366
  walrus operator 366
  WEEKDAY() 371
  YEAR() 370

operator - reports
  LINENO 2058
  PAGENO 2058
  SPACES 2059
operators
- arithmetic 350
- assignment 366
- associative syntax 361
- character string 353
- comparison 339
- data type 364
- date and time 367
- dialog handling 374
- logical 348
- SQL related 363

optimization
- bottlenecks 587
- CHAR parameters 587
- concatenating strings 588
- dynamic array 590
- FGLLDPATH 586
- FGLRESOURCEPATH 586
- mmapdisable 586
- parsing strings 588
- private program elements 587
- records by reference 588
- shared module elements 587
- shared program elements (42m, 42s files) 586
- splitting modules 589
- SQL statements 587
- STRING variables 589
- unused variables 589
- upgrading your application (42m/42f files) 586

OPTIONS (compilation)
- SHORT CIRCUIT 558

OPTIONS (runtime)
- COMMENT LINE 559
- DELETE KEY 564
- DISPLAY ATTRIBUTES 560
- ERROR LINE 559
- FIELD ORDER CONSTRAINED 561
- FIELD ORDER FORM 561
- FIELD ORDER UNCONSTRAINED 561
- FORM LINE 559
- HELP FILE 563
- HELP KEY 564
- INPUT ATTRIBUTES 560
- INPUT NO WRAP 561
- INPUT WRAP 561
- INSERT KEY 564
- MENU LINE 559
- MESSAGE LINE 559
- NEXT KEY 564
- ON CLOSE APPLICATION 563
- ON TERMINATE SIGNAL CALL 563
- PREVIOUS KEY 564
- PROMPT LINE 559
- RUN IN FORM MODE 565
- RUN IN LINE MODE 565
- SQL INTERRUPT ON 565

ora.rowid.retrieve 128

Oracle
- database client environment 659

Oracle DB
- BIGINT 924

Oracle MySQL
- database client environment 659

Oracle MySQL/MariaDB
- BIGINT 886
- BOOLEAN 883
- CHAR / VARCHAR 883
- CHECK 894
Index

cordovacallback 1741
delete 1741
dialogtouched 1741
decopy 1741
derenterbackground 1741
derenterforeground 1741
dexpandall 1741
find 1741
findnext 1741
firstrow 1741
help 1741
insert 1741
interrupted 1741
lastrow 1741
nextrow 1741
notificationpushed 1741
prevrow 1741
verifiernotifications 1741
windowresized 1741

predefined values

presentation styles 1173

preprocessor

__FILE__ 2140
__LINE__ 2140
&define (simple) 2136
&define (with parameters) 2138
#include 2134
&undef 2141
# stringification 2140
## concatenation 2141
basics 2132
command line options 2133

presentation styles

combining 1171
defining 1167
loading 1172
precedence 1170

profiler

basics 2165
call graph 2166
output file format 2165
syntax 2165

program execution
fglrn 488

programming

localized strings 542
logging options 2192
optimization 586
user interface 1706
Web Component (gICAPI) 1932
Web Component (URL) 1928

property

autoflush 1972
mouse... 1988
selection 1979, 1988
spellcheck 1972
toolbar 1972

protocol

authentication 3303
front-end 1117
HTTP 3303

HTTPS 3303
proxy 3303
pseudo selectors 1168
push notification
Apple Push Notification Service (APNs) 3625
Firebase Cloud Messaging (FCM) 3621
Mobile app handling notifications 3637
Token maintainer 3631

Q

quick start
application locale 515
localized strings 538

R

RECORD

returned from functions 499
records
accessing members 412
attributes 411
comparing 414
copying 412
function parameters 415
initializers 410
passed as parameter 494

reference

Exceptions 3154
file extensions 3153
Genero BDL errors 3154
Genero BDL library 2268
Genero Web Services 3463
GWS 3463
key codes 2300
keyboard accelerators 1773
predefined action list 1773
presentation styles 1182
registers 567
remote execution
mobile front call 2710
rendering actions
mobile devices 1777

REPORT 2039

report routine
aggregate functions 2061
instructions 2052
operators 2058
report-routines 2039

REST high-level
handle file attachments 3386

RESTful web services
server application 3430

RESTful Web services
calculator source 3436, 3443
client source 3442
server source 3436
row/cell highlighting - scrollgrid
highlightColor 1204
highlightCurrentCell 1204
highlightCurrentRow 1204
highlightTextColor 1204
row/cell highlighting - table
  highlightColor 1208
  highlightCurrentCell 1208
  highlightCurrentRow 1208
  highlightTextColor 1208
row/cell highlighting - tree
  highlightColor 1212
  highlightCurrentRow 1212
  highlightTextColor 1212
ROWBOUND
  on a mobile device 1873
run time options
  DEFER INTERRUPT 567
  DEFER QUIT 567
runOnServer 3616
runtime
  FGLRESOURCEPATH 544
  fglrun.localization 544
  localized strings 544
  reloadResources 544

S
samples 28
SAP ASE
  BIGINT 1044
  BOOLEAN 1041
  CHAR / VARCHAR 1041
  Character case sensitivity 1052
  CHECK 1054
  CLOB / BLOB 1053
  Constraints 1054
  CREATE TEMP TABLE 1058
  Data type conversions 1040
  database client environment 660
  Database concepts 1036
  Database object names 1054
  Database privileges 1039
  Database users 1039
  DATE 1046
  DATETIME 1046
  DECIMAL 1044
  FOREIGN KEY 1054
  INSERT cursors 1063
  INTEGER 1044
  INTERVAL 1048
  ISOLATION LEVEL 1036
  LENGTH() 1062
  LOAD / UNLOAD 1066
  LOCK MODE 1036
  MATCHES / LIKE 1061
  MONEY 1044
  NCHAR / NVARCHAR 1041
  NOT NULL 1054
  OUTER join 1056
  PRIMARY KEY 1054
  Reserved words 1055
  row limiting 1063
  ROWID 1052
  SCROLL CURSOR 1067
  SELECT ... FOR UPDATE 1036, 1065
  SELECT ... INTO TEMP 1058
  SERIAL / BIGSERIAL 1048
  Single row SELECT 1061
  SMALLINT 1044
  SQL interrupt 1067
  Strings delimiters 1060
  Substrings 1059
  TEXT / BYTE 1053
  Transactions 1038, 1057
  UNICHAR / UNIVARCHAR 1041
  UNIQUE 1054
  WHERE CURRENT OF 1066, 1102
  WITH HOLD 1064
SAP HANA
  BIGINT 1079
  BOOLEAN 1077
  CHAR / VARCHAR 1077
  CHECK 1088
  CLOB / BLOB 1087
  Constraints 1088
  Data type conversions 1075
  database client environment 661
  Database concepts 1071
  Database object names 1089
  Database privileges 1075
  Database users 1074
  DATE 1080
  DATETIME 1080
  DECIMAL 1079
  For update 1071
  FOREIGN KEY 1088
  INSERT cursor 1100
  INTEGER 1079
  INTERVAL 1083
  Isolation level 1071
  LENGTH() 1098
  LOAD / UNLOAD 1102
  Lock mode 1071
  MATCHES / LIKE 1097
  MONEY 1079
  NCHAR / NVARCHAR 1077
  NOT NULL 1088
  OUTER join 1091
  PRIMARY KEY 1088
  Reserved words 1091
  row limiting() 1099
  ROWID 1087
  SCROLL CURSOR 1103
  SELECT ... FOR UPDATE 1101
  SERIAL / BIGSERIAL 1084
  Simple row SELECT 1097
  SMALLINT 1079
  SQL interrupt 1103
  Storage concepts 1071
  String delimiters 1096
  Substrings 1095
  Temporary tables 1094
  TEXT / BYTE 1087
  Transactions 1073, 1093
  UNIQUE 1088
  WITH HOLD 1100
SAX 573
SBCS 517
schema
defining the database schema 468
screen orientation
mobile apps 3581
SCROLL
dialog programming 1716
Scrollable cursors
SQL performances 654
scrollgrid
display a record list 1880
on mobile devices 1887
security
eexample 3152
front-end identification 1118
SSL - Secure Sockets Layer 3299
TLS - Transport Layer Security 3299
web services
Diffie-Hellman key agreement 3318
encrypted SOAP message 3366
HTTP 401 error 3318
policy binding 3363
signed SOAP message 3366
SOAP message security policy 3365
SSL/TLS certificate retrieval 3316
troubleshooting security issues 3318
security.Base64
methods 3129
security.BCrypt
methods 3150
security.Digest
methods 3142
security.HexBinary
methods 3136
security.PBKDF2
methods 3148
security.RandomGenerator
methods 3127
SERIAL columns 626
server
database 38
setActionActive()
dialog programming 1763
setActionHidden()
dialog programming 1764
setFieldActive()
dialog programming 1728
single-byte 517
SMS
mobile front call 2701
SOAP features
exceptions 3281
faults 3281
SOAP 1.1 protocol 3280
SOAP 1.2 protocol 3280
SOAP web services
client application
client runtime configuration 3324
client stubs and handlers 3328
server application
Get HTTP headers information 3356
high-level server function 3342
server stubs and handlers 3344
source
database 661
source documenting
basics 2126
constants 2127
files structure 2126
functions 2129
globals 2131
module 2131
package 2131
prerequisites 2126
project 2131
reports 2130
types 2128
variables 2128
sources
character set 529
split views
mobile apps 3579
SQL
basics 590
performance 651
portability 607
programming 590
security 605
SQL CHAR/VARCHAR portability
Best practices 622
byte / char length semantics 616
database character set for UTF-8 617
Empty strings / NULL 619
size limits 618
Trailing blanks 620
SQL connection
APP_NAME() 678
APP=name 678
application identifier 678
application_name=name 678
SQL diagnostics 591
SQL interruption 601
SQL portability
auto-incremented columns 626
CHAR/VARCHAR 616
Concurrent data access 599
CREATE DATABASE 608
Cursors WITH HOLD + FOR UPDATE 631
data definition statements 613
data manipulation statements 615
database entities 607
Date and time handling 635
GROUP BY 646
Insert cursors 633
LENGTH() 647
LOAD/UNLOAD 639
LOBs (TEXT/BYTE) 648
MATCHES vs LIKE 645
Naming tables and columns 637
Optimistic locking 624
OUTER joins 643
Positionned updates/deletes 631
Row limiting clause 650
ROWIDs 644
SAVEPOINTS 613
Scrollable cursors 623
SELECT FIRST 650
SQL data types 614
Stored procedures 649
String literals 634
Substring expressions 644
Temporary tables 641
user definition 608
SQL programming
base.SqlHandle 713
SQL resources
SQL performances 653
SQL security
SQL injection 605
user authentication 605
SQL Server
database client environment 659
WHERE CURRENT OF 871
SQLCA
record members 594
SQLite
BIGINT 1012
BOOLEAN 1010
CHAR / VARCHAR 1011
CLOB / BLOB 1020
CREATE TEMP TABLE 1024
Data type conversions 1009
database client environment 660
Database concepts 1006
Database object names 1020
Database users 1008
DATE 1013
DATETIME 1013
DECIMAL 1012
FOREIGN KEY 1019
in-memory database (:memory:) 1006
INSERT cursors 1028
INTEGER 1012
INTERVAL 1016
ISOLATION LEVEL 1007
LENGTH() 1026
LOAD / UNLOAD 1030
LOCK MODE 1007
MATCHES / LIKE 1025
MONEY 1012
NCHAR / NVARCHAR 1011
Optimization
Modifying many rows 1031
VACUUM 1032
OUTER join 1021
PRAGMA foreign_keys 1019
row limiting 1027
ROWID 1019
SCROLL CURSOR 1031
SELECT ... FOR UPDATE 1028
SELECT ... INTO TEMP 1024
SERIAL / BIGSERIAL 1016
SMALLINT 1012
SQL interrupt 1030
Substrings 1025
TEXT / BYTE 1020
Transactions 1007, 1023
WHERE CURRENT OF 1029
WITH HOLD 1029
stack-based layout
group rendering 1434
label internationalization 1432
STARTLOG 508
static arrays
passed as parameter 496
status bar
presentation styles 1181
Stored procedure
IBM DB2 795
Informix 747
MariaDB 907
Oracle DB 963
Oracle MySQL 907
PostgreSQL 1002
SAP ASE 1067
SAP HANA 1103
SQL Server 873
stored settings
front-end 1418
string resources 538
style attribute - Action/MenuAction
androidActionPosition 1192
androidActionWithIcon 1192
androidActionWithText 1192
scaleIcon 1192
style attribute - button
buttonType 1185
scaleIcon 1185
style attribute - buttonedit
scaleIcon 1186
style attribute - checkbox
customWidget 1187
iosCheckBoxOnTintColor 1187
style attribute - dateedit
buttonIcon 1190
calendarType 1190
daysOff 1190
firstDayOfWeek 1190
showCurrentMonthOnly 1190
showGrid 1190
showWeekNumber 1190
style attribute - datetimedit
buttonIcon 1191
calendarType 1191
daysOff 1191
enableCalendar 1191
firstDayOfWeek 1191
showCurrentMonthOnly 1191
showGrid 1191
showWeekNumber 1191
style attribute - edit
dataTypeHint 1194
spellCheck 1194
style attribute - folder
collapserPosition 1195
position 1195
style attribute - form
resetFormSize 1196
style attribute - group
collapserPosition 1196
collapsible 1196
initiallyCollapsed 1196
style attribute - hbox
splitViewRendering 1197
style attribute - image
alignment 1198
imageContainerType 1198
style attribute - label
textFormat 1198
style attribute - menu
position 1199
style attribute - message
position 1199
textFormat 1199
style attribute - progressbar
percentageVisible 1201
style attribute - radiogroup
autoSelectionStart 1201
completionTimeout 1201
style attribute - scrollgrid
customWidget 1202
highlightColor 1202
highlightCurrentCell 1202
highlightCurrentRow 1202
highlightTextColor 1202
itemsAlignment 1202
rowActionTrigger 1202
style attribute - table
allowWebSelection 1204
forceDefaultSettings 1204
headerAlignment 1204
headerHidden 1204
highlightColor 1204
highlightCurrentCell 1204
highlightCurrentRow 1204
highlightTextColor 1204
leftFrozenColumns 1204
reduceFilter 1204
resizeFillsEmptySpace 1204
rightFrozenColumns 1204
rowActionTrigger 1204
showGrid 1204
summaryLineAlwaysAtBottom 1204
tableType 1204
style attribute - textedit
integratedSearch 1212
showEditToolBar 1212
spellCheck 1212
textFormat 1212
wrapPolicy 1212
style attribute - toolbar
scaleIcon 1217
toolBarTextPosition 1217
style attribute - toolbarseparator
iosSeparatorStretch 1218
style attribute - tree
allowWebSelection 1209
forceDefaultSettings 1209
headerAlignment 1209
headerHidden 1209
highlightColor 1209
highlightCurrentRow 1209
highlightTextColor 1209
leftFrozenColumns 1209
resizeFillsEmptySpace 1209
rightFrozenColumns 1209
rowActionTrigger 1209
showGrid 1209
tableType 1209
style attribute - userinterface
androidKeepForeground 1219
browserMultiPage 1219
reverse 1219
style attribute (common)
backgroundColor 1182
border 1182
fontFamily 1182
fontSize 1182
fontStyle 1182
fontWeight 1182
localAccelerators 1182
showAcceleratorInToolTip 1182
textColor 1182
textDecoration 1182
style attributes
inheritance 1171
TTY attributes 1172
style attributes - combobox
autoSelectionStart 1188
comboboxCompleter 1188
completionTimeout 1188
style attribute - window 1220
style attribute - window - action panel
actionPanelButtonSize 1228
actionPanelButtonSpace 1228
actionPanelButtonTextAlign 1228
actionPanelButtonTextHidden 1228
actionPanelDecoration 1228
actionPanelHAlign 1228
actionPanelPosition 1228
actionPanelScroll 1228
actionPanelScrollStep 1228
style attribute - window - basics
border 1220
defaultTTFColor 1220
errorMessagePosition 1220
forceDefaultSettings 1220
position 1220
sizable 1220	tabbedContainer 1220
tabbedContainerCloseMethod 1220
thinScrollbarDisplayTime 1220	toolBarDocking 1220
toolBarPosition 1220
windowOptionClose 1220
windowOptionMaximize 1220
windowOptionMinimize 1220
windowState 1220
windowType 1220
style attribute - window - GMI/iOS
iosNavigationBarTintColor 1232
iosNavigationBarTextColor 1232
iosRenderSystemActions 1232
iosTabBarTintColor 1232
iosTabBarUnselectedColor 1232
iosTintColor 1232
iosToolBarTintColor 1232

style attribute - window - miscellaneous
allowedOrientations 1225
backButtonAction 1225
formScroll 1225
ignoreMinimizeSetting 1225
materialFABActionList 1225
materialFABType 1225
menuPopupPosition 1225
messagePosition 1225
statusBarType 1225
windowMenu 1225
windowSystemMenu 1225

style attribute - window - ring menu
ringMenuButtonSize 1230
ringMenuButtonSpace 1230
ringMenuButtonTextAlign 1230
ringMenuButtonTextHidden 1230
ringMenuDecoration 1230
ringMenuHAlign 1230
ringMenuPosition 1230
ringMenuScroll 1230
ringMenuScrollStep 1230

style attribute - window - start menu
startMenuAccelerator 1231
startMenuExecShortcut2 1231
startMenuPosition 1231
startMenuShortcut 1231
startMenuSize 1231

sub-routines 437
support 28
suspend
SLEEP statement 388

syntax - action attributes
ACCELERATOR 1750
ACCELERATOR2 1751
ACCELERATOR3 1752
ACCELERATOR4 1752
COMMENT 1753
CONTEXTMENU 1754
DEFAULTVIEW 1755
DISCLOSUREINDICATOR 1756
IMAGE 1756
ROWBOUND 1757
TEXT 1758
VALIDATE 1759

syntax - basics
case sensitivity 285
escape symbol (backslash) 287
identifiers 289
preprocessor directives 289
quotation marks 286
source comments 287
statement terminator (semicolon) 287
white space separator 286

syntax - compile time options
OPTIONS 557

syntax - file format
action defaults files (.4ad) 1162
message files (.msg) 1160
presentation styles file (.4st) 1167
toolbar files (.4tb) 1437
topmenu files (.4tm) 1446

syntax - form file
ACCELERATOR attribute 1352
ACCELERATOR2 attribute 1352
ACCELERATOR3 attribute 1353
ACCELERATOR4 attribute 1353
ACTION attribute 1353
ACTION DEFAULTS section 1296
AGGREGATETEXT attribute 1354
AGGREGATETYPE attribute 1354
AGGREGATE item definition 1332
ATTRIBUTES section 1330
AUTONEXT attribute 1355
AUTOSCALE attribute 1355
BUTTON item definition 1333
BUTTON stack item 1310
BUTTONEDIT attribute 1356
BUTTONEDIT item definition 1333
BUTTONEDIT stack item 1311
CANVAS item definition 1334, 2013
CENTURY attribute 1356
CHECKBOX item definition 1335
CHECKBOX stack item 1312
CLASS attribute 1357
COLOR attribute 1357
COLOR WHERE attribute 1358
COMBOBOX item definition 1336
COMBOBOX stack item 1312
COMMENT attribute 1359
COMPLETER attribute 1360
COMPONENTTYPE attribute 1361
CONFIG attribute 1358
CONTTEXTMENU attribute 1359
DATEEDIT item definition 1337
DATEEDIT stack item 1313
DATETIMEEDIT item definition 1338
DATETIMEEDIT stack item 1313
DEFAULT attribute 1361
DEFAULTVIEW attribute 1362
DISCLOSUREINDICATOR attribute 1363
DISPLAY LIKE attribute 1363
DOUBLECLICK attribute 1364
dOWNSHIFT attribute 1364
EDIT item definition 1338
EDIT stack item 1314
EXPANDEDCOLUMN attribute 1365
FOLDER container 1307
FOLDER stack item 1315
FONTPITCH attribute 1365
FORM clause 1303
form file structure (.per) 1294
FORMAT attribute 1366
GRID container 1308
GRIDCHILDRENINPARENT attribute 1367
GROUP container 1306
GROUP item definition 1339
GROUP stack item 1315
Index

STATUS 567

syntax - reports
AFTER GROUP OF 2048
BEFORE GROUP OF 2048
DEFINE in REPORT 2042
FINISH REPORT 2038
FIRST PAGE HEADER 2047
FORMAT EVERY ROW 2046
FORMAT section 2045
ON EVERY ROW 2050
ON LAST ROW 2051
ORDER BY section 2044
OUTPUT section 2043
OUTPUT TO REPORT 2036
PAGE HEADER 2047
PAGE TRAILER 2048
START REPORT 2034
TERMINATE REPORT 2038

syntax - run time options
OPTIONS 558

system messages
$msg files 530

system packages
requirements 37

T

TABINDEX
dialog programming 1729
tables
display a list 1853
on mobile devices 1867
tap
on a mobile device 1872
TERM
terminal definition 1123
termcap
de file syntax 1124
TERMCAP
configuration 1123
terminal
colors 1125
ESCDelay 1125
tty attributes 1125
TERMINFO
configuration 1123
text mode 1109
TEXT/BYTE
passed as parameter 499
returned from functions 501
TEXT/BYTE (FTM/ESM DB drivers) 169
THROUGH 417
toolbars
mobile apps 3576
tools
localized string extraction 542
topupmenus
mobile apps 3577
trace errors 508
traditional mode 1114
transactions
across connections 612

nested 611
Transactions
SQL performances 652
troubleshooting
locale issues 535
tty 1125
TTY attributes 1172
tutorial 28

U

UNBUFFERED
dialog programming 1720
UNICODE 518
Universal Rendering 1112
upgrade guide
All versions 109
Version 1.10 210
Version 1.20 210
Version 1.30 201
Version 1.31 201
Version 1.32 201
Version 1.33 200
Version 2.00 191
Version 2.01 191
Version 2.02 191
Version 2.10 190
Version 2.11 189
Version 2.20 183
Version 2.21 180
Version 2.30 178
Version 2.32 176
Version 2.40 172
Version 2.41 172
Version 2.50 167
Version 2.51 162
Version 3.00 151
Version 3.10 129
Version 3.20 109

upgrading
Genero BDL 40
upgrade notes 48

URL-based WEBCOMPONENT
form definition 1929
program control 1930
source URL 1929

usage
base.Channel 2424
base.SqlHandle 2443
CONSTRUCT 1568, 1569
DIALOG (declarative dialogs) 1655
DIALOG (multiple dialogs) 1594
DIALOG (parallel dialogs) 1704
dictionary 428
DISPLAY ARRAY 1504, 1504
dynamic arrays 422
INPUT 1485
INPUT ARRAY 1535
interface 459
MENU 1470
ON IDLE 1712
ON TIMER 1713
<table>
<thead>
<tr>
<th>Feature</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>static arrays</td>
<td>420</td>
</tr>
<tr>
<td>ui.Dialog</td>
<td>2554</td>
</tr>
<tr>
<td>ui.Form</td>
<td>2510</td>
</tr>
<tr>
<td><strong>usage</strong> - SOAP web services**</td>
<td></td>
</tr>
<tr>
<td>server application implementation (high-level)</td>
<td>3348</td>
</tr>
<tr>
<td><strong>usage</strong> - web service client**</td>
<td></td>
</tr>
<tr>
<td>tGlobalEndpointType</td>
<td>3325</td>
</tr>
<tr>
<td><strong>usage</strong> - web services**</td>
<td></td>
</tr>
<tr>
<td>Error -15553</td>
<td>3555</td>
</tr>
<tr>
<td>exception handling</td>
<td>3554</td>
</tr>
<tr>
<td>generate BDL from WSDL or XSD (fglwsdl)</td>
<td>3564</td>
</tr>
<tr>
<td>INT_FLAG</td>
<td>3555</td>
</tr>
<tr>
<td>interruption handling</td>
<td>3555</td>
</tr>
<tr>
<td>STATUS</td>
<td>3554</td>
</tr>
<tr>
<td>TRY / CATCH</td>
<td>3554</td>
</tr>
<tr>
<td>XML serialization attributes</td>
<td>3523</td>
</tr>
<tr>
<td><strong>usage</strong> - web services client**</td>
<td></td>
</tr>
<tr>
<td>BDL stub functions</td>
<td>3329</td>
</tr>
<tr>
<td>BDL stub generation</td>
<td>3328</td>
</tr>
<tr>
<td>callback handlers</td>
<td>3330</td>
</tr>
<tr>
<td>calling stub function</td>
<td>3333</td>
</tr>
<tr>
<td>endpoint record definition</td>
<td>3329</td>
</tr>
<tr>
<td>fglwsdl</td>
<td>3328</td>
</tr>
<tr>
<td>HTTP connection timeout</td>
<td>3326</td>
</tr>
<tr>
<td>HTTP cookie</td>
<td>3326</td>
</tr>
<tr>
<td>HTTP headers</td>
<td>3327</td>
</tr>
<tr>
<td>HTTP protocol version</td>
<td>3326</td>
</tr>
<tr>
<td>HTTP read/write timeout</td>
<td>3327</td>
</tr>
<tr>
<td>multipart in stub</td>
<td>3332</td>
</tr>
<tr>
<td>server errors</td>
<td>3324, 3328</td>
</tr>
<tr>
<td>server location</td>
<td>3326</td>
</tr>
<tr>
<td>WS-Addressing endpoint parameters</td>
<td>3325</td>
</tr>
<tr>
<td><strong>WSDL</strong></td>
<td>3328</td>
</tr>
<tr>
<td><strong>user authentication</strong></td>
<td></td>
</tr>
<tr>
<td>database</td>
<td>679</td>
</tr>
<tr>
<td><strong>user interaction</strong></td>
<td></td>
</tr>
<tr>
<td>inactivity</td>
<td>1712</td>
</tr>
<tr>
<td><strong>user interface</strong></td>
<td></td>
</tr>
<tr>
<td>mobile apps</td>
<td>3574</td>
</tr>
<tr>
<td><strong>utility functions</strong></td>
<td></td>
</tr>
<tr>
<td>db_finish_transaction()</td>
<td>2311</td>
</tr>
<tr>
<td>db_get_database_type()</td>
<td>2307</td>
</tr>
<tr>
<td>db_get_last_serial()</td>
<td>2308</td>
</tr>
<tr>
<td>db_get_sequence()</td>
<td>2309</td>
</tr>
<tr>
<td>db_is_transaction_started()</td>
<td>2311</td>
</tr>
<tr>
<td>db_start_transaction()</td>
<td>2309</td>
</tr>
<tr>
<td>fgl_winbutton()</td>
<td>2302</td>
</tr>
<tr>
<td>fgl_winmessage()</td>
<td>2303</td>
</tr>
<tr>
<td>fgl_winprompt()</td>
<td>2304</td>
</tr>
<tr>
<td>fgl_winquestion()</td>
<td>2305</td>
</tr>
<tr>
<td>fgl_winwait()</td>
<td>2306</td>
</tr>
<tr>
<td>winexec()</td>
<td>2314</td>
</tr>
<tr>
<td>winexecwait()</td>
<td>2315</td>
</tr>
<tr>
<td>winopendir()</td>
<td>2313</td>
</tr>
<tr>
<td>winopenfile()</td>
<td>2313</td>
</tr>
<tr>
<td>winsavefile()</td>
<td>2314</td>
</tr>
<tr>
<td>winshellexec()</td>
<td>2315</td>
</tr>
<tr>
<td><strong>utility functions - fglgallery</strong></td>
<td></td>
</tr>
<tr>
<td>addImage()</td>
<td>2324</td>
</tr>
<tr>
<td>clean()</td>
<td>2325</td>
</tr>
<tr>
<td>create()</td>
<td>2326</td>
</tr>
<tr>
<td>deleteImages()</td>
<td>2326</td>
</tr>
<tr>
<td>destroy()</td>
<td>2327</td>
</tr>
<tr>
<td>display()</td>
<td>2327</td>
</tr>
<tr>
<td>finalizer()</td>
<td>2328</td>
</tr>
<tr>
<td>flush()</td>
<td>2328</td>
</tr>
<tr>
<td>getimagecount()</td>
<td>2329</td>
</tr>
<tr>
<td>getPath()</td>
<td>2329</td>
</tr>
<tr>
<td>getTitle()</td>
<td>2330</td>
</tr>
<tr>
<td>initialize()</td>
<td>2331</td>
</tr>
<tr>
<td>setAspectRatio()</td>
<td>2332</td>
</tr>
<tr>
<td>setMultipleSelection()</td>
<td>2331</td>
</tr>
<tr>
<td>t_struct_value_type</td>
<td>2324</td>
</tr>
<tr>
<td><strong>utility functions - fglsvgcanvas</strong></td>
<td></td>
</tr>
<tr>
<td>animateTransform()</td>
<td>2336</td>
</tr>
<tr>
<td>circle()</td>
<td>2336</td>
</tr>
<tr>
<td>clean()</td>
<td>2337</td>
</tr>
<tr>
<td>clipPath_rect()</td>
<td>2337</td>
</tr>
<tr>
<td>color_shade()</td>
<td>2338</td>
</tr>
<tr>
<td>color_tint()</td>
<td>2338</td>
</tr>
<tr>
<td>create()</td>
<td>2339</td>
</tr>
<tr>
<td>createChars()</td>
<td>2340</td>
</tr>
<tr>
<td>createElement()</td>
<td>2340</td>
</tr>
<tr>
<td>def()</td>
<td>2341</td>
</tr>
<tr>
<td>destroy()</td>
<td>2342</td>
</tr>
<tr>
<td>display()</td>
<td>2342</td>
</tr>
<tr>
<td>ellipse()</td>
<td>2342</td>
</tr>
<tr>
<td>filter()</td>
<td>2343</td>
</tr>
<tr>
<td>finalize()</td>
<td>2344</td>
</tr>
<tr>
<td>gt()</td>
<td>2344</td>
</tr>
<tr>
<td>getBBBox()</td>
<td>2345</td>
</tr>
<tr>
<td>getItemId()</td>
<td>2345</td>
</tr>
<tr>
<td>image()</td>
<td>2346</td>
</tr>
<tr>
<td>initialize()</td>
<td>2346</td>
</tr>
<tr>
<td>line()</td>
<td>2347</td>
</tr>
<tr>
<td>linearGradient()</td>
<td>2347</td>
</tr>
<tr>
<td>marker()</td>
<td>2348</td>
</tr>
<tr>
<td>mask()</td>
<td>2350</td>
</tr>
<tr>
<td>nl_to_tspan()</td>
<td>2350</td>
</tr>
<tr>
<td>path()</td>
<td>2351</td>
</tr>
<tr>
<td>pattern()</td>
<td>2351</td>
</tr>
<tr>
<td>polygon()</td>
<td>2352</td>
</tr>
<tr>
<td>polyline()</td>
<td>2353</td>
</tr>
<tr>
<td>radialGradient()</td>
<td>2353</td>
</tr>
<tr>
<td>rect()</td>
<td>2354</td>
</tr>
<tr>
<td>removeElement()</td>
<td>2355</td>
</tr>
<tr>
<td>setAttributes()</td>
<td>2355</td>
</tr>
<tr>
<td>setCurrent()</td>
<td>2356</td>
</tr>
<tr>
<td>setRootSVGAttributes()</td>
<td>2356</td>
</tr>
<tr>
<td>stop()</td>
<td>2357</td>
</tr>
<tr>
<td>styleAttributeList()</td>
<td>2358</td>
</tr>
<tr>
<td>styleDefinition()</td>
<td>2359</td>
</tr>
<tr>
<td>styleList()</td>
<td>2359</td>
</tr>
<tr>
<td>svg()</td>
<td>2360</td>
</tr>
<tr>
<td>SVGATT_ constants</td>
<td>2335</td>
</tr>
<tr>
<td>t_svg_rect type</td>
<td>2335</td>
</tr>
<tr>
<td>text_path()</td>
<td>2361</td>
</tr>
<tr>
<td>text_tref()</td>
<td>2362</td>
</tr>
<tr>
<td>text()</td>
<td>2361</td>
</tr>
<tr>
<td>title()</td>
<td>2363</td>
</tr>
<tr>
<td>tspan()</td>
<td>2363</td>
</tr>
<tr>
<td>url()</td>
<td>2364</td>
</tr>
</tbody>
</table>
utility functions - getopt
  use()  2365
  constants  2370
  copyArguments()  2371
  displayUsage()  2371
  getMoreArgument()  2372
  getMoreArgumentCount()  2372
  getopt type  2370
  getopt()  2373
  GetoptOptions type  2369
  initDefault()  2375
  initialize()  2375
  invalidOptionSeen()  2376
  isEof()  2377
  isSuccess()  2377
utility functions - getopt usage  2367
utility functions - vcard
  format_person()  2319
  scan_address()  2320
  scan_email()  2320
  scan_name()  2321
  scan_person()  2321
  scan_phone()  2322
  VCAddress type  2317
  VCName type  2318
  VPerson type  2318

V

VALIDATE
  dialog programming  1762
values
  passed as parameter  494
  returned from functions  499
variables
  attributes  398
  database column types  394
  default values  395
  initializers  396
  scope  392
  structured  393
  types  394
vendor specific parameter
  database  672
version
  upgrade notes  48

W

wait
  SLEEP statement  388
Web Component
  basics  1922
  debug  1928
  layout  1924
Web Component - layout
  grid  1924
  HTML layout  1926
  stack  1925
  SVG  1926

web service
  RESTful client
    endpoint record  3417
    set server endpoint  3418
web services
  BDL/XML mapping
    Attributes list  3524
    Data type attributes  3523
    Default type mapping  3523
    Serialization options  3499
  BDL/XML type attributes
    XSDAnySimpleType  3526
    XSDAnyURI  3526
    XSDBase64Binary  3526
    XSDBoolean  3526
    XSDByte  3527
    XSDDate  3527
    XSDDateTime  3527
    XSDDecimal  3527
    XSDDouble  3527
    XSDDateTime  3532
    XSDDateTime  3533
    XSDDuration  3528
    XSDEntity  3528
    XSDEntity  3528
    XSDFloat  3528
    XSDDouble  3528
    XSDDay  3528
    XSDDate  3528
    XSDMonth  3529
    XSDDayOfYear  3529
    XSDMonthOfYear  3529
    XSDHexBinary  3529
    XSDID  3530
    XSDIDREF  3530
    XSDIDREFS  3530
    XSDInt  3530
    XSDInteger  3530
    XSDLanguage  3530
    XSDLong  3530
    XSDMonthDay  3529
    XSDName  3531
    XSDNCName  3531
    XSDNegativeInteger  3531
    XSDNMTOKEN  3531
    XSDNMTOKENS  3531
    XSDNonNegativeInteger  3531
    XSDNonPositiveInteger  3531
    XSDNormalizedString  3532
    XSDNotation  3532
    XSDPositiveInteger  3532
    XSDQName  3532
    XSDShort  3532
    XSDString  3532
    XSDUnsignedByte  3533
    XSDUnsignedInt  3533
    XSDUnsignedLong  3533
    XSDUnsignedShort  3533
  BDL/XML type attributes attributes
    XSDAnyType  3526
  client application
    authenticate.idauth.login (FGLPROFILE)  3341, 3429
    authenticate.idauth.password (FGLPROFILE)  3341, 3429
HTTP proxy 3340, 3428
ip.global.v6.interface.id (FGLPROFILE) 3340, 3428
ip.global.v6.interface.name (FGLPROFILE) 3340, 3428
ip.global.version (FGLPROFILE) 3340, 3428
IPv6 3340, 3428
proxy.http.authenticate (FGLPROFILE) 3342, 3430
proxy.http.list (FGLPROFILE) 3340, 3428
proxy.http.location (FGLPROFILE) 3340, 3428
server URL mapping 3334
steps 3321
concepts 3277
configuration
apache HTTPS security 3460
Customizing XML serialization
Attributes list 3538
how to
calculate a hash 3457
SOAP security 3361
howto
SOA .NET API 3450
SOA Java API 3445
HTTP basic authentication 3341, 3429
HTTP proxy authentication 3342, 3430
missing certificate problems 3311
mobile apps 3586
OpenSSL certificates 3313
OpenSSL tool 3313
RESTful
create resource with WSPost 3378
define functions 3376
delete resource with WSDelete 3379
download file multipart 3396
generate OpenAPI file 3406
get resource with WSGet 3377
getting started 3375
handle errors 3399
handling expected errors 3400
high-level framework 3375
low-level APIs 3418
publish service 3404
register service with IdP 3403
request body 3383
resource operations 3377
response body 3384
retrieve HTTP headers 3385
security 3402
security with WSScope 3402
service information 3404
unexpected error 3401
update resource with WSPost 3379
upload file 3388
upload file multipart 3397
versioning 3409
versioning with WSPatch 3410
web service design 3574
WSMedia 3393
WSParam WSPatch 3380
WSQuery W$Header W$Cookie 3382
XML format 3394
RESTful client
call service functions 3413
call stub functions 3415
handle server errors 3416
main code 3413
openapi specification 3410
set server endpoint 3417
troubleshooting stub file 3413
security 3299
server application
deployment (GAS) 3459
DOC style 3358
enable MTOM 3354
RPC style 3358
web services style 3358
server authority problems 3310
steps to writing a RESTful client application 3418
troubleshooting certificate issues 3310
upgrade
Version 2.10 to 2.11 WSDL format 3359
V1.3 API
fgl_ws_server_generateWSDL 3558
fgl_ws_server_getFault 3560
fgl_ws_server_process 3559
fgl_ws_server_publishFunction 3557
fgl_ws_server_setFault 3560
fgl_ws_server_setNamespace 3556
fgl_ws_server_start 3557
V1.3 API options
http_invoketimeout 3562
soap_ignoretimezone 3562
soap_usetypedefinition 3562
tcp_connectiontimeout 3562
wsdl_arraysize 3562
wsdl_decimalsize 3562
wsdl_stringsize 3562
XML facet constraint
Attributes list 3534
XML facet constraint attributes
XSDAnySimpleType 3534
XSDEnumeration 3535
XSDFractionDigits 3538
XSDMaxExclusive 3537
XSDMaxInclusive 3537
XSDMaxLength 3535
XSDMinExclusive 3537
XSDMinInclusive 3536
XSDMinLength 3535
XSDPattern 3536
XSTotalDigits 3537
XSDWhiteSpace 3536
XML serialization
dynamic arrays 3553
XML serialization attributes
XMLAll 3543
XMLAny 3547
XMLAnyAttribute 3548
XMLAttribute 3543
XMLAttributeNamespace 3551
XMLBase 3543
XMLChoice 3544
XMLElement 3541
XMLElementNamespace 3551
XMLElementNillable 3542
XMLList 3546
XMLName 3548
XMLNamespace 3549
XMLNullable 3540
XMLOptimizedContent 3552
XMLOptional 3539
XMLSelector 3547
XMLSequence 3545
XMLSimpleContent 3546
XMLType 3549
XMLTypenamespace 3550
XSComplexType 3546
XSTypename 3550
XSTypenamespace 3550

Web services
devsupport 3276
demos 3275
known issues
  RPC style 3277
legal notices 3277
platform notes
  AIX 3276
  GMA / Android 3277
  GMI / iOS 3276
samples 3275
upgrade 3366
web services - fglwsdl
  generate type definitions 3564
Web services API functions, version 1.3
configuration APIs
  fgl_ws_getOption() 3562
  fgl_ws_setOption() 3561
server APIs 3555

what's new
  upgrade guides 109
  Version 1.10 108
  Version 1.20 105
  Version 1.30 101
  Version 1.31 100
  Version 1.32 100
  Version 1.33 99
  Version 2.00 95
  Version 2.01 95
  Version 2.02 94
  Version 2.10 91
  Version 2.11 88
  Version 2.20 85
  Version 2.21 80
  Version 2.30 78
  Version 2.32 77
  Version 2.40 73
  Version 2.41 73
  Version 2.50 69
  Version 2.51 66
  Version 3.00 61
  Version 3.10 54
  Version 3.20 48
version list 48

WHERE CURRENT OF
  SQL portability 631
WITH HOLD + FOR UPDATE
  SQL portability 631

WITHOUT DEFAULTS
dialog programming 1722
WITHOUT DEFAULTS attribute 1603, 1663
WS client application configuration
  create X.509 certificate 3306, 3336, 3424
  HTTPS root certificate authority 3306, 3336, 3424
WSHelper
  definitions 3500
  library 3499
  WSQueryType 3500
  WSServerCookiesType 3500

X

XML output
  reports 2027
xml.serializer.setOption
  xml_ignoretimezone 3046
  xml_ignoreunknownattributes 3046
  xml_ignoreunknownelements 3046
  xml_usetypedefinition 3046
  xml_useutctime 3046
  xs_processcontents 3046